The semantic core of multi-agent interactions: an operational model[1]

Sergio Saugar Juan Manuel Serrano

Department of Computing, University Rey Juan Carlos
C/Tulipán s/n, 28937, Móstoles (Madrid), Spain
{sergio.saugar,juanmanuel.serrano}@urjc.es

Abstract

The social stance advocated by institutional frameworks and most multi-agent system (MAS) methodologies has resulted in a wide catalogue of software abstractions to encapsulate multi-agent interactions. This paper attempts to expose a possible semantic core underlying the wide spectrum of interaction types between autonomous, social and situated software components. In the realm of software architectures, this core has been formalised as an operational model of social connectors describing both the structure and dynamics of multi-agent interactions. This formal model is aimed as the first steps towards the abstract machine of a language to program multi-agent interactions.

1 Introduction

The suitability of agent-based computing to manage the complex patterns of interactions naturally occurring in the development of large scale, open systems, has become one of its major assets over the last few years [27, 28]. Particularly, the organizational or social stance advocated in various degrees by most multi-agent system (MAS) methodologies, provides an excellent basis to deal with the complexity and dynamism of the interactions among system components [28]. This approach has resulted in a wide spectrum of organizational and communicative abstractions, such as institutions, organizations, groups, communicative interactions, etc., to effectively model the interaction space of MAS [11, 13, 26, 28].

Still, software engineering has witnessed an increasing attention to component interactions from different research fields: coordination models and languages [7], and software architectures [1], amongst others, offer alternative analysis at different levels of abstractions. From an architectural perspective, for instance, component interactions are explicated in terms of software *connectors*: explicit semantic entities characterized in terms of the participant roles and protocol regulating the interaction. Component & Connector architectural styles, such as Pipe& Filter, Peer-to-Peer, or Client-Server, define different types of connectors and computational models in general.

The main hypothesis motivating this work is that the variety of organizational and communicative abstractions found in the current literature share a common semantic core. This paper attempts to elucidate this common core from an architectural point of view. Particularly, a connector-based semantics of social interactions among autonomous and situated agents is put forward, which attempts to identify the essential structure and dynamics underlying multi-agent interactions.

The paper is structured as follows: first, the major entities and relationships which constitute the structure of social interactions is introduced. Next, the dynamics of social interactions will show how these entities and relationships evolve. The model will be formalised using Structural Operational Semantics [23]. Then, the third section attempts to test the productivity of the proposed semantics by analyzing common organizational and communicative abstractions of MAS. Last, relevant work in the literature is discussed with respect to the proposal, limitations are addressed, and current and future work is described.

[1]Research sponsored by the Spanish Ministry of Science and Education (MEC), project TIN2006-15455-C03-03

2 Social interaction structure

From an architectural point of view, interactions between software components are represented by means of connectors: first-class entities defined on the basis of the different roles displayed by software components and the protocols that regulate the behaviour of these participating components [1]. The analysis of social interactions introduced in this section refines this generic model in several respects, attending to the common features of software agents. Firstly, in accordance with the autonomy and sociality of agents, the specification of social interaction protocols will rely in *normative concepts* such as permissions, obligations and empowerments. Secondly, by considering situatedness, social interactions will take place in an *environment* consisting of resources which the participants create and manipulate. Unlike agents, resources are software components lacking autonomy, so that their state may be externally controlled by agents or other resources. Besides *interactions*, *resources*, *agents* and *protocols*, two other kinds of entities are of major relevance in our analysis: *social actions*, which represent the way in which agents alter the environmental and social state of the interaction; and *events*, which represent the changes in the social interaction resulting from the performance of social actions or the activity of the environmental resources.

In the following, we describe the basic entities involved in social interactions. Each kind of entity T will be specified as a labeled record $T \triangleq \langle l_1 : T_1, \dots l_n : T_n \rangle$, possibly followed by a number of invariants, definitions, and the social actions affecting their state. Instances or values v of a record type T will be represented as $v = \langle v_1, \dots, v_n \rangle \ : \ T$. The type $Set[T]$ represents a collection of values drawn from type T. The type $Queue[T]$ represent a queue of values $v : T$ waiting to be processed. The value v in the expression $[v|\_] : Queue[T]$ represents the head of the queue. The type $Enum\langle v_1, \dots, v_n \rangle$ represents an enumeration type whose values are v_1, ..., v_n. Given some value $v \ : \ T$, the term v^l refers to the value of the field l of a record type T. Given some labels l_1, l_2, ..., the expression $v^{l_1, l_2, \cdots}$ is syntactic sugar for $((v^{l_1})^{l_2})\dots$. The special term *nil* will be used to represent the absence of proper value for an optional field, so that $v^l = nil$ will be true in those cases and false otherwise. The formal model will be illustrated with several examples drawn from the design of a virtual organization to aid in the management of university courses.

2.1 Social Interactions

Social interactions shall be considered as composite connectors [20], structured in terms of a tree of nested sub-interactions. Let's consider an interaction representing a university *course* (e.g. on data structures). On the one hand, this interaction is actually a complex one, made up of lower-level interactions. For instance, within the scope of the course agents will participate in *programming assignment groups*, *lectures*, *tutoring meetings*, *examinations* and so on. Assignment groups, in turn, may hold a number of *assignment submissions* and *test requests* interactions. A test request may also be regarded as a complex interaction, ultimately decomposed in the atomic, or bottom-level interactions represented by communicative actions (CAs) (e.g. *request*, *agree*, *refuse*, ...). On the other hand, courses are run within the scope of a particular *degree* (e.g. computer science), a higher-level interaction. Traversing upwards from a degree to its ancestors, we find its *faculty*, the *university* and, finally, the multi-agent *community* or agent society. The community is thus the top-level interaction which subsumes any other kind of multi-agent interaction[2].

The organizational and communicative interaction types identified above clearly differ in many ways. However, we may identify four major components in all of them: the participating agents, the resources that agents manipulate, the social protocol regulating the agent activities and the sub-interaction space. Accordingly, we may specify the type $\mathcal{I}$ of *social interactions*, ranged over by the meta-variable i, as follows:

$$\begin{aligned}
\mathcal{I} \ \triangleq \ & \langle state : \mathcal{S}_{\mathcal{I}}, ini : \mathcal{A}, mem : Set[\mathcal{A}], env : Set[\mathcal{R}], \\
& \ sub : Set[\mathcal{I}], prot : \mathcal{P}, ch : \mathcal{CH} \rangle \\
\textbf{def.}: \ & (1) \ \ i^{context} = i_1 \Leftrightarrow i \in i_1^{sub} \\
\textbf{inv.}: \ & (2) \ \ i^{ini} = nil \Leftrightarrow i^{context} = nil \\
\textbf{act.}: \ & setUp, join, leave, create, destroy, close
\end{aligned}$$

[2]In the context of this application, a one-to-one mapping between human users and software components attached to the community as agents would be a right choice.

where the *mem*ber and *env*ironment fields represents the agents ($\mathcal{A}$) and local resources ($\mathcal{R}$) participating in the interaction; the *sub*-interaction field, its set of inner interactions; and the *proto*col field the rules that govern the interaction ($\mathcal{P}$). The event *ch*annel, to be described in the next section, allows the dispatching of local events to external interactions. The *context* of some interaction is defined as its super-interaction (def. 1), so that the context of the top-level interaction is *nil*.

The type $\mathcal{S_I} \triangleq Enum\langle open, closing, closed \rangle$ represents the possible execution *states* of the interaction. Any interaction, but the top-level one, is *set up* within the context of another interaction by an *ini*tiator agent. The initiator is thus a mandatory feature for any interaction different to the community (inv. 2). The life-cycle of the interaction begins in the *open* state. Its sets of agent and resource participants, initially empty, varies as agents *join* and *leave* the interaction, and as they *create* and *destroy* resources from its local environment. Eventually, the interaction may come to an end (according to the protocol's rules), or be explicitly *closed* by some agent, thus prematurely disabling the activity of its participants. The transient *closing* state will be described in the next section.

2.2 Agents

Components engage as agents in social interactions with the *purpose* of achieving something. The purpose declared by some agent when it joins an interaction shall be regarded as the institutional goal that it purports to satisfy within that context[3]. The types of agents participating in a given interaction are primarily identified from their purposes. For instance, *students* are those agents participating in a course who purport to obtain a certificate in the course's subject. Other members of the course include *lecturers* and *teaching assistants*.

The type $\mathcal{A}$ of agents, ranged over by meta-variable a, is defined as follows:

$$\begin{array}{lll} \mathcal{A} & \triangleq & \langle state : \mathcal{S_A}, player : \mathcal{A}, purp : \mathcal{F}, att : Queue[\mathcal{ACT}], \\ & & \quad ev : Queue[\mathcal{E}], obl : Set[\mathcal{O}] \rangle \\ & \textbf{def.}: & (3)\ \ a^{context} = i \Leftrightarrow a \in i^{mem} \\ & & (4)\ \ i \in a^{partIn} \Leftrightarrow a_1 \in i^{mem} \ \wedge \ a_1^{player} = a \\ & \textbf{inv.}: & (5)\ \ a^{player} = nil \Leftrightarrow a^{context, context} = nil \\ & \textbf{act.}: & see \end{array}$$

where the *purp*ose is represented as a well-formed boolean formula, of a generic type $\mathcal{F}$, which evaluates to *true* if the purpose is satisfied and *false* otherwise. The context of some agent is defined as the interaction in which it participates (def. 3).

The type $\mathcal{S_A} \triangleq Enum\langle playing, leaving, succ, unsuc \rangle$ represents the execution *state* of the agent. Its life-cycle begins in the *playing* state when its *player* agent joins the interaction. Any agent who is not a member of the top-level interaction is played by another one (inv. 5)[4]. The derived *partIn* feature represents the interactions in which the agent is playing some member role (def. 4)[5]. An agent may play roles at interactions within or outside the scope of its context. For instance, students of a course are played by student agents belonging to the (undergraduate) degree, whereas lecturers may be played by teachers of a given department and the assistant role may be played by students of a Ph.D degree (both, the department and the Ph.D. degrees, are modelled as sub-interactions of the faculty).

Components will normally attempt to perform different social actions (e.g. to *set up* sub-interactions) in order to satisfy their purposes within some interaction. Moreover, components needs to be aware of the current state of the interaction, so that they will also be capable of observing certain events from the interaction. Both, the visibility of the interaction and the attempts of members, are subject to the rules governing the interaction. The *att*empts and *ev*ents fields of the agent structure represents the queues of attempts to execute some social actions ($\mathcal{ACT}$), and the events ($\mathcal{E}$) received by the agent which has not been observed yet. An agent may update its event

[3]Thus, it may or may not correspond to actual internal "goals" or "intentions" of the component.

[4]The player of a top-level agent role is actually its software component.

[5]Free variables in the antecedents/consequents of implications shall be understood as universally/existentially quantified.

queue by *seeing* the state of some entity of the community. The last field of the structure represents the *obl*igations ($\mathcal{O}$) of agents, to be described later.

Eventually, the participation of some agent in the interaction will be over. This may either happen when certain conditions are met (specified by the protocol rules), or when the agent takes the explicit "decision" of leaving the interaction. In either case, the final state of the agent will be *successful* if its purpose was satisfied; *unsuccessful* otherwise. The transient *leaving* state will be described in the next section.

2.3 Resources

Resources are software components which may represent different types of non-autonomous informational or computational entities. For instance, *objectives*, *topics*, *assignments*, *grades* and *exams* are different kinds of informational resources created by lecturers and assistants in the context of the course interaction. Students may also create *programs* to satisfy the requirements of some assignment. Other types of computational resources put at the disposal of students by teachers include *compilers* and *interpreters.*

The type $\mathcal{R}$ of resources, ranged over by meta-variable r, can be specified by the following labeled record:

$$\begin{aligned}\mathcal{R} \;\triangleq\; & \langle cr : \mathcal{A}, owners : Set[\mathcal{A}], op : Set[\mathcal{OP}]\rangle \\ \textbf{def.}: \;& (6)\;\; r^{context} = i \Leftrightarrow r \in i^{env} \\ \textbf{act.}: \;& take, share, give, invoke\end{aligned}$$

Essentially, resources can be regarded as *objects* deployed in a social setting. At least, this means that resources are created, accessed and manipulated by agents in a social interaction *context* (def. 7), according to the rules specified by its protocol. The mandatory feature *cr*eator represents the agent who created this resource. Moreover, resources may have *owners.* The ownership relationship between members and resources is considered as a normative device aimed at the simplification of the protocol's rules that govern the interaction of agents and the environment. Members may gain ownership of some resource by *taking* it, and grant ownership to other agents by *giving* or *sharing* their own properties. For instance, the ownership of programs may be shared by several students if the assignment can be performed by groups of two or more students.

The last *op*erations feature represents the interface of the resource, consisting of a set of operations. A resource is structured around several public operations that participants may *invoke*, in accordance to the rules specified by the interaction's protocol. The set of operations of a resource makes up its interface. Resources that own a thread of control are called *active.*

2.4 Protocols

The protocol of some interaction is made up of the rules which govern its overall state. The present specification abstracts away the particular formalism used to specify these rules, and focuses instead in several requirements concerning the structure and interface of protocols. Accordingly, the type $\mathcal{P}$ of protocols, ranged over by meta-variable p, is defined as follows[6]:

$$\begin{aligned}\mathcal{P} \;\triangleq\; & \langle emp : \mathcal{A} \times \mathcal{ACT} \rightarrow Boolean, \\ & perm : \mathcal{A} \times \mathcal{ACT} \rightarrow Boolean, \\ & vis : \mathcal{A} \times \mathcal{E} \rightarrow Boolean, \\ & obl : \mathcal{A} \rightarrow Set[\mathcal{O}] \times Set[\mathcal{E}], \\ & over : \mathcal{A} \rightarrow Boolean, \\ & finish : \;\rightarrow Boolean\rangle \\ \textbf{def.}: \;& (7)\;\; p^{context} = i \Leftrightarrow p = i^{prot} \\ \textbf{inv.}: \;& (8)\;\; p^{finish}() \;\wedge\; s \in p^{context,sub} \Rightarrow s^{prot,finish}() \\ & (9)\;\; p^{finish}() \;\wedge\; a \in p^{context,mem} \Rightarrow p^{over}(a) \\ & (10)\;\; p^{over}(a) \;\wedge\; a_i^{player} = a \Rightarrow a_i^{context,prot,over}(a_i)\end{aligned}$$

[6]Additional social actions, such as *permit*, *forbid*, *empower*, etc., to update the rules of protocols are yet to be identified in future work.

We demand from protocols four major kinds of functions. Firstly, protocols shall include rules to identify the *empowerments* and *permissions* of any agent attempting to alter the state of the interaction (e.g. its members, the environment, etc.) through the execution of some social action (e.g. join, create, etc.). Empowerments shall be regarded as the institutional capabilities which some agent possesses in order to satisfy its purpose. Corresponding rules, encapsulated by the *emp*owered function field, shall allow to determine whether some agent is capable to perform a given action over the interaction[7]. Empowerments may only be exercised under certain circumstances – that permissions specify. Permission rules shall allow to determine whether the attempt of an empowered agent to perform some particular action is satisfied or not (cf. *perm*itted field). For instance, the course's protocol specifies that the agents empowered to *join* the interaction as students are those students of the degree who have payed the fee established for the course's subject, and own the certificates corresponding to its prerequisite subjects. Permission rules, in turn, specifies that those students may only join the course in its admission stage. Hence, even if some student has paid the fee, the attempt to join the course will fail if the course has not entered the corresponding stage[8].

Secondly, the protocol shall allow to specify *monitoring* rules which establishes the visibility of incoming events for each of its members. Corresponding rules shall establish whether some event must be notified to a specified agent. For instance, this functionality is exploited by teachers in order to monitor the enrollment of students to the course.

Thirdly, protocols shall allow to determine the *obligations* of its members. Obligations represent a normative device of social enforcement, fully compatible with the autonomy of agents, used to bias their behaviour in a certain direction. These kinds of rules shall allow to determine whether some agent must perform an action of a given type, as well as if some obligation was fulfilled, violated or needs to be revoked. The function *obl*igations of the protocol structure thus updates the set of obligations of the specified member agent. Moreover, it returns a collection of events representing the changes in the obligation set. For instance, the course's protocol establishes that teachers must create the course's objectives and topics in the *preparation* stage.

Last, the protocol shall allow to control the *state of the interaction* as well as the *states of its members*. Corresponding rules identify the conditions under which some interaction will be automatically finished, and whether the participation of some agent in the interaction will be automatically over. Thus, the function field *finish* returns *true* if the regulated interaction must finish its execution. If so happens, a well-defined set of protocols must ensure that its sub-interactions and members are finished as well (inv. 8,9). Similarly, the function *over* returns *true* if the participation of the specified member must be over. Well-formed protocols must ensure the consistency between these functions across playing roles (inv. 10). For instance, the course's protocol establishes that the participation of students is over when they gain ownership of the course's certificate or the chances to get it are exhausted. It also establishes that the course must be finished when the admission stage has passed and all the students finished their participation.

3 Social interaction dynamics

The dynamics of the multi-agent community is influenced by the external actions executed by software components and the rules governing their interactions. This section focuses on the dynamics resulting from a particular kind of external action: the *attempt* of some component to execute a given (internal) social action, qua agent attached to the community. The description of other external actions concerning agents (e.g. *observe* the events from its event queue, *enter* or *exit* from the community) and resources (e.g. a timer resource may *signal* the pass of time) will be skipped.

The processing of some attempt may give raise to changes in the scope of the target interaction, such as the instantiation of new participants (agents or resources) or the setting up of new sub-interactions. These resulting events may cause further changes in the state of other interactions (the target one included), namely, in its execution state as well as in the execution state, obligations and visibility of their members. This section will also describe the way in which these

[7]Protocol's functions may check the current and past state of different interactions to compute their results.

[8]The *hasPaidFee* relationship between (degree) *students* and *subject* resources is represented by an additional, application-dependent field of the agent structure for this kind of roles. Similarly, the *course*'s stage is an additional field of the structure for course interactions. The generic types $\mathcal{I}$, $\mathcal{A}$, $\mathcal{R}$ and $\mathcal{P}$ are thus extendable.

events are processed. The resulting dynamics described bellow allows for social actions and events corresponding to different agents and interactions to be processed simultaneously. Due to lack of space, we only include some of the operational rules that formalise their execution semantics.

3.1 Attempt processing

An attempt is defined by the structure $\mathcal{ATT} \triangleq \langle perf : \mathcal{A}, act : \mathcal{ACT}\rangle$, where the *perf*ormer represents the agent in charge of executing the specified *act*ion. This action is intended to alter the state of some *target* interaction (possibly, the performer's context itself), and notify a collection of *add*ressees of the changes resulting from a successful execution. Accordingly, the type $\mathcal{ACT}$ of (internal) social actions, ranged over by meta-variable α, is specified as follows:

$$\begin{array}{lll} \mathcal{ACT} & \triangleq & \langle state : \mathcal{S}_{\mathcal{ACT}}, target : \mathcal{I}, ev : \mathcal{E}, add : Set[\mathcal{A}]\rangle \\ & \textbf{def.}: & (11)\ \ \alpha^{perf} = a \Leftrightarrow \alpha \in a^{att} \end{array}$$

where: the *perf*ormer is formally defined as the agent who stores the action in its queue of attempts; the *ev*ent field represents the changes caused by a successful or unsuccessful execution of the action; and the *stage* field represents the current phase of processing. This process goes through four major phases, as specified by the enumeration type $\mathcal{S}_{\mathcal{ACT}} \triangleq Enum\langle emp, perm, exec, disp\rangle$: *empowerment checking*, *permission checking*, *action execution* and *addressee dispatching*, described in the sequel.

3.1.1 Empowerment checking

The post-condition of an attempt consists of inserting the action in the queue of attempts of the specified performer. As rule 1 specifies, this will only be possible if the performer is *empowered* to execute that action according to the rules that govern the state of the target interaction. If this condition is not met, the attempt will simply be ignored. Moreover, the performer agent must be in the *playing* state (this pre-condition is also required for any rule concerning the processing of attempts). If these pre-conditions are satisfied the rule is fired and the processing of the action continues in the *permission checking* stage.

$$\frac{\epsilon = \langle a, \alpha\rangle : \mathcal{ATT} \ \wedge\ \alpha^{target,prot,emp}(a,\alpha)}{a = \langle playing, \_, \_, q_{ACT}, \_, \_\rangle \xrightarrow{\epsilon} \langle playing, \_, \_, q'_{ACT}, \_, \_\rangle} \tag{1}$$

$$\begin{array}{llcl} Where: & (\alpha')^{state} & = & perm \\ & (q'_{ACT}) & = & push(\alpha', q_{ACT}) \end{array}$$

3.1.2 Permissions checking

The processing of the action resumes when the possible preceding actions in the performer's queue of attempts are fully processed and removed from the queue. Moreover, there should be no pending events to be processed in the interaction, for these events may cause the member or the interaction to be finished (as will be shortly explained in the next sub-section). If these conditions are met the permissions to execute the given action (and notify the specified addressees) are checked. If the protocol of the target interaction grants permission, the processing of the attempt moves to the *action execution* stage (rule 2). Otherwise, the action is discharged and removed from the queue. Unlike unempowered attempts, a forbidden one will cause an event to be generated and transfered to the event channel for further processing.

$$\frac{\alpha^{state} = perm \ \wedge\ a^{context,ch,in} = \emptyset \ \wedge\ \alpha^{target,prot,perm}(a,\alpha)}{a = \langle playing, \_, \_, [\alpha|\_], \_, \_\rangle \longrightarrow \langle playing, \_, \_, [\alpha'|\_], \_, \_\rangle} \tag{2}$$

$$Where: \ (\alpha')^{state} \ = \ exec$$

3.1.3 Action execution

The transitions fired in this stage are classified according to the different types of actions to be executed. The intended effects of some actions may directly be achieved in a single step, while others will required an indirect approach and possibly several execution steps. Actions of the first kind are “constructive” ones such as *set up* and *join*. The second group of actions include those, such as *close* and *leave*, whose effects are indirectly achieved by updating the interaction protocol.

As an example of constructive action, let’s consider the execution of a *set up* action, whose type is defined as follows[9]:

$$\text{SetUp} \quad \triangleq \quad \mathcal{ACT}\langle new : \mathcal{I}\rangle$$

$$\begin{array}{lll}\textbf{inv.}: & (12) & \alpha^{new,mem} = \alpha^{new,res} = \alpha^{new,sub} = \emptyset \\ & (13) & \alpha^{new,state} = open\end{array}$$

where the *new* field represents the new interaction to be initiated. Its sets of participants (agents and resources) and sub-interactions must be empty (inv. 12) and its state must be *open* (inv. 13). The setting up of the new interaction may thus affect its protocol and possible application-dependent fields (e.g. the *subject* of a course interaction). According to rule 3, the outcome of the execution is threefold: firstly, the performer’s attempt queue is updated so that the executing action moves to the next *addressee dispatching* stage and an event is placed in its corresponding auxiliary field; secondly, the new interaction is added to the target’s set of sub-interactions (moreover, its initiator field is set to the performer agent); last, the event representing this change is inserted in the output port of the target’s event channel.

$$\frac{\alpha^{state} = exec \ \wedge \ \alpha : \text{SetUp} \ \wedge \ \alpha^{\text{sub}} = \text{i}}{\begin{array}{c}\langle playing, \_, \_, [\alpha|\_], \_, \_\rangle \longrightarrow \langle playing, \_, \_, [\alpha'|\_], \_, \_\rangle \\ \alpha^{target} = \langle open, \_, \_, \_, \_, s_I, c\rangle \longrightarrow \langle open, \_, \_, \_, \_, s_I \cup i', c'\rangle\end{array}} \tag{3}$$

$$\begin{array}{rcl} Where: \ (\alpha')^{state} & = & disp \\ (\alpha')^{ev} & = & sub(\alpha^{target}, i) \\ (i')^{ini} & = & \alpha^{perf} \\ (c')^{out} & = & insert(sub(\alpha^{target}, i), c^{out}) \end{array}$$

Let’s consider now the case of a *close* action. This action represents an attempt by the performer to force some interaction to finish, thus bypassing its current protocol rules (those concerning the *finish* function). The way to achieve this effect is to cause an update on the protocol so that the *finish* function returns true afterwards[10]. Accordingly, we may specify this type of action as follows:

$$\text{Close} \quad \triangleq \quad \mathcal{ACT}\langle upd : (\rightarrow Bool) \rightarrow (\rightarrow Bool)\rangle$$

$$\begin{array}{lll}\textbf{inv.}: & (14) & \alpha^{target,state} = open \\ & (15) & \alpha^{target,context} \neq nil \\ & (16) & \alpha^{upd}(\alpha^{target,prot,finish})()\end{array}$$

where the inherited *target* field represents the interaction to be closed (which must be open and different to the top-interaction, according to invariants 14 and 15) and the new *upd*ate field represents a proper higher-order function to update the target’s protocol (inv. 16). The transition which models the execution of this action, specified by rule 4, defines two effects in the target interaction: its protocol is updated and the event representing this change is inserted in its input channel. This event will actually trigger the closing process of the interaction as described in the next sub-section.

$$\frac{\alpha^{state} = exec \ \wedge \ \alpha : \text{Close}}{\begin{array}{c}\langle playing, \_, \_, [\alpha|\_], \_, \_\rangle \longrightarrow \langle playing, \_, \_, [\alpha'|\_], \_, \_\rangle \\ \alpha^{target} = \langle open, \_, \_, \_, \_, p, c\rangle \longrightarrow \langle open, \_, \_, \_, \_, p', c'\rangle\end{array}} \tag{4}$$

[9]The resulting type consists of the fields of the labelled record $\mathcal{ACT}$ extended plus an additional field *new*.

[10]This strategy is also followed in the definition of *leave* and may also be used in the definition of other types of actions such as *fire*, *permit*, *forbid*, etc.

$$\begin{array}{rcl} Where: \; (\alpha')^{state} & = & disp \\ (\alpha')^{ev} & = & finish(\alpha^{target}) \\ (p')^{finish} & = & \alpha^{upd}(p^{finish}) \\ (c')^{out} & = & insert(finish(\alpha^{target}), c^{out}) \end{array}$$

3.1.4 Addressee dispatching

In this last stage the specified addressees are notified of the changes caused by the action executed in the last phase (stored in the action's *ev* field). Transition 5 simply iterate over the addressees field until no agent is left. Once the set is empty the action is removed from the queue and the processing of the attempt is finished.

$$\frac{\alpha^{state} = disp \;\wedge\; a \in \alpha^{add}}{\begin{array}{c}\langle playing, \_, \_, [\alpha|\_], \_, \_\rangle \longrightarrow \langle playing, \_, \_, [\alpha'|\_], \_, \_\rangle \\ a = \langle playing, \_, \_, \_, q_E, \_\rangle \longrightarrow \langle playing, \_, \_, \_, q'_E, \_\rangle\end{array}} \tag{5}$$

$$\begin{array}{rcl} Where: \; (\alpha')^{add} & = & \alpha^{add} \setminus \{a\} \\ q'_E & = & insert(\alpha^{ev}, q_E) \end{array}$$

3.2 Event Processing

Events generated in the scope of some interaction (e.g. due to the successful execution of a social action or a forbidden attempt) must first be dispatched to those interactions whose state may be affected by those changes in order to trigger the corresponding updates[11]. Then, each potentially affected interaction have to process the external events to check its execution state and members for possible updates. These activities are encapsulated in the event channels of interactions. Channels, ranged over by meta-variable c, are defined by two input and output ports, according to the following definition:

$$\begin{array}{lcl} \mathcal{CH} & \triangleq & \langle out : Queue[\mathcal{E}], disp : \mathcal{E} \rightarrow Set[\mathcal{I}], int : Set[\mathcal{I}], \\ & & \;in : Queue[\mathcal{E}], agents : Set[\mathcal{A}]\rangle \\ & \mathbf{inv.}: & (17)\;\; c^{context} \in c^{disp}(finish(c^{context})) \\ & & (18)\;\; c^{context} \in c^{disp}(over(a)) \\ & & (19)\;\; c^{context,sub} \subseteq c^{disp}(closing(c^{context})) \\ & & (20)\;\; a^{partsIn} \subseteq c^{disp}(leaving(a)) \\ & & (21)\;\; c^{context} \subseteq c^{disp}(closed(i)) \\ & & (22)\;\; \{c^{context}, a^{player,context}\} \subseteq c^{disp}(left(a)) \end{array}$$

The output port of the channel is defined by the *out*, *disp* and *int* fields. This port stores the set of events originated within the interaction, and defines the function which determines the interactions to which these events will be dispatched. The constraints on the dispatching function will be explained bellow. The *int* auxiliary field represents those interactions identified by the dispatching function which have not being notified yet. The input port consists of the two last fields. It stores the incoming events dispatched to the interaction which has not been processed yet. The *agents* auxiliary field represents those agents whose state still needs to be checked for updates, given the current event been processed.

Accordingly, the processing of some event goes through three major stages: *interaction dispatching*, *interaction state update* and *members update*. The first one takes place in the interaction in which the event originated, whereas the second and third ones execute in separate control threads of the interactions to which the event was dispatched.

[11] Alternatively, we may have assumed that interactions are fully aware of any change in the multi-agent community. In this scenario, interactions would trigger themselves without requiring any explicit notification. On the contrary, we adhere to the more realistic assumption of limited awareness.

3.2.1 Interaction dispatching

The processing of some event stored in the output port is triggered when all its preceding events have been dispatched. As a first step, the auxiliary *int* field is initialised with the returned value of the dispatching function. This function shall identify the set of interactions (possibly, empty) whose state may be affected by the event[12]. This set may include the channel's interaction itself. Then, additional rules simply iterate over this collection (rule 6) and re-set its value to *nil* when all interactions have been notified.

$$\frac{c_s^{context,state} = i_d^{state} = open \;\wedge\; i_d \in s_I}{\begin{array}{c} c_s = \langle [e|\_], \_, s_I, \_, \_\rangle \longrightarrow \langle [e|\_], \_, s_I \setminus \{i_d\}, \_, \_\rangle \\ i_d^{ch} = \langle \_, \_, \_, q_i, \_\rangle \longrightarrow \langle \_, \_, \_, insert(e, q_i), \_\rangle \end{array}} \tag{6}$$

3.2.2 Interaction state update

Input port activity is triggered when a new event is received. Irrespective of the kind of incoming event, the first processing action is to check whether the channel's interaction must be finished. Thus, the dispatching of the *finish* event resulting from a close action (inv. 17) serves as a trigger of the closing procedure. The kind of *closing* events to be described bellow is similarly used as a coordination device.

If the interaction has not to be finished, the *part* field is initialised to its members and the process enters the *members update* stage. Otherwise, we can consider two possible scenarios. In the first one, the interaction has no members and no sub-interactions. In this case, the interaction can be inmediately closed down. As rule 7 shows, the interaction is closed, removed from the context's set of sub-interactions and a *closed* event is inserted in its output channel. According to invariant 21, this event is inserted in its input channel to allow for further treatment.

$$\frac{c^{in} \neq \emptyset \;\wedge\; c^{agents} = nil \;\wedge\; i \in s_I \;\wedge\; p^{finish}()}{\begin{array}{c} i = \langle \_, \_, \emptyset, \_, \emptyset, p, c\rangle \longrightarrow \langle closed, \_, \_, \_, \_, \_, \_\rangle \\ \langle \_, \_, \_, \_, s_I, \_, c\rangle \longrightarrow \langle \_, \_, \_, \_, s_I \setminus \{i\}, \_, c'\rangle \end{array}} \tag{7}$$

$$Where: \quad (c')^{out} \quad = \quad insert(closed(i), c^{out})$$

In the second scenario, the interaction has some member or sub-interaction. In this case, clean-up is required prior to the disposal of the interaction. As rule 8 shows, the interaction is moved to the transient *closing* state and a corresponding event is inserted in the output port. According to invariant 19, the closing event will be dispatched to every sub-interaction in order to activate its closing procedure (guaranteed by invariant 8). Moreover, the *agents* field is initialised so that the process goes on in the next *members update* stage. This stage will further initiate the *leaving* process of the members (according to invariant 9).

$$\frac{c^{in} \neq \emptyset \;\wedge\; c^{agents} = nil \;\wedge\; p^{finish}() \;\wedge\; (s_A \neq \emptyset \vee s_I \neq \emptyset)}{i = \langle open, \_, s_A, \_, s_I, p, c\rangle \longrightarrow \langle closing, \_, s_A, \_, s_I, p, c'\rangle} \tag{8}$$

$$\begin{array}{rcl} Where: \; (c')^{out} & = & insert(closing(i), c^{out}) \\ (c')^{agents} & = & s_A \end{array}$$

Eventually, every member will leave the interaction and every sub-interaction will be closed. Corresponding events will be received by the interaction (according to invariants 22 and 21) so that the conditions of the first scenario will hold.

[12] This is essentially determined by the protocol rules of these interactions. The way in which the dispatching function is initialised and updated is out of the scope of this paper.

3.2.3 Members update

The possible consequences in the overall state of members are three-fold: firstly, it may happen that the agent have to finished its participation in the interaction; secondly, the event may affect its normative state, causing the creation of some obligations or the satisfaction/violation/revoking of existing ones; last, the member may be one of those who must be notified of the given change (even if the event was the result of some action and the member was not part of the addressees).

If the member has to finish its participation in the interaction and it is not playing any role, it will be inmediately abandoned (successfully or unsuccessfully, according to the satisfaction of its purpose). The corresponding event will be forwarded to its interaction and to the interaction of its player role to account for further changes (inv. 22). Otherwise, the member enters the transient *leaving* state, thus preventing any action performance. Then, it waits for the completion of the leaving procedures of its played roles, triggered by proper dispatching of the *leaving* event (inv. 20).

If the conditions to finish its participation does not hold, the obligations and events queue will be updated accordingly. The member is removed from the *agents* auxiliary field and the event processing goes on with the next participant. When the *agents* set gets empty, the field is re-set to the *nil* value.

4 Discussion

This paper has attempted to expose a possible semantic core underlying the wide spectrum of interaction types between autonomous, social and situated software components. In the realm of software architectures, this core has been formalised as an operational model of social connectors, intended to describe both the basic structure and dynamics of multi-agent interactions, from the largest (the agent society itself) down to the smallest ones (communicative actions). Thus, top-level interactions may represent the kind of agent-web pursued by large-scale initiatives such as the Agentcities/openNet one [12]. Large-scale interactions, modelling complex aggregates of agent interactions such as those represented by e-institutions or virtual organizations [2, 28], are also amenable to be conceptualised as particular kinds of first-levels inner social interactions. The last levels of the interaction tree may represent small-scale multi-agent interactions such as those represented by interaction protocols [15], dialogue games [19], or scenes [2]. Finally, bottom-level interactions may represent communicative actions. From this perspective, the member types of a CA include the *speaker* and possibly many *listeners*. The purpose of the speaker coincides with the illocutionary purpose of the CA [24], whereas the purpose of any listener is to declare that it (actually, the software component) successfully processed the meaning of the CA.

The analysis of social interactions put forward in this paper draws upon current proposals of the literature in several general respects, such as the institutional and organizational character of multi-agent systems [9, 11, 14, 28] and the normative perspective on multi-agent protocols [16, 25]. These proposals as well as others focusing in relevant abstractions such as power relationships [4], trust and reputation mechanisms in organizational settings [17], etc., could be further exploited in order to characterize more accurately the organizational character of some multi-agent interactions. Similarly, the informal and preliminary analysis of communicative actions introduced in the last section may similarly benefit from public semantics of communicative actions such as the one introduced in [3]. Last, the abstract model of protocols may be refined taking into account existing operational models of normativity [8, 16]. These analyses shall result in new organizational and communicative abstractions obtained through a refinement and/or extension of the general model of social interactions.

Besides the conceptual objective of analysing the structure and dynamics of multi-agent interactions, the formal model of social interactions presented here is also aimed as a contribution to the field of multi-agent system programming [5]. Unlike the development of individual agents, which has greatly benefited from the design of several *agent* programming languages, non-individual features of multi-agent systems (i.e. those conforming the multi-agent organization or institution [9, 11, 28]) are mostly implemented through development platforms and infrastructures that must be programmed in a low-level language such as Java [18] or in terms of visual modelling [10, 21]. We argue that the current field of multi-agent system programming may greatly benefit from *multi-*

agent programming languages that incorporate as programming constructs the abstractions that have found currency in current organizational methodologies. The model of social interactions put forward in this paper is intended as the execution semantics or the abstract machine of a language of this type. This abstract machine would be independent of particular agent architectures and languages (i.e. the software components may be programmed in a BDI language such as Jason [6] or in a non-agent oriented language). On top of this execution semantics, current and future work aims at the specification of the type system [22] which allows to program the abstract machine, the specification of the corresponding surface syntaxes (both textual and visual) and the design and implementation of a virtual machine over existing middleware technologies such as FIPA platforms and Web services. We also plan to study particular refinements and limitations to the proposed model, particularly with respect to the dispatching of events, the dynamic updates of protocols and rule formalism. In this later aspect, we plan to investigate the use of Answer Set Programming to specify the rules of protocols, attending to the role that incompleteness (rules may only specify either necessary or sufficient conditions, for instance), explicit negation (e.g. on prohibitions) and defaults play in this application domain.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, June 1997.

[2] J. L. Arcos, M. Esteva, P. Noriega, J. A. Rodríguez, and C. Sierra. Engineering open environments with electronic institutions. *Journal on Engineering Applications of Artificial Intelligence*, 18(2):191–204, 2005.

[3] G. Boella, R. Damiano, J. Hulstijn, and L. W. N. van der Torre. Role-based semantics for agent communication: embedding of the 'mental attitudes' and 'social commitments' semantics. In *AAMAS*, pages 688–690, 2006.

[4] G. Boella and L. W. N. van der Torre. Delegation of power in normative multiagent systems. In *DEON*, pages 36–52, 2006.

[5] R. H. Bordini, L. Braubach, M. Dastani, A. E. F. Seghrouchni, J. J. G. Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30:33–44, 2006.

[6] R. H. Bordini, J. F. Hübner, , and R. Vieira. Jason and the golden fleece of agent-oriented programming. In R. H. Bordini, D. M., J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 1. Springer-Verlag, 2005.

[7] P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2):300–302, June 1996.

[8] O. Cliffe, M. D. Vos, and J. A. Padget. Specifying and analysing agent-based social institutions using answer set programming. In *EUMAS*, pages 476–477, 2005.

[9] V. Dignum, J. Vázquez-Salceda, and F. Dignum. Omni: Introducing social structure, norms and ontologies into agent organizations. In R. Bordini, M. Dastani, J. Dix, and A. Seghrouchni, editors, *Programming Multi-Agent Systems Second International Workshop ProMAS 2004*, volume 3346 of *LNAI*, pages 181–198. Springer, 2005.

[10] M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. In M. Gini, T. Ishida, C. Castelfranchi, and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 1045–1052. ACM Press, July 2002.

[11] M. Esteva, J. A. Rodriguez, C. Sierra, P. Garcia, and J. L. Arcos. On the formal specifications of electronic institutions. In F. Dignum and C. Sierra, editors, *Agent-mediated Electronic Commerce (The European AgentLink Perspective)*, volume 1191 of *LNAI*, pages 126–147, Berlin, 2001. Springer.

[12] S. W. et al. Agentcities / opennet testbed. `http://x-opennet.net`, 2004.

[13] J. Ferber and O. Gutknecht. A meta-model for the analysis of organizations in multi-agent systems. In Y. Demazeau, editor, *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS'98)*, pages 128–135, Paris, France, July 1998. IEEE Press.

[14] J. Ferber, O. Gutknecht, and F. Michel. From agents to organizations: An organizational view of multi-agent systems. In *AOSE*, pages 214–230, 2003.

[15] Foundation for Intelligent Physical Agents. *FIPA Interaction Protocol Library Specification.* http://www.fipa.org/repository/ips.html, 2003.

[16] A. García-Camino, J. A. Rodríguez-Aguilar, C. Sierra, and W. Vasconcelos. Norm-oriented programming of electronic institutions. In *AAMAS*, pages 670–672, 2006.

[17] R. Hermoso, H. Billhardt, and S. Ossowski. Integrating trust in virtual organisations. In *Workshop in Coordination, Organisation, Institutions and Norms in MultiAgent Systems*, May 2006.

[18] JADE. The JADE project home page. `http://jade.cselt.it`, 2005.

[19] P. McBurney and S. Parsons. A formal framework for inter-agent dialogues. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 178–179, Montreal, Canada, May 2001. ACM Press.

[20] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 178–187. ACM Press, June 2000.

[21] J. Pavón and J. Gómez-Sanz. Agent oriented software engineering with ingenias. In V. Marik, J. Muller, and M. Pechoucek, editors, *Proceedings of the 3rd International Central and Eastern European Conference on Multi-Agent Systems.* Springer Verlag, 2003.

[22] B. C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, MA, 2002.

[23] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Sept. 1981.

[24] J. Searle. *Speech Acts.* Cambridge University Press, 1969.

[25] M. Sergot. A computational theory of normative positions. *ACM Transactions on Computational Logic*, 2(4):581–622, Oct. 2001.

[26] J. M. Serrano, S. Ossowski, and A. Fernández. The pragmatics of software agents - analysis and design of agent communication languages. *Intelligent Information Agents - An AgentLink Perspective (Klusch, Bergamaschi, Edwards & Petta, ed.), Lecture Notes in Computer Science*, 2586:234–274, 2003.

[27] M. P. Singh. Agent-based abstractions for software development. In F. Bergenti, M.-P. Gleizes, and F. Zambonelli, editors, *Methodologies and Software Engineering for Agent Systems*, chapter 1, pages 5–18. Kluwer, 2004.

[28] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, July 2003.