AMELI: An Agent-based Middleware for Electronic Institutions[*]

Marc Esteva, Bruno Rosell, Juan A. Rodríguez-Aguilar, Josep Ll. Arcos
IIIA-CSIC, Campus UAB
08193 Bellaterra, Barcelona, Spain
{marc,rosell,jar,arcos}@iiia.csic.es

Abstract

The design and development of open multi-agent systems (MAS) is a key aspect in agent research. We advocate that they can be realised as electronic institutions*. In this paper we focus on the execution of electronic institutions by introducing AMELI, an infrastructure that mediates agents' interactions while* enforcing *institutional rules. An innovative feature of AMELI is that it is of* general purpose *(it can interpret any institution specification), and therefore it can be regarded as* domain-independent*. The combination of ISLANDER [5] and AMELI provides full support for the design and development of electronic institutions.*

1. Introduction

So far MAS researchers have bargained for well-behaved agents immersed in reliable infrastructures in relatively simple domains. Such assumptions are not valid any longer when considering *open systems* whose components are unknown beforehand, can change over time, and can be self-interested human and software agents developed by different parties. Thus, open MAS can be regarded as complex systems where (possibly) large, varying populations of agents exhibiting different (possibly deviating, or even fraudulent) behaviours interact. Openness without control may lead to chaotic behaviours. Therefore, the design and development of open MAS appears as a highly complex task. Hence, it seems apparent the need for introducing regulatory structures establishing what agents are permitted and forbidden to do. Notice that human societies have successfully deal with regulation by deploying institutions. Thus, we advocate for the introduction of their electronic counterpart, namely *electronic institutions* (EIs)[7], to shape the environment wherein agents interact (*environment engineering*) by introducing sets of artificial constraints that articulate their interactions. Hence that we advocate that open multi agent systems can be designed and developed as EIs [4]. At this aim, we clearly differentiate two stages: the *specification* of institutional rules; and their subsequent *execution*. On the one hand, the specification focuses on macro-level (social) aspects of agents, establishing norms. On the other hand, the institution is in charge of enforcing the specified norms to participating agents at run time. In our previous work, we focused on the support of the (graphical) specification of institutions via ISLANDER [5]. In this paper, we draw our attention to the support of the execution of EIs via an infrastructure that mediates agents' interactions while enforcing institutional rules.

The paper is organised as follows. First, in section 2 we succinctly introduce EIs to subsequently identify (section 3) the required features and functionalities of an infrastructure for EIs. Next, we focus on AMELI, our software infrastructure for EIs (available at http://e-institutions.iiia.csic.es). More concretely, in section 4 we elaborate on the architecture of AMELI, while section 5 details how AMELI computationally realises the requirements identified in section 3. A distinguishing, innovative feature of AMELI is that it is of *general purpose* (it can interpret any ISLANDER specification), and therefore it can be regarded as *domain-independent*. The combination of ISLANDER and AMELI allows to support both the design and development of open MAS adopting a social perspective. In order to illustrate how AMELI works we present an example in subsection 5.3. Finally, our contributions are summarised in section 6.

2. Electronic Institutions

Above we identified as our main goal the enactment of a constrained environment that shapes open agent societies. We argue that such artificial constraints can be effectively introduced by means of EIs [4]. In general terms, EIs structure agent interactions, establishing what agents are permitted and forbidden to do as well as the consequences of their actions. Next, we summarise the notion of EI (thoroughly

[*] Partially supported by project Web-i(2) (TIC-2003-08763-C02-01).

described in [4]), illustrated via a double auction (DA) market institution. Within this market, traders (both buyers and sellers) meet to trade their goods under the supervision of trade manager agents. The market consists of two major activities. Firstly, a trade manager agent receives requests for buying and selling goods from trading agents. When there are enough traders interested in a certain commodity, the trade manager opens a new trading as a DA.

In general, an EI regulates multiple, distinct, concurrent, interrelated, dialogic activities, each one involving different groups of agents playing different roles. For each activity, interactions between agents are articulated through agent group meetings, the so-called *scenes*, that follow well-defined interaction protocols whose participating agents may change over time (agents may enter or leave). A scene protocol is specified by a directed graph whose nodes represent the different states of a dialogic interaction between roles. Its arcs are labelled with illocution schemes (whose sender, receiver and content may contain variables) or timeouts. At execution time agents interact by uttering grounded illocutions matching the specified illocution schemes, and so binding their variables to values, building up the *scene context*. Moreover, arcs labelled with illocution schemes may have constraints attached based on the scene context to impose restrictions on the paths that the scene execution can follow. For instance, once all bids and offers are submitted after a DA round, we can specify by means of constraints that buyers can only accept the minimum offer (according to the bound values in the scene context). Formally, a scene is a tuple $s = \langle R, CL, W, w_0, W_f, (WA_r)_{r\in R}, (WE_r)_{r\in R}, \Theta, \lambda, min, Max\rangle$ where R is the set of scene roles; CL is a communication language; W is the set of scene states; $w_0 \in W$ is the initial state; $W_f \subseteq W$ is the set of final states; $(WA_r)_{r\in R} \subseteq W$ is a family of sets such that WA_r stands for the set of access states for role $r \in R$; $(WE_r)_{r\in R} \subseteq W$ is a family of non-empty sets such that WE_r stands for the set of exit states for role $r \in R$; $\Theta \subseteq W \times W$ is a set of directed edges; $\lambda : \Theta \longrightarrow L$ is a labelling function, where L can be a timeout, or an illocutions scheme and a list of constraints;$min, Max : R \longrightarrow \mathbb{N}$ $min(r)$ and $Max(r)$ return the minimum and maximum number of agents that must and can play role $r \in R$.

More complex activities can be specified by establishing networks of scenes (activities), the so-called *performative structures*. These define how agents can legally move among different scenes (from activity to activity) depending on their role. Furthermore, a performative structure defines when new scene executions start, and if a scene can be multiply executed at run time. A performative structure can be regarded as a graph whose nodes are both scenes and *transitions* (scene connectives), linked by directed arcs. The type of transition allows to express choice points (*Or* transitions) for agents to choose which target scenes to enter, or

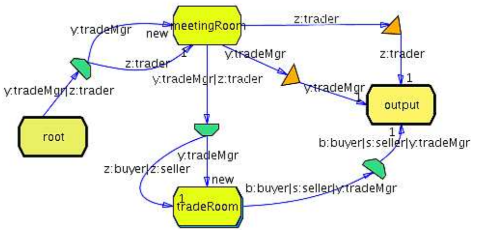


Figure 1. Double auction market specification

synchronisation/parallelisation points (*And* transitions) that force agents to synchronise before progressing to different scenes in parallel. The labels on the directed arcs determine which agents, depending on their roles, can progress from scenes to transitions, or the other way round. Since the very same scene specification can be multiply executed, the type of the arcs connecting transitions to scenes define whether an agent following the arc can join a *new*, *one*, *some* or *all* execution(s) of the target scene. Formally, a performative structure is a tuple $PS = \langle S, T, s_0, s_\Omega, E, f_L, f_T, f_E^O, \mu\rangle$ where S is a set of scenes; T is a set of transitions; $s_0 \in S$ is the *initial* scene; $s_\Omega \in S$ is the *final* scene; $E = E^I \bigcup E^O$ is a set of arc identifiers where $E^I \subseteq S \times T$ is a set of edges from scenes to transitions and $E^O \subseteq T \times S$ is a set of edges from transitions to scenes; $f_L : E \longrightarrow FND_{2^{V_A \times R}}$ maps each arc to a disjunctive normal form of pairs of agent variable and role identifier representing the arc label; $f_T : T \longrightarrow \mathcal{T}$ maps each transition to its type; $f_E^O : E^O \longrightarrow \mathcal{E}$ maps each arc to its type; $\mu : S \longrightarrow \{0, 1\}$ sets if a scene can be multiply instantiated at run time. Figure 1 depicts the specification of the performative structure of the DA market as shown by ISLANDER [5]. Its activities are represented by the *meetingRoom* scene, where traders are matched by a trade manager based on their commodities' interests, and the *tradeRoom* scene, where a DA is run to rule the trading. Observe that trading agents switch their role to either buyer or seller when moving from the *meetingRoom* to the *tradeRoom*. Moreover, while there is a sole execution of the *meetingRoom* scene, multiple executions of the *tradeRoom* scene may occur, being dynamically created depending on trading agents' interests. Finally, the *root* scene and the *output* scene represent the institution's entry and exit.

Agent's actions *within* scenes may create commitments for future actions, interpreted as obligations, captured by a special type of rules called *norms*. Norms establish the actions that activate obligations as well as the actions required to fulfill them. Formally:

$$done(s_1, \gamma_1) \wedge \ldots \wedge done(s_m, \gamma_m) \wedge e_1 \wedge \ldots \wedge e_k \wedge \qquad (1)$$
$$\wedge \neg done(s_{m+1}, \gamma_{m+1}) \wedge \ldots \wedge \neg done(s_{m+n}, \gamma_{m+n}) \rightarrow obl_1 \wedge \ldots \wedge obl_p$$

expresses a norm, where $(s_1, \gamma_1), \ldots, (s_{m+n}, \gamma_{m+n})$ are pairs of scenes and illocution schemes (representing dialogic actions), $e_1, \ldots e_k$ are boolean expressions over illocution schemes' variables, $\neg$ is a defeasible negation, and $obl_1, \ldots, obl_p$ are obligations. The meaning of these rules is that if illocutions matching $\gamma_1, \ldots, \gamma_m$ have been uttered in $s_1, \ldots, s_m$, expressions $e_1, \ldots, e_k$ are satisfied and illocutions $\gamma_{m+1}, \ldots, \gamma_{m+n}$ have *not* been uttered in $s_{m+1}, \ldots, s_{m+n}$, obligations $obl_1, \ldots, obl_p$ hold.

3. Institution infrastructure

3.1. Required features

An EI defines a normative environment that shapes agents' interactions at execution time. Notice though that such environments are open in the sense that any agent is allowed to participate, and thus the number of participating agents within an EI may dynamically vary as agents join in and leave. Therefore, the participants in EIs may be heterogeneous, self-interested agents, written by different people, in different languages and with different architectures. Hence we can not assume that these agents behave according to the institutional rules. And so, what are the required features of an infrastructure that supports such EI environment? First, the main task of an infrastructure must be to facilitate agent participation within the institutional environment while enforcing the institutional rules encoded in the specification. Thus, we demand an institution infrastructure to be capable of *interpreting* any institution specification to be of *general purpose* (the very same infrastructure to realise multiple electronic institutions), and ensure *domain independence*. Furthermore, the infrastructure has to implement the necessary communication and coordination mechanisms that facilitate agent communication. In this way, participating agents can communicate in a higher level language (no need for implementing low-level communication and coordination mechanisms), allowing agent designers to primarily focus on decision making. Lastly, an institution infrastructure is required to be both *architecturally neutral*, to accept agents developed in any language and architecture, and *scalable*, to cope with possibly large, varying agent populations. To summarise, we demand that an institution infrastructure satisfies as requirements: it must facilitate agents' participation within the institution; it must enforce institutional rules; it must prevent participating agents from jeopardising the functioning of institutions; it must be architecturally neutral; it must interpret any specification to guarantee re-usability and domain independence; and it must be scalable.

3.2. Functional requirements

The execution of an institution can be regarded as the concurrent execution of its different scenes. In this environment, the activity of participating agents amounts to interacting with other agents within different scene executions and moving among them. Agents' actions make the institution execution evolve. It is the responsibility of the infrastructure to *control* the institution execution by guaranteeing that all agent interactions abide by the institutional rules. Hence, the infrastructure must control: the flow of agents (when entering/leaving the institution and moving among scene executions), the execution of scenes and transitions; and the adoption and fulfilment of agents' obligations. At this aim, the infrastructure must employ the institutional rules encoded in the specification along with the current execution state. This contains information about the participating agents, scenes' and transitions' executions, and each agent's pending obligations. Formally, we define an execution state as a tuple $\Omega = \langle Ag, \Sigma, T, Obl \rangle$ where $Ag = \{ag_1, \ldots, ag_n\}$ is a finite set of participating agents; $\Sigma = \{\sigma_i^k | s_i \in S, k \in I\!N\}$ is the set of all scene executions (where σ_k^i stands for the $k-$th scene executions of scene s_i); $T = \{t_1, \ldots, t_p\}$ stands for all transition executions ; and $Obl = \{\langle ag, \iota, s\rangle | ag \in Ag, \iota \in CL(s), s \in S\}$ is the set of pending obligations (where $\langle ag, \iota, s\rangle$ stands for the obligation of agent ag to utter illocution ι at scene s). Formally, a scene execution σ_i^k is described as $\sigma_i^k = \langle \omega, \mathcal{A}, \mathcal{B}\rangle$, where ω stands for the current execution state, $\mathcal{A} = \{(ag, r) | ag \in Ag, r \in R(s_i)\}$ is the set of agents participating in the scene along with the roles they play; and $\mathcal{B} = \beta_1, \ldots, \beta_q$ stands for the list of bindings produced by each uttered illocution uttered (representing the context of the conversation). Furthermore, a transition execution is described as a set $\{(ag, \delta) | ag \in Ag, \delta = \{(\sigma_i^k, r) | \sigma_i^k \in \Sigma, k \in I\!N, r \in \mathcal{R}\}\}$ where each agent ag is associated to δ, the scene executions it aims at joining, along with the role to play in each execution, being $\mathcal{R}$ the set of all institutional roles.

Specification	Functionality
Institution	$enter(ag, Roles)$ $exit(ag)$
Performative structure	$create\_scene(s)$ $close\_scene(\sigma)$
Scene	$join(\sigma, SAgents)$ $update\_state(\sigma, \iota)$ $update\_state(\sigma, \tau)$ $leave(\sigma, SAgents)$
Transition	$add\_agents(t, TAgents)$ $move\_to(t, ag, Target)$ $fire(t)$ $remove\_agents(t, TAgents)$
Norm	$add\_obligations(Obligations)$ $remove\_obligations(Obligations)$

Table 1. Infrastructure operations

Next, we identify the operations that an institution infrastructure must implement in order to make an institution execution evolve from state Ω to Ω' as a consequence of agents' actions. At this aim, we first analyse *how* an institution is required to function.

At the outset, any institution execution starts out with the creation of an *initial* scene execution and an *final* scene execution. Thereafter, participating agents can enter and exit and participate within scenes' executions as they are created. Then, we first focus on the execution of a scene described as $\sigma_k^j = \langle \omega, \mathcal{A}, \mathcal{B} \rangle$. It evolves as state transitions (in the scene protocol) occur and as agents join and leave. A state transition occurs by either the utterance of a *valid* illocution or a time-out expiration. An illocution is assessed as valid whenever it complies with the scene protocol considering the current scene's execution (i.e. it matches one of the labels of the outgoing arcs of the current scene state, and the constraints associated to the arc are satisfied). The new execution state is determined according to the scene transition fired, and the context of the conversation β is extended with the new bindings produced by the illocution. As to the time-out case, a time-out expiration causes the scene state to evolve to the target state of the arc labelled with the time-out. Furthermore, the infrastructure must also control that agents enter or leave at access and exit states respectively without violating the restrictions on the minimum and maximum number of agents per role.

Since the flow of agents among scenes' executions are mediated by transitions, agents are required to move to transitions prior to jump into target scenes. At this point, the infrastructure must guarantee that an agent within a scene execution can only move out to a reachable transition (connected to the scene) for its role. Moreover, the infrastructure must control how to *fire* transitions to allow agents to move from transitions to scenes' executions. In order to check when transitions can be fired, the infrastructure must consider the types of transitions (*and*,*or*), the types of arcs connecting scenes at the specification level (*one*,*some*,*all*,*new*), and the current scenes' executions. As to *and* transitions, agents are forced to synchronise prior to move into their target scenes' executions. And notice that when agents follow a *new* arc, a new execution of the target scene is created for them to join in.

Finally, as to the bookkeeping of each agent's pending obligations, the infrastructure must control when to activate a norm (to assign new obligations), and when an agent has carried out the actions that fulfil some of its pending obligations (to unassign them).

Now, we are ready to collect the operations to be undertaken by the infrastructure (we here limit to describe their functionality due to space restrictions):

• $enter(ag, Roles)$. It incorporates agent ag with a subset of roles $Roles \subseteq \mathcal{R}$ into the (to the agents in Ag). Agent ag initially joins the *initial* scene.

• $exit(ag)$. It removes agent ag from the institution, i.e. from the set of participating agents in Ag. An agent can only be removed from an institution when it no longer participates in any scene or transition execution.

• $create\_scene(s)$. It creates a new scene execution for scene $s \in S$ to be added into Σ.

• $close\_scene(\sigma)$. It removes scene execution σ from the scene executions in Σ after reaching a final state and the participating agents are gone.

• $join(\sigma, SAgents)$. It incorporates a set of agents $SAgents = \{(ag, r) | ag \in Ag, r \in R(s)\}$ into scene execution σ, each agent ag playing scene role r. This involves the updating of the participating agents in $\mathcal{A}$.

• $update\_state(\sigma, \iota)$. It updates the state of scene execution σ after the utterance of a valid illocution $\iota \in CL(s)$. This involves updating the current state $w \in \sigma$ along with the list of bindings $\mathcal{B} \in \sigma$.

• $update\_state(\sigma, \tau)$. It updates the state of scene execution σ after the expiration of timeout τ. This solely involves updating the current state $w \in \sigma$.

• $leave(\sigma, SAgents)$. It allows the agents in $SAgents = \{(ag, r) | ag \in Ag, r \in R(s)\}$ to leave scene execution σ, each agent ag playing role r. This involves the updating of the participating agents in $\mathcal{A}$.

• $add\_agents(t, TAgents)$. It incorporates the agents in $TAgents = \{(ag, r) | ag \in Ag, r \in \mathcal{R}\}$ into transition execution $t \in T$, each agent ag playing role r.

• $move\_to(t, ag, Target)$. It adds a valid request from agent ag to join the scene executions in $Target = \{(\sigma_i^k, r) | \sigma_i^k \in \Sigma, r \in \mathcal{R}\}$, playing role r in each scene execution σ_i^k.

• $fire(t)$. It evaluates whether a transition can be fired. If so it returns for each agent in the transition the scene executions that it can join along with the role to play.

• $remove\_agents(t, TAgents)$. It removes the agents in $TAgents = \{ag | ag \in Ag\}$ from transition execution $t \in T$, each agent ag playing role r.

• $add\_obligations(Obligations)$. It includes the obligations in $Obligations$ into the set of obligations Obl. Each obligation is a triple $\langle ag, \iota, s \rangle$ standing for the obligation of agent ag to utter illocution ι in scene $s \in S$.

• $remove\_obligation(Obligations)$. It removes the satisfied obligations in $Obligations$ from Obl.

Table 1 relates the above-described operations with the institution components outlined in section 2. In section 5 we describe an implementation of EI infrastructure that realises such operations.

4. An architecture for electronic institutions

In this section we present an EI architecture encompassing both the institution infrastructure outlined in section 3 and its participating agents. Our architecture, depicted in figure 2, is composed of the following layers:

• **External agent layer.** External agents taking part in the institution.

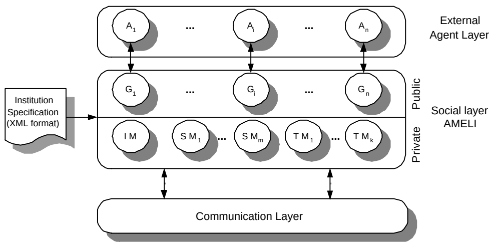


Figure 2. Electronic institution architecture

• **Social layer (AMELI).** Implementation of the control functionality of the institution infrastructure.

• **Communication layer.** In charge of providing a reliable and orderly transport service.

Notice that participating agents in the institution do not interact directly; they have their interactions *mediated* by AMELI. Moreover, AMELI also provides external agents with the information they need to successfully participate in the institution. And more importantly, AMELI takes care of the institutional enforcement: guaranteeing the correct evolution of each scene execution (preventing errors made by the participating agents by filtering erroneous illocutions, thus protecting the institution); guaranteeing that agents' movements between scene executions comply with the specification; and controlling which obligations participating agents acquire and fulfil. The current implementation of AMELI that realises the above-mentioned functionalities is composed of four types of agents:

• **Institution Manager (IM)**. It is in charge of starting an EI, authorising agents to enter the institution, as well as managing the creation of new scene executions. It keeps information about all participants and all scene executions. There is one institution manager per institution execution.

• **Transition Manager (TM)**. It is in charge of managing a transition controlling agents' movements to scenes. There is one transition manager per transition.

• **Scene manager (SM)**. Responsible for governing a scene execution (one scene manager per scene execution).

• **Governor (G)**. Each one is devoted to mediating the participation of an external agent within the institution. There is one governor per participating agent.

Since external agents can only communicate with their governors, we can regard AMELI as composed of two layers: a *public* layer, formed solely by governors; and a *private* layer, formed by the rest of agents, not accessible to external agents. In order for agents to communicate with their governors, they are solely required to be capable of opening a communication channel. Since no further architectural constraints are imposed on external agents, we can regard AMELI as *architecturally neutral*. Observe that AMELI is of general purpose in the sense that the same infrastructure can be deployed to realise different institutions. At this aim, agents composing AMELI load institution specifications as XML documents generated by the ISLANDER editor [5]. Thus, the implementation impact of introducing institutional changes amounts to the loading of a new (XML-encoded) specification. Based on an institution specification (roles, scenes, performative structure, and norms), along with the information about its current execution, AMELI is capable of validating agents' actions and assessing their consequences as detailed in section 5. As depicted in figure 2, the infrastructure is divided into two layers: AMELI and a communication layer offering a reliable and orderly transport service. In this manner, AMELI agents do not need to deal with low-level communication issues, and therefore focus on handling the institution execution. The current implementation of the infrastructure can either use JADE [1] or a publish-subscribe event model as communication layer. When employing JADE, the execution of AMELI can be readily distributed among different machines, permitting the *scalability* of the infrastructure. Finally, participating agents regard our architecture as *communication neutral* since they are not affected by changes in the communication layer.

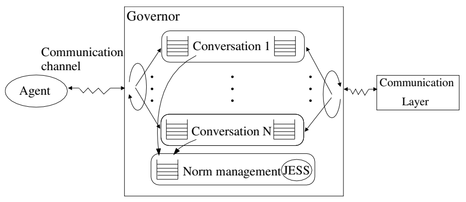


Figure 3. Governor architecture

5. AMELI: an agent-based middleware

5.1. Agent Mediation

Each participating agent in an EI is connected to a governor that mediates all its interactions in the institution once admitted by the institution manager [via operation *enter* in table 1]. The communication between a governor and an agent is structured in conversations corresponding to either a scene or a transition execution in which the agent participates (see figure 3). Conversations are dynamically created and destroyed as the agent joins or leaves scenes and transitions. In the current version, an agent can communicate with its governor either via Java events or *sockets*. Governors are also in charge of managing norms controlling its associated agent pending obligations (detailed in section 5.2.3).

Since an agent can only communicate with its governor, a fundamental aspect of our implementation is the agent-governor protocol, defining the valid messages that an agent and its governor may exchange. They exchange messages in FIPA-ACL whose content has the following elements: *Con-*

Action	Description
enterInstitution	Request to enter the institution
moveToTransition	Request to move from a scene to a transition
moveToScenes	Request to move from a transition to several scenes
saySceneMessage	Request to say a message in a scene
accesScenes	Ask for the scenes the agent can join from a transition
accesTransitions	Ask for the transitions the agent can join from a scene
agentObligations	Ask for pending obligations
sceneState	Ask for a scene's current state
scenePlayers	Ask for agents in a scene

Table 2. Actions from agent to governor.

vID (a conversation identifier); *Action* (an action request or information request to do, or an action result or information the receiver is informed about); and *Parameters* (additional information needed to specify the action).

Table 2 summarises the actions contained in the messages an agent can send to its governor. We differentiate among three types of actions: *illocutionary* (illocutions that agents try to utter within scenes), *motion* (movements between scenes and transitions and the other way around), and *information request* (scenes reachable from a transition, transitions reachable from a a scene, agent's obligations, scenes' states, and scenes' participants). Notice though that an agent can cancel any sent message by sending a cancel message before the request has been processed.

For each received message, the governor replies to the agent with one of the following messages: *agree* (correct message), *refuse* (incorrect message), or *unknown* (message not understood). Correct messages are processed later on considering the context of the conversation it belongs to. Any *illocutionary* or *motion* action requested by an agent to its governor results in either a *success* message (reporting that the action has been successfully done) or a *failure* message (reporting the reason why the governor failed when trying to perform the action). The governor also responds to *information requests* (scenes reachable from a transition, transitions reachable from a a scene, agent's obligations, scenes' states, and scenes' participants), and informs about the events the agent must be aware of (messages addressed to the agent within scenes, changes on the participants within a scene, state transitions in scenes because of time-out expirations, the end of scenes, and the acquirement or fulfilment of obligations).

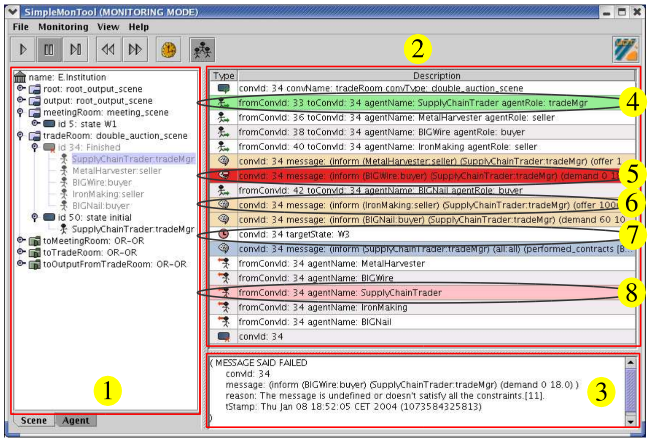
5.2. Institution Management

5.2.1. Scene Management Several agents in AMELI are involved in controlling the execution of a scene, namely: a scene manager and one governor per participating agent. They all coordinate in order to guarantee its sound execution. The execution of a scene starts with the creation of a scene manager aware of the scene protocol, the roles that participating agents may play, and the maximum and minimum number of agents per role. Once the scene manager is brought up, agents may start to join the scene [operation *join*]. Nonetheless, the scene cannot start until the minimum number of agents per role is reached. Thereafter, the scene execution may evolve because of the utterance of a valid illocution or because of a time-out expiration.

As agents interact within a scene, their governors and the scene manager coordinate to evaluate agents' actions employing the scene specification and the execution information. They also coordinate to maintain a shared view of the execution information (participating agents' identifiers with the roles they play, current scene state, and the variable bindings —representing the context of the conversation— caused by uttered illocutions).

As to evaluating illocutions, when a governor receives a request from its agent for uttering an illocution, it forwards it to the scene manager. Thereafter, the scene manager checks whether it is valid according to the scene specification and the execution information. If the message is correct, a scene transition comes about [operation *update_state*]. In this case, the scene manager sends it to the governors of the addressees of the illocution, which in turn forward it to their assigned agents. Moreover, the scene manager updates the scene's execution information. Lastly, the agent requesting the utterance of the illocution is informed by its governor about the success of the action. Otherwise, if the illocution is not correct, the agent is informed about the failed action. If the scene execution reaches a state where there is an outgoing arc labelled with a timeout, the scene manager evaluates the timeout expression to start the timeout countdown. If the timeout expires without a valid illocution, the state transition corresponding to the arc labelled with the timeout occurs [operation *update_state*]. The scene manager reports to all governors, and these to their associated agents.

Scene managers are also in charge of authorising agents to join [operation *join*] or leave [operation *leave*] scene executions. On the one hand, requests for joining the scene are received from transition managers. On the other hand, agents intending to leave a scene must send a message to their governor requesting which transition(s) to go to. If the agent can move to the requested transition (if there is an arc in the performative structure from the scene to the requested transition labelled by the agent's role) the governor informs the scene manager that the agent wants to leave the scene. In both cases, the scene manager blocks the scene execution when it reaches an access or exit state for the agents waiting for joining or leaving. Then, it authorises agents to join or leave unless the restrictions on the maximum and minimum agents per role are violated. When an agent is authorised to join a scene execution, its governor updates him with the current state and the scene's participants thanks to the scene execution information received from the scene manager. When an agent is authorised to leave a scene execution, his governor reports him when moving into a se-



```
name: E.Institution
  root: root_output_scene
  output: root_output_scene
  meetingRoom: meeting_scene
    id 5: state W1
  tradeRoom: double_auction_scene
    id 34: Finished
      SupplyChainTrader:tradeMgr
      MetalHarvester:seller
      BIGWire:buyer
      IronMaking:seller
      BIGNail:buyer
    id 50: state initial
      SupplyChainTrader:tradeMgr
  toMeetingRoom: OR-OR
  toTradeRoom: OR-OR
  toOutputFromTradeRoom: OR-OR
```

Type	Description
	convId: 34 convName: tradeRoom convType: double_auction_scene
	fromConvId: 33 toConvId: 34 agentName: SupplyChainTrader agentRole: tradeMgr
	fromConvId: 36 toConvId: 34 agentName: MetalHarvester agentRole: seller
	fromConvId: 38 toConvId: 34 agentName: BIGWire agentRole: buyer
	fromConvId: 40 toConvId: 34 agentName: IronMaking agentRole: seller
	convId: 34 message: (inform (MetalHarvester:seller) (SupplyChainTrader:tradeMgr) (offer 1
	convId: 34 message: (inform (BIGWire:buyer) (SupplyChainTrader:tradeMgr) (demand 0 18
	fromConvId: 42 toConvId: 34 agentName: BIGNail agentRole: buyer
	convId: 34 message: (inform (IronMaking:seller) (SupplyChainTrader:tradeMgr) (offer 100
	convId: 34 message: (inform (BIGNail:buyer) (SupplyChainTrader:tradeMgr) (demand 60 10
	convId: 34 targetState: W3
	convId: 34 message: (inform (SupplyChainTrader:tradeMgr) (all:all) (performed_contracts [B...
	fromConvId: 34 agentName: MetalHarvester
	fromConvId: 34 agentName: BIGWire
	fromConvId: 34 agentName: SupplyChainTrader
	fromConvId: 34 agentName: IronMaking
	fromConvId: 34 agentName: BIGNail
	convId: 34

```
( MESSAGE SAID FAILED
    convId: 34
    message: (inform (BIGWire:buyer) (SupplyChainTrader:tradeMgr) (demand 0 18.0) )
    reason: The message is undefined or doesn't satisfy all the constraints.[11].
    tStamp: Thu Jan 08 18:52:05 CET 2004 (1073584325813)
)
```

Figure 4. Double auction market monitoring

lected transition [operation *add_agents*]. In both cases the rest of governors are informed about the changes concerning the participants so that they report to their agents. Finally, the scene manager closes a scene execution when it reaches a final state and all participating agents are gone [operation *close*], acknowledging the institution manager.

5.2.2. Transition management Each transition is managed by a transition manager agent, devoted to route agents to their target scene executions. Within a transition, an agent can request its reachable scene executions to its governor. Thereafter, the agent can request which scene executions to join. The governor forwards the request to the transition manager for analysis. If incorrect, the agent will have its request refused; otherwise the transition manager keeps it [operation *move_to*], and checks if the transition can be fired along with the agents that can start moving to their target scene executions [operation *fire*]. Notice that *And* transitions synchronise their agents prior to their firing. Movements are made asynchronously: agents leave the transition execution [operation *remove_agents*] to be incorporated into each requested scene execution [operation *join*] independently. In case of movements to active scenes, the transition manager informs the scene manager so that this authorises the agent(s) to join the execution as soon as it reaches an access state for its(their) role(s). When the movement aims at a new scene execution, the transition manager informs the institution manager, who creates the new scene execution [operation *create_scene*] by launching a scene manager for it. Thereafter, the agent(s) is(are) incorporated.

5.2.3. Norm management Our approach is that governors manage norms as a rule-based system. In order to construct the rule base, each institutional norm (following the norm schema in (1)) is split into: one rule for the activation of the norm (e.g. accepting an offer in a DA, generating the obligation to pay); and another rule for the fulfilment of obligations (e.g. paying the amount of money due for the accepted offer). The facts of the system are the illocutions uttered and received by the agent. Therefore, a norm N_i splits into:

$$\begin{aligned}
&R1_i : done(s_1, \gamma_1) \wedge \ldots \wedge done(s_m, \gamma_m) \wedge e_1 \wedge \ldots \wedge e_k \rightarrow \\
&assert(obl_1 \ldots obl_p) \wedge addRule(R2'_i, RB) \\
&R2_i : done(s_{m+1}, \gamma_{m+1}) \wedge \ldots \wedge done(s_{m+n}, \gamma_{m+n}) \rightarrow \\
&retract(obl_1 \ldots obl_p) \wedge dropRule(R2_i, RB)
\end{aligned} \tag{2}$$

Rule $R1_i$ corresponds to norm activation: if illocutions $\gamma_1 \ldots \gamma_n$ have been uttered in scenes $s_1, \ldots, s_m$ and events $e_1 \ldots e_k$ are satisfied, then obligations $obl_1 \ldots obl_p$ are added to the set of agent pending obligations [operation *add_obligations*], and a rule to check the obligations fulfilment is added to the rule base via $addRule(R2'_i, RB)$. Notice that the illocution schemes on norm definitions contain variables whose scope is the complete norm. Hence, the bindings of these variables must be taken into account in the rule of the second type added to the rule base. Thus, $R2'_i$ is a particularization of $R2_i$ where variables are replaced by their bound values. Rule $R2_i$ checks whether norm obligations are fulfilled: if illocutions $\gamma_{m+1} \ldots \gamma_{m+n}$ have been uttered in scenes $s_1, \ldots, s_m$, then obligations $obl_1 \ldots obl_p$ are eliminated from the set of agent pending obligations [operation *remove_obligations*], and the rule is removed from the rule base via $dropRule(R2_i, RB)$. Governors only need to add to their rule bases the first type of rules because the second type will be added and removed dynamically in the rule base as obligations are acquired or fulfilled.

In order to manage rules we use the Java Expert System Shell (JESS) (http://herzberg.ca.sandia.gov/jess), a rule engine and scripting environment that permits the creation and management of rule-based systems from JAVA programs. Governors continuously keep the pending obligations of their assigned agents, checking whether their subsequent interactions alter them. At this aim, governors have a thread devoted to manage its interaction with JESS (see figure 3): to add rules and facts into JESS; to run JESS inference engine; and to collect information from JESS when rules are fired. Thereafter, each governor informs its assigned agent about new obligations or fulfilled obligations.

5.3. Example

In order to illustrate how AMELI works, figure 4 shows the monitoring of an execution of the DA market described in section 2. Frame 1 contains a list of the institution's scenes and transitions along with their executions. The list includes a single execution of the *meetingRoom* scene (*id 5*) at state *W1*. Furthermore, there are two different executions of the *tradeRoom* scene: one ongoing execution (id 50 at the initial state), and a finished one (id 34). The figure shows that while five agents (a trade manager –*tradeMgr*, two buyers, and two sellers) have participated in scene execution 34, a single agent (a trade manager) is waiting for buyers and sellers to join in scene execution 50. According to section 5, there is a scene manager agent per ongoing scene execution (e.g. id 5, 50). Besides, no scene manager agent is required any longer for scene execution 34 since it is finished. Furthermore, there is one transition manager agent per transition. Frame 2 depicts the events occurring during scene execution 34: agents' entrance (e.g. label 4), the utterance of valid (e.g. label 6) and wrong (e.g. label 5) illocutions, transitions caused by timeouts (e.g. label 7), agents' exit (e.g. label 8). We must remind the reader that the coordinated activity of the scene manager of the scene execution and the participating agents' governors guarantee that all these events abide by the scene specification. To illustrate the control of AMELI agents, frame 3 visualises an illocution rejected because a constraint in the specification is violated when buyer *BIGWire* attempts at submitting a demand of 0 units at 18 EUR. Since the scene manager evaluates the illocution as not valid, *BIGWire* is informed by its governor about the failed action.

6. Contributions and Related work

Engineering MAS appears as an intricate task. Recently, a remarkable number of MAS methodologies (e.g. GAIA [11], Tropos [6], or [9] to name a few) have been proposed. Although they are based on strong agent-oriented foundations, offering original contributions at the design level, they are unsatisfactory at the development level because of the lack of support to bridge design an implementation. On the other hand, although MAS methodologies agree on the need of adopting an organisational stance, to the best of our knowledge the formal definition of organisation-centered patterns and social structures in general, along with their computational realisation, remain open issues (as noted in [11]). In addition to methodologies, although further agent research has focused on the introduction of social concepts such as organisations or institutions (e.g. [8],[3],[10]), there is no infrastructure supporting their computational realisation.

In this paper we have attempted to make headway in the above-identified shortcomings. Thus, we have presented AMELI, an infrastructure devoted to support the computational realisation of EIs. Notice that AMELI is a key piece that allows us to fill the gap between the design (specification) of EIs and their implementation: while ISLANDER [5] supports the graphical specification of EIs as formalised in [4], AMELI supports their subsequent execution. Notably, an innovative feature of AMELI is that it is of *general purpose* (it can interpret any institution specification), and therefore it can be regarded as *domain-independent*. This feature differs from the work in [2], where specifications of interactions protocols (and not higher-level social structures) must be subsequently compiled into executable protocol moderators. It also differs from Tropos, whose specifications are transformed into agent skeletons that must be extended with code, since AMELI requires neither nor additional programming or compilation since it works as a *specification interpreter*. Notice too that AMELI has been realised as a cooperative multi-agent system that mediates all interactions of agents participating in the institution in order to cope with openness, particularly guaranteeing the enforcement of institutional rules. Finally, it is our belief that AMELI represents a higher (social) level of abstraction than other agent infrastructures such as DARPA COABS (http://coabs.globalinfotek.com) or JADE [1].

References

[1] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with jade. In *Intelligent Agents VII*, number 1571 in LNAI, pages 89–103. Springer-Verlag, 2001.

[2] C. S.-B. C. Hanachi. Protocol moderators as active middle-agents in multi-agent systems. *Journal of Autonomous Agents and Multiagent Systems*, 8(2), March 2004.

[3] V. Dignum. *A Model for Organizational Interaction*. PhD thesis, Dutch Research School for Information and Knowledge Systems, 2004. ISBN 90-393-3568-0.

[4] M. Esteva. *Electronic Institutions: from specification to development*. IIIA PhD Monography. Vol. 19, 2003.

[5] M. Esteva, D. de la Cruz, and C. Sierra. Islander: an electronic institutions editor. In *Proceedings of AAMAS 2002*, pages 1045–1052, 2002.

[6] F. Giunchiglia, J. Mylopoulos, and A. Perini. The tropos software development methodology: Processes. Technical Report 0111-20, ITC-IRST, November 2001.

[7] P. Noriega. *Agent-Mediated Auctions: The Fishmarket Metaphor*. Number 8 in IIIA Phd Monograph. 1997.

[8] H. Parunak and J. Odell. Representing social structures in uml. In *Agent-Oriented Software Engineering II. LNCS 2222*, pages 1–16. Springer-verlag edition, 2002.

[9] A. Sturm, D. Dori, and O. Shehory. Single-model mehtod for specifying multi-agent systems. In *Proceedings of AAMAS 03*, pages 121–128, Melbourne, Australia, 2003.

[10] J. Vazquez and F. Dignum. Modelling electronic organizations. In *Multi-Agent Systems and Applications III*, volume 2691 of *LNAI*, pages 584–593. Springer-verlag edition, 2003.

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