Operational Semantics of Multiagent Interactions

Juan M. Serrano
University Rey Juan Carlos
C/Tulipan S/N
Madrid, Spain
juanmanuel.serrano@urjc.es

Sergio Saugar
University Rey Juan Carlos
C/Tulipan S/N
Madrid, Spain
sergio.saugar@urjc.es

ABSTRACT

The social stance advocated by institutional frameworks and most multi-agent system methodologies has resulted in a wide spectrum of organizational and communicative abstractions which have found currency in several programming frameworks and software platforms. Still, these tools and frameworks are designed to support a limited range of interaction capabilities that constrain developers to a fixed set of particular, pre-defined abstractions. The main hypothesis motivating this paper is that the variety of multi-agent interaction mechanisms – both, organizational and communicative, share a common semantic core. In the realm of software architectures, the paper proposes a connector-based model of multi-agent interactions which attempts to identify the essential structure underlying multi-agent interactions. Furthermore, the paper also provides this model with a formal execution semantics which describes the dynamics of social interactions. The proposed model is intended as the abstract machine of an organizational programming language which allows programmers to accommodate an open set of interaction mechanisms.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*multi-agent systems*

General Terms

Languages, Theory, Design

Keywords

Social interactions, software connectors, operational semantics

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 [26, 24, 15]. Particularly, the organizational or social stance advocated by institutional frameworks [2] and most multi-agent system (MAS) methodologies [26, 10], provides an excellent basis to deal with the complexity and dynamism of the interactions among system components. This approach has resulted in a wide spectrum of organizational and communicative abstractions, such as institutions, normative positions, power relationships, organizations, groups, scenes, dialogue games, communicative actions (CAs), etc., to effectively model the interaction space of MAS. This wealth of computational abstractions has found currency in several programming frameworks and software platforms (AMELI [9], MadKit [13], INGENIAS toolkit [18], etc.), which leverage multi-agent middlewares built upon raw ACL-based interaction mechanism [14], and minimize the gap between organizational metamodels and target implementation languages.

Still, these tools and frameworks are designed to support a limited range of interaction capabilities that constrain developers to a fixed set of particular, pre-defined abstractions. The main hypothesis motivating this paper is that the variety of multi-agent interaction mechanisms – both, organizational and communicative, share a common semantic core. This paper thus focuses on the fundamental building blocks of multi-agent interactions: those which may be composed, extended or refined in order to define more complex organizational or communicative types of interactions.

Its first goal is to carry out a principled analysis of multi-agent interactions, departing from general features commonly ascribed to agent-based computing: autonomy, situatedness and sociality [26]. To approach this issue, we draw on the notion of *connector*, put forward within the field of software architectures [1, 17]. The outcome of this analysis will be a connector-based model of multi-agent interactions between autonomous social and situated components, i.e. agents, attempting to identify their essential structure. Furthermore, the paper also provides this model with a formal execution semantics which describes the dynamics of multi-agent (or social) interactions. Structural Operational Semantics (SOS)[21], a common technique to specify the operational semantics of programming languages, is used for this purpose.

The paper is structured as follows: first, the major entities and relationships which constitute the structure of social interactions are introduced. Next, the dynamics of social interactions will show how these entities and relationships evolve. Last, relevant work in the literature is discussed

AAMAS'07 May 14–18 2007, Honolulu, Hawai'i, USA.

978-81-904262-7-5 (RPS)

with respect to the proposal, limitations are addressed, and current and future work is described.

2. SOCIAL INTERACTION STRUCTURE

From an architectural point of view, interactions between software components are embodied in software connectors: first-class entities defined on the basis of the different roles played by software components and the protocols that regulate their behaviour [1]. The roles of a connector represent its participants, such as the *caller* and *callee* roles of an RPC connector, or the *sender* and *receiver* roles in a message passing connector. The attachment operation binds a component to the role of a given connector.

The analysis of social interactions introduced in this section gives rise to a new kind of *social connector*. It refines the generic model in several respects, attending to the features commonly ascribed to agent-based computing:

- According to the *autonomy* feature, we may distinguish a first kind of participant (i.e. role) in a social interaction, so-called *agents*. Basically, agents are those software components which will be *regarded* as autonomous within the scope of the interaction[1].
- A second group of participants, so-called *environmental resources*, may be identified from the *situatedness* feature. Unlike agents, resources represent those non-autonomous components whose state may be externally controlled by other components (agents or resources) within the interaction. Moreover, the participation of resources in an interaction is not mandatory.
- Last, according to the *sociality* of agents, the specification of social connector protocols – the glue linking agents among themselves and with resources, will rely on *normative concepts* such as permissions, obligations and empowerments [23].

Besides *agents*, *resources* and *social protocols*, two other kinds of entities are of major relevance in our analysis of social interactions: *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 interaction resulting from the performance of actions or the activity of environmental resources.

In the following, we describe the basic entities involved in social interactions. Each kind of entity T will be specified as a record type $T \triangleq \langle l_1 : T_1, \ldots l_n : T_n \rangle$, possibly followed by a number of invariants, definitions, and the actions affecting their state. Instances or values v of a record type T will be represented as $v = \langle v_1, \ldots, v_n \rangle \ : \ T$. The type $SetT$ represents a collection of values drawn from type T. The type $QueueT$ represents 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 $Enum\ \{v_1, \ldots, v_n\}$ represents an enumeration type whose values are $v_1, \ldots, v_n$. 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, $\ldots$, the expression $v^{l_1,l_2,\cdots}$ is syntactic sugar for $((v^{l_1})^{l_2}) \ldots$. 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.

[1]Note that we think of the autonomy feature in a relative, rather than absolute, perspective. Basically, this means that software components counting as agents in a social interaction may behave non-autonomously in other contexts, e.g. in their interactions through human-user interfaces. This conceptualization of agenthood resembles the way in which objects are understood in CORBA: as any kind of software component (C, Prolog, Cobol, etc.) attached to an ORB.

2.1 Social Interactions

Social interactions shall be considered as composite connectors [17], 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 (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 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:

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

where the *mem*ber and *env*ironment fields represent the agents ($\mathcal{A}$) and local resources ($\mathcal{R}$) participating in the interaction; the *sub*-interaction field, its set of inner interactions; and the *prot*ocol 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 $\mathcal{S}_\mathcal{I} \triangleq Enum\ \{open, closing, closed\}$ 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

[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.

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, vary 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 attach themselves 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:

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

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 $\mathcal{S}_{\mathcal{A}} \triangleq Enum\ \{playing, leaving, succ, unsuc\}$ represents the execution *state* of the agent. Its life-cycle begins in the *playing* state when its *player* agent joins the interaction, or some software component is attached as an agent to the multi-agent system (in this latter case, the *player* value is *nil*). The derived *roles* and *partIn* features represent the roles played by the agent and the contexts in which these roles are played (def. 4, 5)[4]. 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 actions (e.g. to *set up* sub-interactions) in order to satisfy their purposes within some interaction. Moreover, components need 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 represent the queues of attempts to execute some actions ($\mathcal{ACT}$), and the events ($\mathcal{E}$) received by the agent which have not been observed yet. An agent may update its event 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 record type:

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

Essentially, resources can be regarded as *objects* deployed in a social setting. This means that resources are created, accessed and manipulated by agents in a social interaction *context* (def. 6), 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.

2.4 Protocols

The protocol of any interaction is made up of the rules which govern its overall state and dynamics. The present specification abstracts away the particular formalism used to specify these rules, and focuses instead on several requirements concerning the structure and interface of protocols. Accordingly, the type $\mathcal{P}$ of protocols, ranged over by meta-

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

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

variable p, is defined as follows[5]:

$$\begin{array}{lll}
\mathcal{P} & \triangleq & \langle emp : \mathcal{A} \times \mathcal{ACT} \rightarrow Boolean, \\
& & perm : \mathcal{A} \times \mathcal{ACT} \rightarrow Boolean, \\
& & obl :\rightarrow Set\ (\mathcal{A} \times Set\ \mathcal{O} \times Set\ \mathcal{E}), \\
& & monitor : \mathcal{E} \rightarrow Set\ \mathcal{A}, \\
& & finish :\rightarrow Boolean, \\
& & over : \mathcal{A} \rightarrow Boolean \rangle \\
\textbf{def}. : & & (7)\ \ p^{context} = i \Leftrightarrow p = i^{prot} \\
\textbf{inv}. : & & (8)\ \ p^{finish}() \ \wedge\ s \in p^{context,sub} \Rightarrow s^{prot,finish}() \\
& & (9)\ \ p^{finish}() \ \wedge\ a \in p^{context,mem} \Rightarrow p^{over}(a) \\
& & (10)\ \ p^{over}(a) \ \wedge\ a_i \in a^{roles} \Rightarrow a_i^{context,prot,over}(a_i) \\
& & (11)\ \ \alpha^{add} \cup \{a\} \subseteq p^{monitor}(\langle a, \alpha, _ \rangle) \\
\textbf{act}. : & & Close, Leave
\end{array}$$

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 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. 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, specify that those students may only join the course in the 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[6].

Secondly, protocols shall allow to determine the *obligations* of agents towards the interaction. 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 identifies the agents whose obligation set must be updated. Moreover, it returns for each agent a collection of events representing the changes in the obligation set. For instance, the course's protocol establishes that members of departments must join the course as teachers whenever they are assigned to the course's subject.

Thirdly, the protocol shall allow to specify *monitoring* rules for the different events originating within the interaction. Corresponding rules shall establish the set of agents that must be awared of some event. For instance, this functionality is exploited by teachers in order to monitor the enrollment of students to the course.

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 member agent 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)[7]. 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 protocols governing their interactions. This section focuses on the dynamics resulting from a particular kind of external action: the *attempt* of some component, attached to the community as an agent, to execute a given (internal) action. 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 rise 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 events are processed. The resulting dynamics described bellow allows for 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 the 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 actions, ranged over by meta-variable α, is specified as follows:

$$\begin{array}{lll}
\mathcal{ACT} & \triangleq & \langle state : \mathcal{S}_{\mathcal{ACT}}, target : \mathcal{I}, add : Set\ \mathcal{A} \rangle \\
\textbf{def}. : & & (12)\ \ \alpha^{perf} = a \Leftrightarrow \alpha \in a^{att}
\end{array}$$

[5]The formalization assumes that protocol's functions implicitly recieve as input the interaction being regulated.

[6]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 *admission stage* is an additional boolean field of the structure for school interactions. The generic types $\mathcal{I}$, $\mathcal{A}$, $\mathcal{R}$ and $\mathcal{P}$ are thus extendable.

[7]The *close* and *leave* actions update the finish and over function fields as explained in the next section. Additional actions, such as *permit*, *forbid*, *empower*, etc., to update other protocol's fields are yet to be identified in future work.

where: the *perf*ormer is formally defined as the agent who stores the action in its queue of attempts, and the *state* field represents the current phase of processing. This process goes through four major phases, as specified by the enumeration type $\mathcal{S}_{\mathcal{ACT}} \triangleq Enum\,\{emp, perm, exec\}$: *empowerment checking*, *permission checking* and *action execution*, 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[8], 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. For instance, when the software component attached as a student in a degree attempts to join as a student the course in which some subject is teached, the empowerment rules of the course interaction are checked. If the (degree) student has passed the course's prerequisite subjects the join action will be inserted in its queue of attempts and considered for execution.

$$\frac{\alpha^{target,prot,emp}(a,\alpha)}{a = \langle playing, \_, \_, q_{ACT}, \_, \_\rangle \overset{\langle a,\alpha\rangle:\mathcal{ACT}}{\longrightarrow} \langle playing, \_, \_, q'_{ACT}, \_, \_\rangle} \quad (1)$$

$$Where: \quad (\alpha')^{state} = perm \qquad (q'_{ACT}) = insert(\alpha', q_{ACT})$$

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 (e.g. it will be checked whether the student paid the fee for the course's subject). 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.

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

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

[8]Labels of record instances are omitted to allow for more compact specifications. Moreover, note that record updates in "where" clauses only affect the specified fields.

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]:

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

$$\textbf{inv.}: \quad (13) \ \alpha^{new,mem} = \alpha^{new,res} = \alpha^{new,sub} = \emptyset \qquad (14) \ \alpha^{new,state} = open$$

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. 13) and its state must be *open* (inv. 14). 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 is removed; 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 (which includes a description of the change, the agent that caused it and the action performed) is inserted in the output port of the target's event channel.

$$\frac{\alpha^{state} = exec \ \wedge \ \alpha : SetUp \ \wedge \ \alpha^{new} = i}{\begin{array}{l} a = \langle playing, \_, \_, [\alpha|q_{ACT}], \_, \_\rangle \longrightarrow \langle playing, \_, \_, q_{ACT}, \_, \_\rangle \\ \alpha^{target} = \langle open, \_, \_, \_, \_, s_I, c\rangle \longrightarrow \langle open, \_, \_, \_, \_, s_I \cup i', c'\rangle \end{array}} \quad (3)$$

$$Where: \quad (i')^{ini} = a \qquad (c')^{out,ev} = insert(\langle a, \alpha, sub(\alpha^{target}, i')\rangle, c^{out,ev})$$

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:

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

$$\textbf{inv.}: \quad (15) \ \alpha^{target,state} = open \qquad (16) \ \alpha^{target,context} \neq nil \qquad (17) \ \alpha^{upd}(\alpha^{target,prot,finish})()$$

where the inherited *target* field represents the interaction to be closed (which must be open and different to the top-interaction, according to invariants 15 and 16) and the new

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

[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.

*upd*ate field represents a proper higher-order function to update the target's protocol (inv. 17). 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 output port. This event will actually trigger the closing process of the interaction as described in the next subsection.

$$\frac{\alpha^{state} = exec \ \wedge \ \alpha : Close}{\begin{array}{l} a = \langle playing, \_, \_, [\alpha | q_{ACT}], \_, \_\rangle \longrightarrow \langle playing, \_, \_, q_{ACT}, \_, \_\rangle \\ \alpha^{target} = \langle open, \_, \_, \_, \_, p, c\rangle \longrightarrow \langle open, \_, \_, \_, \_, p', c'\rangle \end{array}} \quad (4)$$

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

3.2 Event Processing

The processing of events is 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:

$$\mathcal{CH} \ \triangleq \ \langle out : OutP, in : InP\rangle$$

$$\begin{array}{lll} \textbf{inv.}: & (18) & c^{context} \in c^{out,disp}(\langle \_, \_, finish(c^{context})\rangle) \\ & (19) & c^{context} \in c^{out,disp}(\langle \_, \_, over(a)\rangle) \\ & (20) & c^{context,sub} \subseteq c^{out,disp}(closing(c^{context})) \\ & (21) & a^{partsIn} \subseteq c^{out,disp}(leaving(a)) \\ & (22) & c^{context} \in c^{out,disp}(closed(i)) \\ & (23) & \{c^{context}, a^{player,context}\} \subseteq c^{out,disp}(left(a)) \end{array}$$

$$OutP \triangleq \langle ev : Queue \ \mathcal{E}, disp : \mathcal{E} \rightarrow Set \ \mathcal{I}, int : Set \ \mathcal{I}, ag : Set \ \mathcal{A}\rangle$$
$$InP \triangleq \langle ev : Queue \ \mathcal{E}, stage : Enum \ \{int, mem, obl\}, ag : Set \ \mathcal{A}\rangle$$

The *out*put port stores and processes the *ev*ents originated within the scope of the channel's interaction. Its first purpose is to dispatch the local events to the agents identified by the protocol's monitoring function. Moreover, since these events may influence the results of the finishing, over and obligation functions of certain protocols, they will also be dispatched to the input ports of the interactions identified through a *dis*patching function – whose invariants will be explained later on. Thus, input ports serve as a coordination mechanism which activate the re-evaluation of the above functios whenever some event is received[11]. Accordingly, the processing of some event goes through four major stages: *event dispatching*, *interaction state update*, *member state update* and *obligations update*. The first one takes place in the output port of the interaction in which the event originated, whereas the other ones execute in separate control threads associated to the input ports of the interactions to which the event was dispatched.

3.2.1 Event 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* and *ag* fields are initialised with the returned values of the dispatching and protocol's monitoring functions, respectively (rule 5). Then, additional rules simply iterate over these collections until all agents and interactions have been notified (i.e., both sets are empty). Last, the event is removed from the queue and the auxiliary fields are re-set to *nil*.

The dispatching function shall identify the set of interactions (possibly, empty) that may be affected by the event (which may include the channel's interaction itself)[12]. For instance, according to the finishing rule of university courses mentioned in the last section, the event representing the end of the admission stage, originated within the scope of the school interaction, will be dispatched to every course of the school's degrees. Concerning the monitoring function, according to invariant 11 of protocols, if the event is generated as the result of an action performance, the agents to be notified will include the performer and addressees of that action. Thus, according to the monitoring rule of university courses, if a student of some degree joins a certain course and specifies a colleague as addressee of that action, the course's teachers and itself will also be notified of the successful execution.

$$\frac{c_s^{context,state} = open \ \wedge \ c_s^{context,prot,monitor} = mon}{c_s = \langle\langle [e|\_], d, nil, nil\rangle, \_\rangle \longrightarrow \langle\langle [e|\_], \_, d(e), mon(e)\rangle, \_\rangle} \quad (5)$$

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. 18) serves as a trigger of the closing procedure. If the interaction has not to be finished, the input port *stage* field is set to the *mem*ber state update stage and the auxiliary *ag* field is initialised to the interaction members. 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 6 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 22, this event will be later inserted to its input channel to allow for further treatment.

$$\frac{\begin{array}{l} c_1^{in,ev} \neq \emptyset \ \wedge \ c_1^{in,stage} = int \ \wedge \ p^{finish}() \\ \langle \_, \_, \_, \_, \{i\} \cup s_I, \_, c\rangle \longrightarrow \langle \_, \_, \_, \_, s_I, \_, c'\rangle \\ i = \langle \_, \_, \emptyset, \_, \emptyset, p, c_1\rangle \longrightarrow \langle closed, \_, \_, \_, \_, \_, \_\rangle \end{array}}{Where: \ (c')^{out,ev} \ = \ insert(closed(i), c^{out,ev})} \quad (6)$$

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 (e.g. if the admission period ends and no student has matriculated for the course, teachers has to be finished before finishing the course itself). As rule 7 shows, the interaction is moved to the transient *closing* state and a corresponding event is inserted in the output port. According to invariant 20, the closing event will be dispatched to every sub-interaction in order to activate its closing procedure (guaranteed by invariant 8). Moreover,

[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.

[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.

the *stage* and *ag* fields are properly initialised so that the process goes on in the next *member state update* stage. This stage will further initiate the *leaving* process of the members (according to invariant 9).

$$\frac{c^{in,ev} \neq \emptyset \ \wedge \ c^{in,stage} = int \wedge \ p^{finish}() \ \wedge \ (s_A \neq \emptyset \vee s_I \neq \emptyset)}{i = \langle open, \_, s_A, \_, s_I, p, c\rangle \longrightarrow \langle closing, \_, s_A, \_, s_I, p, c'\rangle} \quad (7)$$

$$\begin{aligned} Where: \ (c')^{out,ev} &= insert(closing(i), c^{out,ev}) \\ (c')^{in,stage} &= mem \\ (c')^{in,ag} &= s_A \end{aligned}$$

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 23 and 22) so that the conditions of the first scenario will hold.

3.2.3 Member state update

This stage simply iterates over the members of the interaction to check whether they must be finished according to the protocol's *over* function. When all members have been checked, the *stage* field will be set to the next *obligation* update stage and the auxiliary *ag* field will be initalised with the agents identified by the protocol's *obligation* update function.

If some member has to end 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 agent to account for further changes (inv. 23). 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. 21).

3.2.4 Obligations update

In this stage, the obligations of agents (not necessaryly members of the interaction) towards the interaction are updated accordingly. When all the identified agents have been updated, the event is removed from the input queue and the *stage* field is set back to the *int*eraction state update. For instance, when a course interaction receives an event representing the assignment of some department member to its subject, an obligation to join the course as a teacher is created for that member. Moreover, the event representing this change is added to the output channel of the department interaction.

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 [25]. Large-scale interactions, modelling complex aggregates of agent interactions such as those represented by e-institutions or virtual organizations [2, 26], are also amenable to be conceptualised as particular kinds of first-level social interactions. The last levels of the interaction tree may represent small-scale multi-agent interactions such as those represented by interaction protocols [11], dialogue games [16], 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 [22], 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 [2, 26, 10, 7] and the normative perspective on multi-agent protocols [12, 23, 20]. These proposals as well as others focusing in relevant abstractions such as power relationships, contracts, trust and reputation mechanisms in organizational settings, etc., could be further exploited in order to characterize more accurately the organizational character of some multi-agent interactions. Similarly, the conceptualization of communicative actions as atomic interactions 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 norms [12, 6]. These analyses shall result in new organizational and communicative abstractions obtained through a refinement and/or extension of the general model of social interactions. Thus, the proposed model is not intended to capture every organizational or communicative feature of multi-agent interactions, but to reveal their roots in basic interaction mechanisms. In turn, this would allow for the exploitation of common formalisms, particularly concerning protocols.

Unlike the development of individual agents, which has greatly benefited from the design of several *agent* programming languages [4], societal features of multi-agent systems are mostly implemented in terms of visual modelling [8, 18] and a fixed set of interaction abstractions. We argue that the current field of multi-agent system programming may greatly benefit from *multi-agent* programming languages that allow programmers to accommodate an open set of interaction mechanisms. The model of social interactions put forward in this paper is intended as the abstract machine of a language of this type. This abstract machine would be independent of particular agent architectures and languages (i.e. software components may be programmed in a BDI language such as Jason [5] or in a non-agent oriented language).

On top of the presented execution semantics, current and future work aims at the specification of the type system [19] 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 or Web services. We also plan to study particular refinements and limitations to the proposed model, particularly with respect to the dispatching of events, semantics

of obligations, dynamic updates of protocols and rule formalisms. In this latter 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. prohibitions) and defaults play in this domain.

5. ACKNOWLEDGMENTS

The authors thank anonymous reviewers for their comments and suggestions. Research sponsored by the Spanish Ministry of Science and Education (MEC), project TIN2006-15455-C03-03.

6. 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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] M. Esteva, B. Rosell, J. A. Rodríguez-Aguilar, and J. L. Arcos. AMELI: An agent-based middleware for electronic institutions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 1, pages 236–243, 2004.

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

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

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

[13] O. Gutknecht and J. Ferber. The MadKit agent platform architecture. *Lecture Notes in Computer Science*, 1887:48–55, 2001.

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

[15] M. Luck, P. McBurney, O. Shehory, and S. Willmott. *Agent Technology: Computing as Interaction – A Roadmap for Agent-Based Computing*. AgentLink III, 2005.

[16] 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.

[17] 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.

[18] 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.

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

[20] J. Pitt, L. Kamara, M. Sergot, and A. Artikis. Voting in multi-agent systems. Feb. 27 2006.

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

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

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

[24] 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.

[25] S. Willmot and al. Agentcities / opennet testbed. `http://x-opennet.net`, 2004.

[26] 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.