Weaving a Fabric of Socially Aware Agents

[1] Department of Computing, Goldsmiths, University of London,
London SE14 6NW, United Kingdom
dinverno@gold.ac.uk
[2] Department of Informatics, King's College London, Strand,
London WC2R 2LS, United Kingdom
michael.luck@kcl.ac.uk
[3] IIIA, Artificial Intelligence Research Institute,
CSIC, Spanish National Research Council
{pablo,jar,sierra}@iiia.csic.es

Abstract. The expansion of web-enabled social interaction has shed light on social aspects of intelligence that have not been typically studied within the AI paradigm so far. In this context, our aim is to understand what constitutes intelligent social behaviour and to build computational systems that support it. We argue that social intelligence involves socially aware, autonomous individuals that agree on how to accomplish a common endeavour, and then enact such agreements. In particular, we provide a framework with the essential elements for such agreements to be achieved and executed by individuals that meet in an open environment. Such framework sets the foundations to build a computational infrastructure that enables socially aware autonomy.

1 Introduction

The current expansion of social networks (and tools to support them) provides some valuable opportunities to examine new models of *social intelligence*, in particular in situations in which multiple rational entities engage in a common endeavour. In this sense, social intelligence is an emergent property of the interactions between individuals, whether human, machine, or a combination of both. Key to achieving it is the requirement for *social awareness*: the ability of such individuals (henceforth agents) to understand their context in order to act in such a way that interactions with others can be meaningful and effective. In this context, and in seeking to contribute to the development of systems manifesting social intelligence, this paper addresses the underlying infrastructure needed to support social interaction processes.

These processes are usually *open*, in the sense that they involve autonomous individuals that may participate in (or leave) social interactions at will. Such social processes cannot in general be prescribed *a priori* but need to be adaptive, in view both of the autonomy of individual agents, and the openness of systems in which agents might not know each other in advance. In consequence, agents must themselves come to agreements and then put them in practice. Only by manipulating (creating, modifying, exchanging) such agreements in meaningful ways can agents create and execute complex

D. Kinny et al. (Eds.): PRIMA 2011, LNAI 7047, pp. 263–274, 2011.

social processes. It is this ability of agents to operate in environments in which they can understand, participate in, create and modify agreements that we refer to as *social awareness*. We do not discuss how to create, modify or argue about these agreements, but instead provide a framework with the minimal set of elements for such agreements to be achieved and executed.

In doing so, we provide the conceptual and technological foundation for a wide variety of applications. For instance, to build *ad hoc* coordination support environments beyond what, say, Facebook can currently provide, consider how a group of friends might organise their weekend outings. They might join a virtual group to decide what movie to watch, where and when to meet; then agree on what type of food to have, what restaurant to go to and if a reservation is needed choose someone to make it, and have them actually do so, etc. Similarly, we envisage systems that provide an environment in which companies come together as a supply network where a call for tenders results in contracts for production, purchase and delivery of goods or services that are then enacted, and where incidents and failures are dynamically resolved by the interested parties. In a similar vein, a game designer could design and run multi-party distributed role playing games. In more abstract terms, we seek to develop functionalities for agreed-upon regulated social interaction in a flexible, principled way.

While the notion of intelligence has been considered (and indeed formalised) from many perspectives, there has been little work on the *formalisation* of *interaction* of intelligent entities that facilitates social awareness. The structuring of interactions as a static blueprint similar to classical human institutions has been studied in the area of *electronic institutions* [2], and modelling software as an organisation of components with clear interactions among them was pioneered by Gasser [8]. However, the fact that the most important aspect of any interaction is to represent *joint* activities of agents, makes recent developments in process algebras (e.g. [4]) and workflow languages (e.g. [3]) not well suited, as their emphasis is on the programming of hard-wired processes constituting individual agents along with their interactions. In other words, agents are not allowed to autonomously agree on how and when to interact.

Software environments that enable social intelligence need to include the explicit generation and representation of *social interaction process agreements* (for example, jointly starting, jointly finishing or jointly entering an activity), on top of the simpler capacity to work together once the agreements are set. In response, we seek to contribute to this broad goal by providing a formal framework for the management of such interactions. Our framework includes those aspects that we claim are necessary to support the processes of how an activity may be organised and put into action. In particular, we make explicit (i) the requirements for meaningful communication among agents; (ii) the requirements for the set up of coordination or interaction processes; and (iii) the required operations that the environment needs to support social interactions and those that agents need in order to interoperate.

To ensure that our ideas have a clear and unambiguous semantics, but can also be described at the appropriate level of abstraction, we use formal specification techniques developed in software engineering, and give a formal account of the concepts underpinning interoperation in open systems, including the essential data structures and

operations. While the description omits some aspects due to space limitations, a complete version can be found in [1].

The remainder of the paper is organised as follows. Section 2 presents the main data structures supporting the static definition of activities or interaction processes, Section 3 describes the data structures supporting the execution of interaction processes, and the most basic operation that agents may perform. Section 4 concludes.

2 Social Intelligence

Social intelligence, as considered in this paper, essentially arises from *activities* that are carried out by groups of agents. Hence, in our approach to open systems we need to include the basic notion of group meetings, or *scenes*, in which agents interact. We therefore postulate a framework for open systems in which the *activities* of agents are *social*, *decomposable*, *scalable*, *local* and *dialogical*. We consider each of these in turn below in order to motivate our approach.

For any activities to be performed together by groups of agents, those agents must coordinate their individual activities with each other. Such coordination underpins *social intelligence* and, in this sense, activities are *social*, because agents that interact need to be aware, first of others and their roles, and second that certain capacities (or roles) may be required to achieve a particular goal within a common activity. Activities are *decomposable* in the sense that the goals of an agent, and the overall goals of a group of agents, can be decomposed into simple activities whose performance achieves the individual and collective goals. This composition requires the interconnection of *atomic activities* into a *graph* in which the achievement of individual and social goals can thus be associated with particular paths (or to subgraphs) that represent particular combinations of goals.

Openness means that agents may enter and leave a system at any time, potentially causing a large number of agents to interact. To cope with the *scalability* that is required as a result of this, not only is problem decomposition needed, but so is the possibility of *replicating* the enactment of simple activities so that different groups of agents can be allowed to perform the same activity, concurrently, over an enlarging infrastructure. Openness and dynamism cause knowledge about others necessarily to be limited. In consequence, interactions are naturally *local* within subgroups of agents. This locality of interaction supports scalability by establishing bounds on interactions of agents, and also supports security and privacy.

Finally, activities are *dialogical* as they are achieved via agent interactions composed of non-divisible units that occur at discrete instants of time. These units can be modelled as *point-to-point messages* within a *communication language*, and physical actions can be considered as wrapped by appropriate messages of this form.

Now, in order to provide a computational model capturing this view of social intelligence, we have developed a complete, type-checked specification using the Z specification language. Z has been used to specify many different agent and multi-agent systems in the last 15 years (e.g. [5]); due to its computational, functional-programming style semantics [9], Z can provide an unambiguous formal account of a system and its operation as a basis for implementation. In what follows, we present a type-checked subset of

our formal model for social intelligence in Z, detailing the key concepts, and justifying in text the need or those whose formal description is not given. We begin by identifying the languages and primitives required, before building up to the concept of a society by means of interactions achieved through dialogue.

2.1 Languages

Our concern is with specifying open systems in which agent *interactions* are *meaningful*, *contextual*, *consequential* and *regulated*. We handle these properties through different languages and constructs as follows.

Interactions are *meaningful* in the sense that the meaning of the interaction is the same for all participants. When interacting within a common environment, agents need to communicate about their problem domain, and thus an *ontology* that describes this domain is required. We call this the *domain ontology*, specified in terms of a *domain language* in which to express the *purpose and means* of the interaction. Although this shared language might be the result of some process of semantic alignment, for the interaction to be successful it must be meaningful to all participants.

Interactions between agents are *contextual* in the sense that their effects depend on the specific circumstances under which they take place. In fact, interactions generate a history of information exchanges — a sequence of messages between agents playing various roles — that determines the particular context in which new interactions are to be interpreted. The representation of such evolving circumstances requires a *stateful* architecture in which context is explicitly represented. Changes to the context of an interaction occur as actions are executed when agents speak. Since interactions have *consequences* for the context of the participants, an *action language* determines how to update the state of contexts after illocutions occur. For example, if *John* wins an auction for a box of fish, his credit (a variable associated with any buyer) is decreased by his winning bid amount.

Building on the domain language types, the *property language* provides the means to represent attributes of the components of an interaction model. Such attributes might indicate that buyers have credit, or that there can be only one auctioneer in an auction room. Interactions are further *regulated* in being constrained by context. For this reason, a *constraint language* is needed to guarantee that every atomic interaction occurs in the right context. For example, any bid in an English auction must be higher in value than a preceding bid.

Now, to bootstrap our specification, we need a set of basic types that specify agents, roles and time, variables, terms, the formula of the languages already mentioned above (domain, property, constraint and action), whose syntactic features we abstract out in this paper and simply declare as given sets.

$$[Agent, Role, Time]$$
$$[AgentVar, RoleVar, TimeVar]$$
$$[AgentTerm, RoleTerm, TimeTerm]$$
$$[DLFormula, CLFormula, PLFormula, ALFormula]$$

2.2 Roles and Relationships

Our aim is to specify patterns of interaction to enable meaningful interoperation of multiple socially intelligent individuals. To interact successfully, these individuals need to be *socially aware*: they need to know the actions that they and others may take. Of course it is not possible to define interactions at the level of individual agents in advance, so we need an abstraction of an agent in the context of a social setting for which we introduce the notion of *roles*. Roles enable us to describe, design and understand interactions in an abstract and re-usable sense. First, roles define concrete patterns of behaviour; that is, what can be said and to whom. Second, roles have relationships of different sorts that either further restrict behaviour (for example, a committee member cannot perform the actions of the chair) or determine subsumption policies (for example, the chair may take on the responsibilities of any member). Thus, our framework contains roles and any relationships that we may wish to specify between them. Formally, we define these simply in the schema Roles.

Roles

$$roles : \mathbb{P}\, Role$$
$$socialrels : \mathbb{P}(Role \leftrightarrow Role)$$

2.3 Interaction and Communication

Humans communicate through a variety of forms (visual, phonetic, linguistic) but if we are designing open systems to include mixed societies with software agents, then basing it on anything other than the linguistic would seem impossible. So agents in roles must interact by means of communication or, more precisely, by exchanging speech acts. Given this, the template for any such exchange must include a formula from the domain language (to identify the content), and an illocutionary particle (the basic illocutionary force of the communication, such as promising, commanding or asserting). Along with this, we must also provide a minimal context: a time term (to record when the utterance took place), and a sender agent and a receiver agent, both with their associated roles. For example, we might have an *inform* illocutionary particle, with a domain language formula *view(movie, Jaws)*, at time *36487*, from a *proposer* role to a *friend* role (instantiated at some later point with the agents *Alice* and *Bob* as sender and receiver). The syntax is thus similar to that of agent communication languages, FIPA or KQML [7,6], but we only identify the *core* components that are required for our model of social intelligence.

We call this combination of data items an *IllocutionType*, represented formally in the schema below, which first requires introduction of a set for *IllocutionaryParticle*. The schema includes a predicate that asserts that the sender and receiver agents must be distinct, ruling out agents talking to themselves as social interaction.

$[IllocutionaryParticle]$

IllocutionType

$dlformula : DLFormula;$
$illocutionaryparticle : IllocutionaryParticle$
$time : TimeTerm;\ sender, receiver : AgentTerm$
$sendrole, receiverole : RoleTerm$

$sender \neq receiver$

When considering the kinds of interactions that may take place, some elements must be abstracted away. In particular, it is not known which concrete formulae will be exchanged, who will talk to whom or when. However, since interactions are regarded as dialogue patterns, it is known which roles the unknown agents will incarnate in a particular communicative act and the illocutionary force (or particle) that will be used. Thus, we introduce the notion of *Scheme* as a subschema of *IllocutionType*, in which the agents and the time, at least, are abstracted away as variables, where the time term must always be a free variable.

$isavar\_ : \mathbb{P}\, AgentTerm;\ istvar\_ : \mathbb{P}\, TimeTerm$

Scheme

IllocutionType

$isavar\ sender \land isavar\ receiver \land istvar\ time$

An *Illocution* defines what occurs when an agent acts through an *IllocutionType* for which all variables are bound. (In the definition below, we omit the predicate part that would simply state that all variables are bound.)

$Illocution == IllocutionType$

2.4 Conversations and Scenes

We have stated that agents interact by exchanging illocutions within group meetings that contextualise those exchanges. We now make more precise the fact that speech acts are always uttered in a conversation in a particular environment that involves the goals of the participating agents, the roles they are playing, and a particular shared set of variables modelling the properties of the conversation. If we do not group utterances into these conversations (which we refer to as *scenes* to emphasise the fact that agents are taking on specific roles within a conversation), then we could never be able to interpret an utterance. In our view it is the combination of roles, illocutions, and scenes together that provide agents with the necessary social awareness required to interpret messages.

Our example of agents deciding which movie to go see, is just one of several situations that agents may encounter in the larger set of activities involved in actually going to see a movie, including setting a time and place to meet, making travel arrangements, and actually going to the movie. Each of these separate situations can be regarded as a *scene* where agents exchange utterances. When an agent makes a speech act, the state

of the conversation changes, we thus say it moves to a new *conversation place*. At any time during its life, a scene is in one of these conversation places, transitions between which are achieved either by agents uttering illocutions, or eventually by agents *not* uttering anything at all after some time. Thus, a scene is a directed graph of *conversation places*, where the arcs are *labeled* with speech acts, constraints, and actions. The speech act takes the form of an unbound illocution scheme, the set constraints (or pre-conditions) determine what must be satisfied for the speech act to take place, and the sequence of actions (or post-conditions) to update relevant information if and only if the speech act is successful.

First, we provide the formal definitions required, and then explain them.

$$[ConvPlace, SceneName]$$

Label

$$\begin{aligned} &scheme : Scheme \\ &constraints : \mathbb{P}\, CLFormula \\ &actions : \operatorname{seq} ALFormula \end{aligned}$$

Scene

$$\begin{aligned} &sname : SceneName \\ &sceneroles : \mathbb{P}\, Role \\ &limits : Role \pfun \mathbb{P}_1(\mathbb{N}) \\ &places : \mathbb{P}\, ConvPlace \\ &moves : ConvPlace \rel ConvPlace \\ &label : (ConvPlace \times ConvPlace) \pfun Label \\ &startingpoint : ConvPlace \\ &closing : \mathbb{P}\, ConvPlace \\ &access, leaving : Role \pfun (\mathbb{P}\, ConvPlace) \\ \hline &\forall cs : ConvPlace \mid cs \neq startingpoint \bullet \\ &\qquad cs \in (\operatorname{ran}(\{startingpoint\} \ldres moves)\star) \\ &\forall r : Role;\ cs_1 : ConvPlace \mid cs_1 \in (access\ r) \bullet \\ &\qquad \exists cs_2 : closing \bullet (cs_1, cs_2) \in moves\star \\ &closing \cap (\operatorname{dom} moves) = \{\} \end{aligned}$$

The *Scene* schema above contains the following elements. First, it requires a name so that we can identify it, the set of roles that can act within it, and the limits on the number of agents that play those roles (e.g., one meeting coordinator to determine the movie of a cinema visit). Second, we must define the set of conversation places and those moves between them that may occur because of an utterance. Each such move must be labelled with the illocution scheme that needs to take place, a (possibly empty) set of constraints (e.g., to stop someone changing the cinema venue at the last minute when some people may already have left home), and a set of action formulae (e.g., if we buy tickets, then our balance goes down and the cinema's goes up). Third, every scene has a unique starting point but there are several places where the conversation may

legitimately close; conversations are finished at a *closing* place, so no other place can be reached by a move from them; and, in a scene graph, all places must be reachable from the *starting* place, and there must be at least one *closing* place that is reachable from any place. Fourth, in a truly open interoperational framework, agents playing various roles may either join or leave an ongoing conversation at some socially accepted places. Some simple integrity constraints must be satisfied by the specification of a conversation, but they are omitted here for space reasons.

2.5 Weaving the Society

Because agents are concurrent and autonomous, a society amounts to a set of connected conversations. Any social system that respects the changing goals and evolution of behaviours of autonomous agents must thus provide mechanisms for moving freely between scenes, for weaving together these agents and their actions in the various scenes in which they are involved. Thus, now that we have defined scenes, we need some way to connect them so that agents autonomously move between them once a social decision is reached. In our model, the means of doing this is via transitions and arcs: arcs give routes out of scenes and into scenes, and transitions provide synchronisation and choice points for agents as they leave and enter scenes. The resulting network of scenes allows groups of agents to jointly decide whether to start a new scene, join a scene, leave a scene, or close a scene.

As indicated, arcs link scenes to transitions and transitions to scenes, with each arc associated with a set of actions from the action language and constraints from the constraint language, corresponding to preconditions that govern the ability of an agent playing a role to traverse an arc. For example, an agent is seeking to go to the group movie event, must express a preference for a particular movie, and must pay part of a group rate fee in advance. In such a case, there may be an arc to the movie decision scene with the constraint that the agent has voted for a movie, and an action that decreases the agent's balance by the movie fee. Networking scenes is necessary to capture the causal dependencies between scenes including order, synchronisation, parallelism, choice points, creation, change of roles between scenes and so on.

Formally, this is captured in the *Society* schema, which contains the set of all scenes, of which there must be one entry scene and one exit scene, which are distinct. It also contains arcs linking scenes. Then, there is a labelling function (*disjnorm*), which maps each arc to a disjunctive normal form of agent variables and role identifiers, defining the possible (non-empty) set of agents and associated (non-empty) roles that agree on simultaneously traversing the arc. Similarly, a second labelling function (*constraints*) maps arcs to constraints (in our constraint language), which individual agents must fulfill in order to traverse the arc. A third labelling function (*actions*) maps arcs to actions (in our action language), which are triggered when individual agents traverse the arc. In this way, each arc is labelled with (possibly empty) sets of constraints and actions. Finally, to enforce the earlier restrictions on entry and exit scenes, it is not possible to return to the entry scene, nor is it possible to leave the exit scene and return to another scene in the network of scenes.

$$Arc == (Scene \times Scene)$$

Society

$$
\begin{array}{l}
allscenes : \mathbb{P}_1\, Scene \\
entryscene, exitscene : Scene \\
arcs : \mathbb{P}_1\, Arc \\
disjnorm : Arc \nrightarrow \mathbb{P}_1(\mathbb{P}_1(AgentVar \times Role)) \\
constraints : Arc \nrightarrow (\mathbb{P}\, CLFormula) \\
actions : Arc \nrightarrow (\mathbb{P}\, ALFormula) \\
\hline
entryscene \neq exitscene \\
\neg\, (\exists\, a : Arc \bullet second(a) = entryscene) \\
\neg\, (\exists\, a : Arc \bullet first(a) = exitscene)
\end{array}
$$

3 State of an Open System

3.1 Agent Processes

At run time, agents can concurrently take part in multiple scenes; that is, they may perform several activities at the same time. For example, a human agent may participate in a skype call, in a meeting, and may also send and receive text messages at the same time. Similarly, a software agent in an auction house may simultaneously bid in several auctions that run in parallel. Certainly, these various activities can be interrelated in the sense that a text message may influence the agent's behaviour in the meeting and skype call, and the selling price in one auction may influence what an agent chooses to bid in another. In this respect, any model of social intelligence should be able to account, for each agent, for a set of different *processes* that share memory so that performance in one activity may influence performance in others. Each such process represents an agent playing a single role within an instance of a scene, and the number of such processes may change over time as different instances of scenes are created and terminated at different points in time. Thus, we introduce the notion of *agent processes*, which effectively capture the representation of state. In the *AgProc* schema, we define these processes, which have the owning agent, an identifier, the role of the agent, and the set of roles that the agent is allowed to take on at some later time.

$[Id]$

AgProc

$$
\begin{array}{l}
name : Agent \\
place : Id \\
role : Role \\
allowedroles : \mathbb{P}\, Role \\
\hline
role \in allowedroles
\end{array}
$$

Given this, we can give a more precise account of the contextualisation of actions we had mentioned before. In order to define the state of a scene during its execution, we introduce the notion of a *scene instance*, which requires a record of the history of moves

and the binding contexts in which utterances (speech acts) were made. A *FrameContext* represents the instantiation of variables in schemes, a *MoveContext* defines the move and the frame context in which a speech act occurred, and a generic *Context* is the sequence (or history) of such move contexts that have occurred since the beginning of the instantiation of the scene.

$$[FrameContext]$$
$$MoveContext == ConvPlace \times FrameContext \times ConvPlace$$
$$Context == \mathrm{seq}\, MoveContext$$

In addition to an identifier and the conversation context, any scene instance includes the *scene* of which it is an instance, the current conversation place and the set of *agent processes* involved in the scene instance. The predicates in the *SceneInst* schema guarantee that the scene instance is consistent with the data structures of its scene. In particular, we require that the current conversation place to be well-defined (within the conversation places of the scene), and that the number of agents in the scene instance is within the allowable range of the scene.

SceneInst

$$scene : Scene$$
$$position : ConvPlace$$
$$agents : \mathbb{P}\, AgProc$$
$$context : Context$$
$$sid : Id$$
$$exitingAgents : \mathbb{P}\, AgProc$$

$$position \in scene.places$$
$$\forall r : Role \bullet \#(\{a : agents \bullet (a.name, a.role)\} \rhd \{r\}) \in scene.limits\ r$$

3.2 Making a Move in Group Interaction

The last element of our proposal is to identify the operations that a framework enabling the interoperability of socially aware agents needs to support. We have identified many such operations but, due to space constraints, here we only discuss the key operation *Speak*, which processes an atomic action of an agent: a speech act. Another twelve operations — dealing with the environment in which interactions take place (such as initialising an environment or creating a scene instance), or corresponding to voluntary acts of agents (such as joining a scene instance or allowing an agent to change roles) — are dealt with in a similar fashion; details can be found elsewhere [1].

Thus, using the definitions above we can specify how to process a speech act through the *Speak* schema below. In particular, we specify what happens when a speech act i is uttered successfully. Line by line, the *Speak* schema states the following. First, this is an operation that changes the state of the scene instance. We define the variable *nextmoves*, which is the set of potential next conversation places and the associated schemes that label them. The variable *test* is the potential new history that would be defined if a move took place. It is needed to see whether constraints are satisfied. The first predicate

states that we generate the set of pairs of possible next moves from the current position and their associated schemes. The second predicate makes use of the function *proceed*, which takes a set of pairs described in *nextmoves* and an illocution, and returns a frame context and next position if one can be found that matches the illocution. In other words, this ensures that the illocution defines a specific move. The next predicate generates a test context using the generated frame context and destination place. Next, we determine whether the constraints of the scheme are satisfied within this test context (by making use of the predicate *sat*.) The position is updated as is the context, and the actions that label the move to the next position are queued for action.

$$
\begin{array}{l}
proceed : \mathbb{P}(Scheme \times ConvPlace) \rightarrow Illocution \rightarrow \\
\qquad\qquad (FrameContext \times ConvPlace) \\
sat\_ : \mathbb{P}((\mathbb{P}\, CLFormula) \times Context)
\end{array}
$$

Speak

$$
\begin{array}{l}
\Delta SceneInst \\
i? : Illocution \\
nextmoves : \mathbb{P}(Scheme \times ConvPlace) \\
test : \mathrm{seq}(ConvPlace \times FrameContext \times ConvPlace) \\
actionstodo! : \mathrm{seq}\, ALFormula \\
\hline
nextmoves = \{s : Scheme;\ c : ConvPlace \mid \\
\qquad c \in (\mathrm{ran}(\{position\} \lhd scene.moves)) \bullet \\
\qquad\quad ((scene.label(position, c)).scheme, c)\} \\
nextmoves \in (\mathrm{dom}\, proceed) \\
test = context \frown \langle (position, first\ (proceed\ nextmoves\ i?), \\
\qquad\qquad second\ (proceed\ nextmoves\ i?))\rangle \\
sat((scene.label\ (position, position')).constraints, test) \\
position' = second\ (proceed\ nextmoves\ i?) \\
context' = test \\
actionstodo! = (scene.label\ (position, position')).actions
\end{array}
$$

4 Conclusion

In this paper we have explored fundamental intuitions underpinning the notion of social intelligence, and on those grounds have outlined a framework to construct computational environments that support open interoperability among autonomous entities. We discussed the conceptual assumptions, data structures, constructs and functionalities — for joint activity definition and execution, and for agent flow — of the framework and illustrated their formalisation.

In particular, our framework allows us to incorporate new functionality into mixed societies of human and agents. For instance, such functionality might be used by a group of friends in Facebook to coordinate their weekly outings; by an NGO to set-up a fund raising campaign, by a group of companies and services to run a supply network, or by game designers to create and manage web-supported participatory role-playing

games. Although in this paper we have only provided a formalisation of conceptual aspects, the framework may be supported by an associated computational architecture and corresponding development tools and methodologies. By using the Z specification language, which lends itself well to supporting implementations that directly implement specifications, we address these concerns, and provide a foundational basis for different implementations.

This paper thus provides an outline of those core elements that we believe are necessary in any framework that enables such open interoperability. It is an *outline* only due to space constraints, but the complete specification (in which this draws) [1] does provide full detail. As an outline, however, the specification is especially valuable, since it does not overcommit, but allows different reifications of the framework to suit different purposes, and at the same time provides a *minimal* set of core elements. Indeed, we believe that to contend with open, concurrent, regulated and socially aware interoperability, all those features that we have included are necessary. We have thus been careful not to include any accessory materials and have preferred to err on the side of abstraction, rather than commit to features that are dispensable. Finding the right balance here is a challenge, but one we believe we have met in this paper.

Acknoledgements. This work has been supported by the projects EVE(TIN2009-14702-C02-01) and AT (CSD2007-0022). We are all immensely grateful to Carme Roig for her tolerance, patience and support. Finally, thanks to Marc Esteva for discussions on his work on AMELI and electronic institutions in general.

References

1. A framework for communication in open systems. Technical report, `http://www.mediafire.com/?mfpq42e0l94eqa8`
2. Arcos, J.L., Esteva, M., Noriega, P., Rodríguez-Aguilar, J.A., Sierra, C.: Engineering open environments with electronic institutions. Eng. Appl. of AI 18(2), 191–204 (2005)
3. Boile, J., Cardella, M., Blanyalet, S., Juric, M., Carey, S., Chandran, P., Coene, Y., Geminiuc, K., Zirn, M.: BPEL Cookbook. Packt Publishing (2006)
4. Cardelli, L., Gordon, A.: Mobile Ambients. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)
5. D'inverno, M., Luck, M., Georgeff, M.P., Kinny, D., Wooldridge, M.: The dMARS architechure: A specification of the distributed multi-agent reasoning system. Autonomous Agents and Multi-Agent Systems 9(1–2), 5–53 (2004)
6. Finin, T., Labrou, Y., Mayfield, J.: Kqml as an agent communication language. In: Bradshaw, J. (ed.) Software Agents. MIT Press (1995)
7. FIPA: Fipa 97 specification version 2.0 part 2. Tech. rep., FIPA - Foundation for Intelligent Physical Agents (1998)
8. Gasser, L., Braganza, C., Herman, N.: MACE: A flexible test-bed for distributed AI research, pp. 119–152. Pitman (1987)
9. Spivey, M.: The Z Notation, 2nd edn. Prentice Hall (1992)