OpenKnowledge

FP6-027253

Interaction Model Language

Sindhu Joseph[1], Adrain Perreau de Pinninck[1], Dave Robertson[2], Carles Sierra[1], and Chris Walton[2]

[1] Artificial Intelligence Research Institute, IIIA-CSIC, Spain
[2] School of Informatics, University of Edinburgh, UK

Report Version: final
Report Preparation Date: 1.8.2006
Classification: deliverable D1.1
Contract Start Date: 1.1.2006 Duration: 36 months
Project Co-ordinator: University of Edinburgh (David Robertson)
Partners: IIIA(CSIC) Barcelona
Vrije Universiteit Amsterdam
University of Edinburgh
KMI, Open University
University of Southampton
University of Trento

Interaction Model Language Definition

Sindhu Joseph[1], Adrian Perreau de Pinninck[1], Dave Robertson[2],
Carles Sierra[1], Chris Walton[2]

August 1, 2006

1 Artificial Intelligence Research Institute, IIIA-CSIC, Spain

2 Informatics, University of Edinburgh, UK

Abstract

In this deliverable we introduce *ambient LCC*, a language to program interaction models for P2P networks. The language is based on process algebra concepts and is specially designed to support the execution of electronic institutions. An algorithm that automatically translates electronic institution specifications into ambient LCC code is presented and illustrated though an example. Background material on electronic institutions, LCC, and ambient calculus is provided to facilitate the understanding of the language. This document will serve as the reference for the implementation of the interaction model interpreter of the OpenKnowledge project architecture.

Contents

1 Introduction

A computational agent is intended to perform in a similar way to a human agent. That is, the agent should be capable of acting independently and autonomously on behalf of others, either humans or other agents. To accomplish this, it is necessary for the agent to be able to *communicate* and *coordinate* with other humans and agents in order to achieve its goals. This is surprisingly difficult to achieve as we must address such diverse issues as identifying suitable communication partners, communicating our goals, dealing with rogue agents, and coping with different knowledge representations. Many solutions to these issues have been proposed, and in this document we present our own. We believe the advantage of our approach is that it provides an appealing pragmatic solution, which is nonetheless founded on sound theoretical principles.

To address the issues of communication and coordination that we have highlighted, it is useful to take our inspiration from human *social systems*, where these issues are addressed in the real world. Following this approach, it is common to treat a group of agents as a *society* [13]. The agents in the society will typically share a common interest, e.g. solving a particular problem. To become a member of this society, an agent must agree to observe certain rules and conventions. In return, the agent can itself benefit from the society, e.g. the expertise of other agents or the established safeguards for transactions. In this societal view, it becomes possible to define conventions for the agents to follow, and the incentives for agents to operate together are made clear. This in-turn makes the issues of communication and coordination more manageable.

In any society, the rules and conventions are expressed as *social norms*. These norms may often be implicit, but are necessary nonetheless. For example, when engaging in a telephone conversation,

the participants will take turns to speak. If the participants were to talk at the same time, then the conversation would become unintelligible. Therefore, it is these social norms that ensure the smooth running of the society. Returning to computational agents, we can readily define social norms for a group of agents to follow. These rules should be defined in such a way to facilitate the successful interaction between agents. However, the rules should not be overly restrictive in that they affect the autonomy of the agents. It is generally not possible to compel another agent to perform any particular task, rather we must negotiate in order to reach a consensus on a given task. This negotiation process can be shaped by collections of suitable social norms.

A popular way to express the social norms between groups of agents is by means of an explicit *protocol*. A protocol is a statement of a set of social norms. This protocol can be examined in-advance by an agent to inform their decision on whether to join a particular society, and the protocol can be verified to ensure that specific properties hold. The protocol also acts as a guide for the agents to follow once they have joined a society. It is important to note that a protocol only contains operations which are specific to the mechanisms of communication and coordination between agents. This makes it straightforward to understand the operation of the protocol without extraneous details, and makes it possible to verify the protocols using automated means, e.g. model checking [14]. All of the other agent facilities, e.g. the reasoning processes, are encapsulated by *decision procedures* which are external to the protocol. In effect, the decision procedures provide an interface between the communicative and the rational process of the agent. We allow the decision procedures to be private to a particular agent, or shared between a group of agents.

The protocol-based approach to agent communication and coordination has been successfully adopted in the Electronic Institutions (EI) [3], and Lightweight Coordination Calculus (LCC) [15] formalisms. In the EI approach, protocols are expressed diagrammatically using graphs of finite-state automata (i.e. state-charts). By contrast, LCC defines a protocol language which expresses concurrent agent behaviours in a process calculus. Both EI and LCC have specific advantages over the other. In particular, protocols are easier to construct in the EI approach due to the use of diagram-based notation, rather than the mathematical approach of LCC. The EI approach has two levels of abstraction: scenes in which agents communicate, and institutions which are composed of scenes. These two levels of abstraction make it possible to compose large protocols from smaller ones. LCC has only one level of abstraction, which is loosely equivalent to a single scene. Nonetheless, LCC protocols are designed as executable specifications and are therefore easier to implement in real agent systems. LCC protocols are also designed to be completely Peer-to-Peer and do not rely on any central infrastructure as in the case of EI. Finally, LCC has a more powerful theoretical foundation, which is not limited to finite-state systems.

As we have noted, the EI and LCC approaches both have specific advantages. For this reason, our current work is focussed on the provision of a unified approach that combines the advantages of both. In this document we define a theoretical framework that can capture the features of both LCC and EI. This theoretical framework can be considered as an extension of the LCC language with the additional layers of abstraction from EI. Also, we introduce the initial steps for an automatic translation procedure from EI into our new theoretical framework. This will enable us to retain the diagrammatic approach to protocol construction from EI, while obtaining the formal rigour and ease of implementation from LCC. The theoretical basis for our unified approach is the Ambient Calculus [1]. This is a process calculus[1] which gives us the notion of bounded locations (ambients)

[1]Process calculus is a family of related approached to the formal specification of concurrent systems. Process

in addition to the normal concurrency operations.

These ambients give us the ability to formally define scene composition, and the transitions between scenes, which were previously lacking in LCC. This is specially important as breaking protocols into small pieces (or combining small pieces to obtain more complex ones) is a basis software engineering need that ambient calculus facilitates. Furthermore, the EI concept has a hierarchical notion of scene (or activity) that maps easily into the ambient concept. Electronic institution concepts like joining or leaving a scene can naturally be mapped into *in* and *out* operations over ambients, and information models can be represented as accessible variables within the ambient. The notion of state, so important in P2P protocols, can then be lodged in an ambient and move with it. The potential loss of good synchronisation properties that a centralised EI approach has can be also recovered by the ambient synchonisation operations. FInally, the concept of ambient is very lightweight and model checking can be applied to interaction models written in ambient LCC.

We considered a number of already existing alternatives. Dynamic Logic [6] is a powerful specification tool to describe the consequences of (possibly concurrent) actions in the evolution of the state of computer systems. However, no efficient verification techniques exist, and the conceptual distance between the notion of electronic institution inspiring our approach and dynamic logic concepts is far too large. BPEL4WS [16] is an orchestration language for business stateful protocols over web services. However, it lacks the rich social structure and role flow that electronic institutions permit, as well as it does not provide support for norm specification.

The Web Services Choreography Description Language (WS-CDL) http://www.w3.org/TR/ws-cdl-10/ has a number of similarities to the approach that we define here. For example, it allows interactions between participants to be defined using sequence, choice and parallel operations. It also defines specific roles for the participants within the interaction, and permits separate interactions to be composed. However, WS-CDL has no formal semantics, and it is unclear how WS-CDL specifications should be enacted. There is no basis on which to verify interactions, or check that compositions are meaningful. Furthermore, there are no higher-level abstractions, such as scenes and institutions provided in the EI approach. The WS-CDL language is also incomplete and the specification is still under development at the time of writing.

We start this document in Section 2 with background material on electronic institutions, LCC, and Ambient Calculus. The knowledgeable reader can skip this section. Then, in Section 3 we define the syntax for Ambient LCC) which combines LCC and ambient operations. In Section 4 we introduce the algorithm that automates the translation between EI specification language and ambient LCC code. Finally, we provide in Section 5 a complete example in ambient LCC.

calculus enable the high-level description of interactions, communications, and synchronisations between a collection of agents. Furthermore, the algebraic laws of process calculus allow these descriptions to be manipulated and analysed via formal reasoning techniques.

2 Background material

2.1 Electronic Institutions

The promises and functionality that the proposals of Open Systems anticipated in the eighties (e.g. Hewitt [7]) are now ever more pertinent for system development given the pervasiveness of IT and the added accessibility brought about by the World Wide Web. However, the challenges of building open systems are still considerable, not only because of the inherent complexity involved in having adequate interoperation of heterogeneous, independent, distributed, autonomous components, but also because of the significant difficulties of deployment and adoption of the amalgamated systems.

We have been developing a technology to address these challenges.

We do not claim to be dealing with Open Systems in their full complexity, but rather addressing a restricted —albeit significant enough— type of openness: that present in interactions that involve autonomous, independent entities that are willing to conform to a common, explicit, set of interaction conventions. We will call these *a-open systems*[2].

For that type of open systems we have been engineering an artifact that focuses in the interactions and their compliance. We call it *Electronic Institutions* (EIs).

The idea behind EIs is to mirror the roles traditional institutions play in the establishment of "the rules of the game"–a set of conventions that articulate agents' interactions– but in our case applied to agents (humans or software entities) that interact through messages whose (socially relevant) effects are known to interacting parties. The essential roles EIs play are both descriptive and prescriptive: the institution makes the conventions explicit to participants, and it warrants their compliance[3].

EIs —as artifacts— involve a conceptual framework to describe agent interactions as well as an engineering framework to specify and deploy actual interaction environments. We have been developing the EI artifact for some time and advocating that open MAS can be properly designed and implemented with it [9, 5, 11, 4]. Our experiences in the deployment of applications as EIs, e.g. [12, 2] make us confident of the validity of this approach.

In what follows, in fact, we will look into EIs as a framework for developing multiagent systems (MAS). We do so for two reasons, first because a-open systems can be viewed as a type of MAS, where the entities that interoperate in the open system are simply thought of as agents. Secondly, because, in that light, some recent methodologies and conceptual proposals for MAS engineering are then relevant for a-open systems. Our approach, as we shall show, has things in common with some of those methodologies and conceptual proposals, however we believe that it contributes to the engineering of this type of MAS through three salient distinctive features:

1. It is socially-centered, and neutral with respect to the participating agents internals and the application domain of their interactions.

[2]Openness is limited by the *adscription* to the conventions

[3]In terms of Simon's engineering design abstractions, EIs are the –social– interface layer between the problem space the participating systems deal with, on one side, and the internal decision or functional intricacies of the various participating systems, on the other.

2. It has a uniform conceptual framework to manage components and interactions that prevails through the different views (high-level specification, implementation, monitoring, ...) of a given system.

3. It has an interaction-centered methodology that is embedded in a suit of software tools that support the system development cycle from specification to deployment.

Following D.C. North [10], traditional institutions can be viewed as "a set of artificial restrictions that articulate agent interactions". Analogously, when looking at computer-mediated interactions we think of Electronic Institutions as a regulated virtual environment where the relevant interactions among participating entities take place.

This crude picture may become sharper by describing the theoretical components that operationalize it. We start by making some operational assumptions explicit:

1. Participating entities are agents. In the accepted sense of being persistent, identifiable, communication-capable software or humans, capable of adopting commitments.

2. Interactions are repetitive.

3. All interactions are speech acts. That is, any and every interaction is –or is tagged by– a message that has some effect on the shared environment where agents interact.

4. Only illocutions uttered by participating agents have effect on the shared environment.

All things considered these are rather innocent assumptions whose basic purpose is to facilitate the definition of a regulated environment. Assumption 1 is simply a convenient use of terminology that turns EIs into a sort of MAS without loss of content either way. Assumptions 2, 3, and 4, is what we have called the "dialogical stance" by which we conceive interactions as repetitive dialogues. This dialogical stance is mostly a conventional device that brings dialogical notions – and performatives– into our framework, it allows for a convenient intuitive description of many EI features such as *scenes* and *performative structures* but it burdens other –like *scene transitions*– with some artificiality. Assumptions 3 and 4 are needed to operationalize the normative character of the interaction environment.

We may now get into clarifying what we mean by "relevant interactions in a regulated environment". In order to do that we will discuss the three constituent elements of our theory for electronic institutions. Firstly, the *Dialogical Framework* that allows us to express the syntactic aspects of EIs, and the ontology of a particular EI. Then the two other elements that allow us to express the prescriptive aspects of EIs and, in particular, what the social effects of the speech acts are intended to be.

Dialogical Framework

A traditional institution, say an auction house, restricts and gives meaning to interactions participants engage in, and sees to it that the consecuences of any interaction that takes place within the institution actually happen. In an auction house, for example, if a good is being offered, the only action buyers can take is to rise their hand, indicating they take the bid; any other action is meaningless or inadmissible (and interpreted as a silent "no" to the bid). If a buyer wins a bid, the auctioneer will adjudicate the good to the buyer, charge the buyer and pay the seller for it; thus making the interactions involved relevant and meaningful to all participants. If we want to be able to build an electronic auction house we should be able to express that kind of conventions in a way

that can be implemented, adscribed to by independent agents, and –basic to our purpose– in such a way that actual transactions can properly be carried out in the implemented institution.

We need to settle on a common illocutory language that serves to tag all pertinent interactions, or more properly, the valid speech acts.

Formally, we take interactions to be illocutory formulas:

$\iota(speaker, hearer, \varphi, t)$

Speech acts that we use start with an illocutory particle (declare, request, promise) that a *speaker* addresses to a *hearer*, at a time t, and the content φ of the illocution is expressed in some object language whose vocabulary is the EI's ontology.[4]

To fill in these formulae we need vocabulary and grammar, but we also need to refer to speakers and listeners, actions, time. We call all that the *Dialogical Framework* because it includes all what is needed for agents to participate in admissible dialogues in a given EI.

To make clear what are all the available illocutions for agent dialogues in a given institution we define a *dialogical framework* as a tuple:

$DF = \langle O, L, I, R_I, R_E, R_S \rangle$

where

1. O stands for the EI domain ontology;
2. L stands for a content language to express the information exchanged between agents;
3. I is the set of illocutionary particles;
4. R_I is the set of internal roles;
5. R_E is the set of external roles;
6. R_S is the set of relationships over roles.

Recall that, like in a real auction house, a given agent may act as a buyer at some point and as a seller at another, and many agents may act as buyers. This consideration allows us to think of participants adopting *roles*. Roles allow us to abstract from the individuals, the specific agents, that get involved in an institution's activities. Hence, each agent participating in an EI is required to adopt some role. All agents adopting a given role should be guaranteed to have the same rights, duties and opportunities. We differentiate between the internal and the external roles. The internal roles define a set of roles that will be played by *staff* agents which correspond to employees in traditional institutions. Since an EI delegates their services and duties to the internal roles, never an external agent is allowed to play any of them. Finally, we found useful to define relationships among roles, for instance, roles that cannot be played at the very same time, or roles that have some authority over others.

Through this DF we have all utterances that may ever be admissible in a given EI, we now turn to the task of characterising admissible dialogues.

[4]We take a strong nominalistic view, the institutional ontology is made of every entity referred to in any admissible speech act or in any of the norms (conventions) that govern those acts and their consequences.

Performative Structure: Scenes and Transitions

We assumed repetitive interactions are typical of institutions. For instance, in an auction house every new good is auctioned out in the same way that the previous one was offered, and the previous, ... Bidding rounds are repeated dialogues very much like the scenes in a play: actors playing given roles (buyers, auctioneer) repeat their lines every time they get into the same *scene*. But in theater as in our auction house example, interactions are usually rather involved. In our example of an auction house: goods are introduced, goods are sold, goods are payed for, sellers get their money, ... As in a play, some scenes precede others, some scenes get repeated; the play has a *performative* structure.

Likewise, activities in an electronic institution are organised in a *performative structure* as the composition of multiple, distinct, and possibly concurrent, dialogical activities, each one involving different groups of agents playing different roles. Interactions between agents are articulated through recurrent dialogues which we call *scenes*, each scene following some type of conversation protocol. The protocol of each scene restricts the possible dialogical interactions between roles. Scenes represent the context in which the uttered illocutions must be interpreted. Consequently, the same message in different contexts may certainly have different meanings. A distinguishing feature of scenes is that their participants may change, as agents are allowed either to enter or to leave a scene at some particular moments (states) depending on their role.

A scene protocol is specified by a directed graph whose nodes represent the different states of a dialogical interaction between roles. Its arcs are labelled with illocution schemata from the scene's dialogical framework (whose sender, receiver and content may contain variables) or timeouts.

More formally, a *Scene* is a tuple:

$s = \langle R, DF_S, W, w_0, W_f, (WA_r)_{r\in R}, (WE_r)_{r\in R}, \Theta, \lambda, min, Max\rangle$, where

1. R is the set of scene roles involved in that scene;
2. DF_S is the restriction to the scene s of the EI dialogical framework as defined above;
3. W is the set of scene states;
4. $w_0 \in W$ is the initial state;
5. $W_f \subseteq W$ is the set of final states;
6. $(WA_r)_{r\in R} \subseteq W$ is a family of sets such that WA_r stands for the set of access states for role $r \in R$;
7. $(WE_r)_{r\in R} \subseteq W$ is a family of non-empty sets such that WE_r stands for the set of exit states for role $r \in R$;
8. $\Theta \subseteq W \times W$ is a set of directed edges;
9. $\lambda : \Theta \longrightarrow L$ is a labelling function, where L can be a timeout, or an illocution schemata and a list of constraints;
10. $min, Max : R \longrightarrow I\!N$ $min(r)$ and $Max(r)$ return the minimum and maximum number of agents that must and can play role $r \in R$.

At execution time agents interact by uttering grounded illocutions matching the specified illocution schemata, and so binding their variables to values, building up the *scene context*. Moreover, arcs labelled with illocution schemata may have constraints attached, based on the scene context, to impose restrictions on the paths that the scene execution can follow. For instance, in a scene auctioning goods following the English auction protocol, we can specify by means of constraints that buyers can only submit bids greater than the last one (where values are being bound in the scene context).

While a scene defines a particular dialogical environment, the causal, temporal and other content relationships among scenes are expressed through a special type of scene we call *transitions*. The interlacing of regular scenes and transitions is captured through the *Performative Structure* of the EI.

A performative structure can be seen as a network of scenes, whose connections are mediated by transitions, and determines the role-flow policy among the different scenes by showing *how* agents, depending on their roles, may get into different scenes (other conversations), and showing *when* new scenes (conversations) will be started.

In all EIs we assume that there is always an initial and a final scene, which are the entry and exit points of the institution. We also allow, at run time, to have multiple instances of the same scene.

Technically, we define a *Performative Structure* as follows:

$PS = \langle S, T, s_0, s_\Omega, E, f_L, f_T, f_E^O, C, ML, \mu \rangle$, where

1. S is a set of scenes;
2. T is a set of transitions;
3. $s_0 \in S$ is the *initial* scene;
4. $s_\Omega \in S$ is the *final* scene;
5. $E = E^I \bigcup E^O$ 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;
6. $f_L : E \longrightarrow DNF_{2^{V_A \times R}}$ maps each arc to a disjunctive normal form of pairs of agent variable and role identifier representing the arc label;
7. $f_T : T \longrightarrow \mathcal{T}$ maps each transition to its type;
8. $f_E^O : E^O \longrightarrow \mathcal{E}$ maps each arc to its type (one, some, all or new);
9. $C : E \longrightarrow ML$ maps each arc to a meta-language expression of type boolean, i.e. a formula representing the arc's constraints that agents must satisfy to traverse the arc;
10. ML is a meta-language.
11. $\mu : S \longrightarrow \{0, 1\}$ states whether a scene can be multiply instantiated at run time or not.

Transitions can be thought of as gateways between scenes or as a change of conversation. At run-time, transitions can be seen as a type of router in the context of a performative structure. We represent them as a canonical type of scene with arcs incoming from scenes and arcs outgoing to scenes (see the little triangles and half-circles connecting scenes –pictured as boxes– in figure 1).

Labels on the directed arcs determine which agents, depending on their roles, can progress from scenes to transitions, or from transitions to scenes.

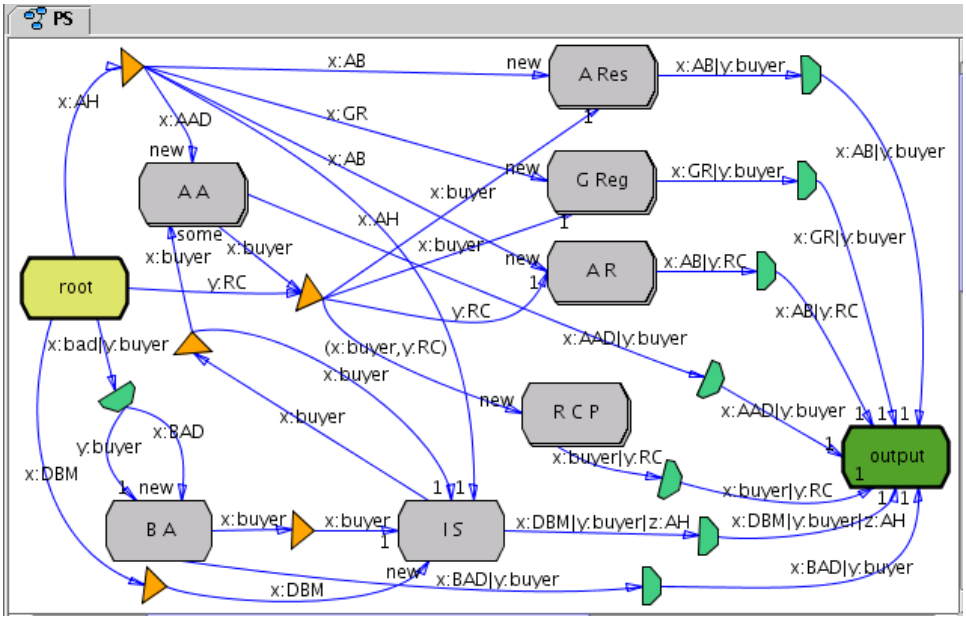

Figure 1: Example of Performative Structure

There are different types of transitions and different types of outgoing arcs. The transition type allows one to express choice points (*Or* transitions) for agents to choose which target scenes to enter, or synchronisation/parallelisation points (*And* transitions) that force agents to synchronise before progressing to different scenes in parallel. On arcs from a transition to a target scene, type determines whether agents following that arc will incorporate into *one*, *some* or *all* of the current executions of the target scene, or if they will go to a *newly* created execution of the target scene.

Norms and Commitments

So far we have dealt with the way interactions within an EI can be defined and organized, we now need to say how they are going to have the intended effect.

We start by noting that an institution (electronic or otherwise) is concerned only by whatever is regulated to happen inside it. Agreements, misconducts, or whatever else that happens outside is in principle disregarded or impertinent to the institution. The main purpose of the institution is to make sure that what happens inside has the effects participants have agreed to.

Actions within an institution, we said, are speech acts. Those speech acts that are made as prescribed by the performative structure of an institution (during an enactment) create obligations or socially binding commitments whose fulfillment is warranted by the institution. We make such intended effects of commitments explicit through what we have been calling *normative rules*.

We define two predicates that will allow us to express the connection between illocutions and norms:

- $uttered(s, w, \mathbf{i})$ denoting that a grounded illocution unifying with the illocution scheme $\mathbf{i}$ has been uttered at state w of scene s.
- $uttered(s, \mathbf{i})$ denoting that a grounded illocution unifying with the illocution scheme $\mathbf{i}$ has been uttered at some (unspecified) state of scene s.

Therefore, we can refer to the utterance of an illocution within a scene or when a scene execution is at a specific state.

Normative rules are first-order formulae of the form

$$\left(\bigwedge_{j=1}^{n} uttered(s_j, [w_{k_j}], \mathbf{i}_{l_j}) \wedge \bigwedge_{k=0}^{m} e_k\right) \rightarrow \left(\bigwedge_{j=1}^{n'} uttered(s'_j, [w'_{k_j}], \mathbf{i}'_{l_j}) \wedge \bigwedge_{k=0}^{m'} e'_k\right)$$

where s_j, s'_j are scene identifiers, w_{k_j}, w'_{k_j} are states of s_j and s'_j respectively; $\mathbf{i}_{l_j}, \mathbf{i}'_{l_j}$ are illocution schemata l_j of scenes s_j and s'_j respectively, and e_k, e'_k are boolean expressions over variables from the illocution schemata $\mathbf{i}_{l_j}$ and $\mathbf{i}'_{l_j}$, respectively.

The intuitive meaning of normative rules is that if grounded illocutions matching $\mathbf{i}_{l_1}, \ldots, \mathbf{i}_{l_n}$ are uttered in the corresponding scene states and the expressions $e_1, \ldots, e_m$ are satisfied, then grounded illocutions matching $\mathbf{i}'_{l_1}, \ldots, \mathbf{i}'_{l'_n}$ satisfying the expressions $e'_1, \ldots, e'_{m'}$ must be uttered in the corresponding scene states.

Notice that $\mathbf{i}'_{l_1}, \ldots, \mathbf{i}'_{l'_n}$ can be regarded as *obligations* that agents acquire where the antecedent of the normative rule is satisfied. Therefore, agents must utter grounded illocutions matching these illocutions schemata and satisfying $e'_1, \ldots, e'_{m'}$, in the corresponding scenes in order to fulfil the normative rule.

Paraphrasing what we have done, the notions we introduce picture the regulatory structure of an EI as a "workflow" (performative structure) of multiagent protocols (scenes) along with a collection of (normative) rules that can be triggered off by agents' actions (speech acts).

Notice that the formalisation of an EI focuses on macro-level (societal) aspects, instead of on micro-level (internal) aspects of agents. Notice also that the whole framework induces an interaction-centered perspective for the design and testing methodologies. Further details about electronic institutions can be found at http://e-institutor.iiia.csic.es.

2.2 The Lightweight Coordination Calculus (LCC)

LCC can be considered as a heavily-sugared variant of the π-calculus [8] with an asynchronous semantics. The extensions to the core calculus are designed to make the language more suited to the concepts found in multi-agent systems and dialogues. LCC was designed specifically for expressing Peer-to-Peer (P2P) style interactions within multi-agent systems, i.e. without any central control.

The abstract syntax of LCC is presented in Figure 2. There is also a concrete XML-based syntax, which is used in the implementation of LCC. However, we only show the abstract syntax here as it is useful to explain the language without the extra syntactic baggage of XML.

$$
\begin{array}{rcl}
Framework & := & \{Clause, \ldots\} \\
Clause & := & Agent :: Dn \\
Agent & := & a(Type, Id) \\
Dn & := & Agent \mid Message \mid Dn\ then\ Dn \mid Dn\ or\ Dn \mid Dn\ par\ Dn \mid null \leftarrow C \\
Message & := & M \Rightarrow Agent \mid M \Rightarrow Agent \leftarrow C \mid M \Leftarrow Agent \mid C \leftarrow M \Leftarrow Agent \\
C & := & Term \mid C \wedge C \mid C \vee C \\
Type & := & Term \\
M & := & Term
\end{array}
$$

Where *null* denotes an event which does not involve message passing; $Term$ is a structured term and Id is either a variable or a unique identifier for the agent.

Figure 2: Abstract Syntax of LCC.

There are five key syntactic categories in the definition, namely: $Framework$, $Clause$, $Agent$, Dn (Definition), and $Message$. These categories have the following meanings. A $Framework$, which bounds an interaction in our definition, comprises a set of clauses. Each $Clause$ corresponds to an agent, and each agent has a unique name a and a $Type$ which defines the role of the agent. The interactions that the agent must perform are given by a definition Dn. These definitions may be composed as sequences (*then*), choices (*or*), or in parallel (*par*). The actual interactions between agents are given by $Message$ definitions. Messages involve sending ($\Rightarrow$) or receiving ($\Leftarrow$) of terms M from another agent, and these exchanges may be constrained by C. The different kinds of operations in LCC are illustrated graphically in Figure 3.

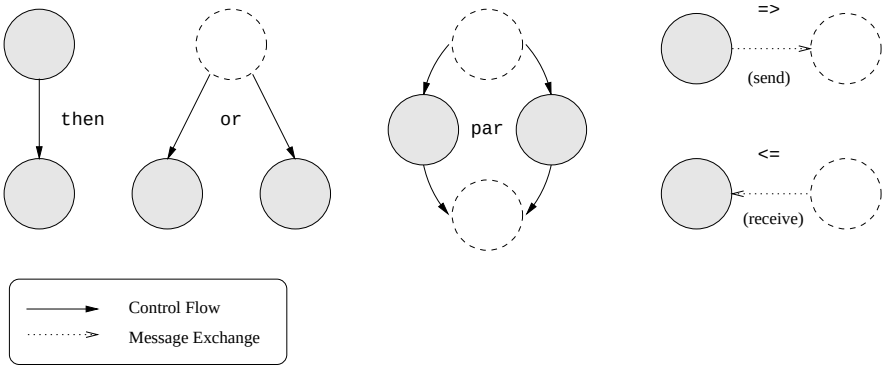

Figure 3: LCC Operations.

The LCC language ensures coherence of interaction between agents by imposing constraints relating to the messages they send and receive in their chosen roles. The clauses are arranged so that, although the constraints on each role are independent of others, the ensemble of clauses operates to give the desired overall behaviour.

As an example, the LCC fragment shown in Figure 4 places two constraints on the variable X: the first ($p(X)$)is a condition on the agent in role $r1$ sending the message $offer(X)$ and the second

$(q(X))$ is a condition on the agent in role $r2$ sending message $accept(X)$ in reply. By (separately) satisfying $p(X)$ and $q(X)$ the agents mutually constrain the variable X.

$$a(r1, A_1) :: offer(X) \Rightarrow a(r2, A_2) \leftarrow p(X) \ then \ accept(X) \Leftarrow a(r2, A_2)$$
$$a(r2, A_2) :: offer(X) \Leftarrow a(r1, A_1) \ then \ accept(X) \Rightarrow a(r1, A_1) \leftarrow q(X)$$

Figure 4: LCC Constraints.

LCC allows two options for satisfying constraints:

- Internally according to whatever knowledge and reasoning strategies it possesses. This is the normal assumption in most multi-agent systems, yet it is not always ideal. In particular we sometimes would like to use knowledge specifically for a social interaction but not require an agent to internalise it (e.g. if that knowledge might be inconsistent with an agent's own beliefs). In such cases LCC offers a second option:
- Externally using a set of Horn clauses defining common knowledge assumed for the purposes of the interaction. Like the LCC protocols themselves, this common knowledge is passed between agents along with messages during interaction so it is ephemeral - lasting only as long as the interaction.

This completes our description of the LCC syntax. Further details on the formal semantics of LCC can be found in [15]. In summary, LCC provides four key features which enable us to define communication and coordination in multiagent systems:

1. Agents are assigned specific roles for interactions.
2. Protocols specify patterns of interaction, using sequences, choices, and parallel composition.
3. The actual interaction between agents is accomplished by message passing.
4. The flow of interactions is controlled by constraints, which correspond to decisions and computations of the agent.

2.3 A Calculus of Mobile Ambients

The ambient calculus was developed as a way to express mobile computation. It can also be considered as an extension of the basic operators of the π-calculus [8]. The inspiration behind the ambient calculus is the observation that many aspects of mobility involve administrative considerations. For example, the authorisation to enter or exit a domain, and the permission to execute code in a particular domain. These issues were principally motivated by the needs of mobile devices. However, they are very similar to the issues faced by agents in an open environment, e.g. the Internet.

The ambient calculus addresses the administrative considerations by defining an ambient (informally) as a "bounded space where computation happens". The crucial point is the existence of a boundary, which determines what is inside and outside the ambient. This boundary is analogous to a firewall. Ambients can also be nested, leading to an administrative hierarchy, which is a commonly occurring structure on the Internet, e.g. intra-nets, and demilitarised zones. An ambient is also something that can be moved. For example, to represent a computer or agent moving from one place to another.

P, Q, R	::=	0	Nil
		$P \mid Q$	Parallel Composition
		$M[P]$	Ambient
		$!P$	Replication
		$M.P$	Capability Action
		$(n).P$	Input Action
		$\langle M \rangle$	Output Action
M	::=	n	Name
		in M	can enter into M
		out M	can exit out of M
		open M	can open M
		ϵ	null
		$M.M'$	composite

Figure 5: Ambient Calculus Syntax.

More precisely, each ambient has a name, a collection of local agents that run directly within the ambient, and a collection of sub-ambients. The abstract syntax of the ambient calculus is shown in Figure 5. The syntactic categories are processes (P, Q, and R), and capabilities (M). A process is analogous to an individual agent. A process may be placed inside an ambient, may be replicated, and may be composed in parallel with another process, which means that the processes execute together. We write $n[P]$ to denote an ambient named n, which contains the process (i.e. agent) P.

The key to the ambient calculus is the definition of capabilities M for processes, which are described by actions. These capabilities allow things to happen within ambients. In particular, there are four key actions that a process can perform, given that it has the required capability. These actions are defined by the transition rules below:

1. Entering an ambient: $n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q]|R]$

2. Exiting an ambient: $m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$

3. Opening an ambient: $open\ n.P \mid n[Q] \rightarrow P \mid Q$

4. Communication: $(n).P \mid \langle M \rangle \rightarrow P\{n \leftarrow M\}$

The *in* action is used by a process to enter an ambient, i.e. to cross the administrative boundary. The result is that the process (and its enclosing ambient) move from the current ambient to the ambient named in the action. Conversely, the *out* action is used by a process to exit an ambient. The result is that the process will move outside the ambient. This may place the process (and its enclosing ambient) in a parent ambient, or completely outside, depending on the ambient hierarchy. The *open* action is used to remove an ambient boundary, i.e. to destroy an ambient. Finally, processes may exchange names by input () and output $\langle\rangle$ actions, which are analogous to message passing communication. The first three operations are illustrated in Figure 6.

This completes our description of the ambient calculus. Further details on the formal definition can

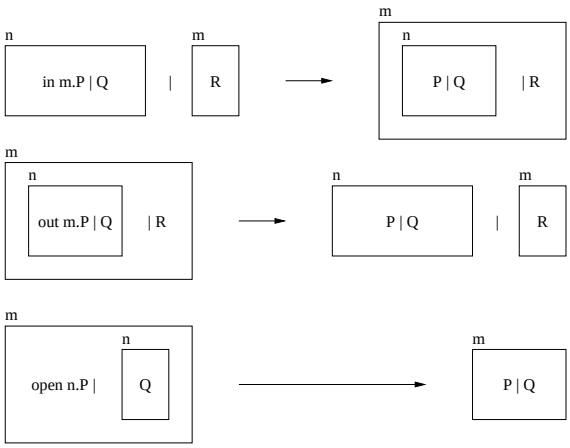

Figure 6: Ambient Operations.

be found in [1]. In summary, the ambient calculus contains four key features that will enable us to compose protocols in useful and meaningful ways:

1. Ambients are used to represent hierarchies and boundaries between locations.
2. Mobility is captured by the movement of agents between ambients.
3. Security is represented by capabilities, which determine the ability or inability of an agent to cross a boundary.
4. Interaction between agents is possible by the use of shared locations with a common boundary, i.e. agents can communicate only within the same ambient.

3 Ambient LCC

Ambient LCC inherits the notion of ambient, the most important extension to LCC, from the theory of ambient calculus. Most of the properties of ambient as defined by Cardelli stands valid in the context of ambient LCC as well. The following highlights the structure and properties in ambient LCC.

3.1 Ambient Structure

An ambient in Ambient LCC consists of two logical components, an ambient specification and an execution environment. The ambient specification specifies the signature of the ambient. It provides the details of what are the parameter values that define the ambient. This may include information such as how many agent processes can be active in an ambient at a given time, what are the entry and exit conditions for agent processes and what are the rules governing communication and message passing within the ambient environment and so on. The execution environment provides

the execution state of the ambient to hold the execution parameters. The execution environment is the runtime counter part of the ambient specification, and follows the specification whenever runtime decisions are taken. A property of the execution environment is that it can be created and destroyed at execution time.

An ambient is also defined recursively to allow layers of abstraction in the protocol definition. This can be mapped into a tree structure, where there is a top most ambient which is the root ambient, and at any layer, there are one or more ambient structures. At each layer, ambients correspond to bounded places, where processes can interact, with ambients providing the context for interaction.

3.2 Ambient Movement

A process entering an ambient must necessarily enter through the top layer of the ambient tree, and can successively traverse the tree structure. A process *in* any ambient node in the tree is necessarily in all the parent ambient nodes. Similarly if a process wants to come *out* of a parent ambient, it should do so only after it exits the child ambient. Processes are also given capabilities to create and destroy ambients upon satisfaction of certain conditions.

Another concept that needs a mention here is the property of processes *holding* ambients. As Ambient LCC works in a distributed setting, there is no assumption of a global infrastructure. As the spirit of the distributed protocol view is carried on to ambient LCC, there will be no infrastructure to hold the ambient execution environment. Thus processes are the only entities having state and hence capable of holding ambients. Hence, at any given point in time, all the ambient execution environments reside in some process. That is, the process that holds it is the only one allowed to modify it. This restricted update mechanism is used to implement the synchronization mechanism for agent communication (see Section 4.3).

3.3 Agent ambients

There is a special kind of ambient defined in the language, associated with agent processes. Though agents can be naturally associated with processes, an *agent ambient* is meant to keep the state of an a*gent process*. Most of the computation happens in ambients that may contain more than one agent, and the state of these computations are preserved in the execution environment of the enclosing ambient. Yet there are certain parameters that are relevant to an agent which need to be preserved throughout the life of an agent. Otherwise, as the enclosing ambients can be destroyed after its execution is finished, there would be no way of preserving the agent specific information throughout. An example of such a parameter is the credit of an agent in an auction setting, when a credit update happens, if the parameter would be part of an ambient execution environment, it will be lost when the ambient execution gets terminated. Whereas the agent needs this value to use in other succeeding ambients as it moves from one ambient to another. The agent ambient is there precisely for this purpose, to store the state of agent relevant parameters in a persistent manner. Thus, in this context, agents are treated as special ambients with a process and an ambient around them. The agent ambients do not get separated from the agent processes at any point during the life of the process and always move along the process. In this sense these are special ambients around

agent processes. Yet in treatment of the ambients, a uniform syntax is adopted which includes the agent ambients, though certain operations are not allowed in the agent ambients. For instance, no *in* operation is allowed over an agent ambient, it would represent that some external code would get access to the private variables of an agent and might be allowed to interfere with the internal decision making machinery, which we definitely don't want to happen.

3.4 Conceptual deviations from original ambient

There are though certain notions within the ambient LCC which depart from the original ambient concepts. The first of such deviations we just discussed is the concept of agent ambients. In the ambient calculus, there is no such special ambient around a process, and it is very relevant in our context. Also, while it is true that agents can move in and out of ambients, it is also true that agents hold the ambients. That is, at any time each ambient is held by one and only one of the participating agents. This is to embed the necessary synchronization capabilities in an inherently distributed interaction model.

Another deviation which is related to the previous is that, while there are capabilities provided as part of the ambient, such as *in*, *out* and capabilities for accessing the variables of the ambient, there are also much restriction on who is authorized to perform this. There is no notion of accessing an ambient from anywhere outside the ambient. Within the ambient, there are two types of members. Any agent process, that has moved *in* an ambient variables is a *primary member* of the ambient. They are able to access the ambient and view the other members of the ambient. The second type is the *executive member* of the ambient. The agent (process) currently holding the ambient belongs to this category. At anytime there is only one unique member in this second category. Along with normal access capabilities this membership gives the agent other capabilities.

An agent holding the ambient can:

- Give access permission for new agents to move *in* the ambient
- Give permission for a member to move *out* of the ambient
- Change the state of the ambient by adding, deleting, or modifying variables
- Pass the ambient to another member agent in the ambient that will become the holder

3.5 Ambient LCC syntax

The previous section dealt with the conceptual background of ambient LCC. The more formal specification of the syntax and functional properties along with notations and conventions are discussed in this section. A detailed discussion on LCC specific syntax is omitted here and can be referred to in the section on LCC and other literature [15]. The ambient LCC syntax is as follows:

$$
\begin{array}{lll}
Framework & ::= & \langle \Delta, Clause^n \rangle \\
Clause & ::= & \mathcal{A} :: Def \\
\mathcal{A} & ::= & a(Id_{\mathcal{A}}, Id_{\mathcal{R}}, Id_{\Delta}) \\
Def & ::= & Action \mid \mathcal{A} \mid [E \leftarrow] Def [\leftarrow C] \mid Def \text{ or } Def \mid Def \text{ then } Def \mid Def \text{ par } Def
\end{array}
$$

$$
\begin{array}{lll}
\Delta & ::= & \langle Id_\Delta, \tau_\Delta \rangle \mid Id_\Delta \mid \Delta(\Delta^n) \\
Action & ::= & Message \mid Op_\Delta \mid E \mid timeout(n) \\
Message & ::= & M \Rightarrow \mathcal{A} \mid M \Leftarrow \mathcal{A} \\
Op_\Delta & ::= & new\ (\langle Id_\Delta, \tau_\Delta \rangle, Id_\delta) \mid new\ (Id_\Delta, Id_\delta) \mid in\ Id_\delta(\mathcal{R}) \mid out\ Id_\delta(\mathcal{R}) \mid open\ Id_\delta \\
\tau_\Delta & ::= & Spec_\Delta \\
C & ::= & get(Term, V) \mid Term \mid C \wedge C \mid C \vee C \\
E & ::= & put(V, Term) \mid C \mid P(V^n, V^n) \\
M & ::= & Term : \tau \\
V & ::= & variable[: \tau] \\
n & ::= & integer
\end{array}
$$

An ambient is represented by the symbol Δ. The agents and roles are denoted by $\mathcal{A}$ and $\mathcal{R}$ respectively. $Id_\mathcal{A}, Id_\mathcal{R}, Id_\Delta$ respectively represent the unique identification of an agent, role and ambient, while Id_δ uniquely determines the specific ambient instance(execution environment). τ represents a type, which can be either an ambient type or the type associated with a term or a variable.

In ambient LCC, a *framework* is the combination of an ambient and a number of clauses. This view is the main extension that is brought to LCC, where there was only the notion of agents and no entity above the agent layer. Thus there are six key syntactic categories in the definition, apart from the five categories that exist in the LCC language, the additional category introduced in ambient LCC is the *Ambient*. Following describes the ambient syntax and the changes induced in other categories by the introduction of ambient into the language in more detail.

- **Ambient Definition**

  An ambient Δ can be defined as $\langle Id_\Delta, \tau_\Delta \rangle \mid Id_\Delta \mid \Delta(\Delta^n)$. Δ consists of the type of the ambient τ_Δ and the ambient identification Id_Δ. There is also a provision for defining an ambient inside another ambient through the definition of $\Delta(\Delta^n)$.The following explains the details of ambient definition.

  The type of the ambient specifies the signature of the ambient. Intuitively, τ_Δ is the set of parameters that define the ambient and that specify the norms of access and conduct within an ambient. τ_Δ is normally derived from the interaction model specification and hence is an interaction model dependent parameter. An example of τ_Δ for electronic institutions is given in section 4.2.1

  Ambients can be defined recursively to include the notion of an ambient inside another ambient. This permits us to define an ambient structure such as $\langle Id_{ps}, \tau_{ps} \rangle (Id_{s_1}, Id_{s_2}, Id_{s_3})$. By referring to an ambient identifier we can easily build complex ambients. The following example illustrates how this can be achieved:

  $\langle Id_{ps}, \tau_{ps} \rangle (\langle Id_{ps_1}, \tau_{ps_1} \rangle (\langle Id_{s_a}, \tau_{s_a} \rangle), Id_{s_a}, Id_{s_b}, \langle Id_{ps_2}, \tau_{ps_2} \rangle (\langle Id_{s_b}, \tau_{s_b} \rangle, \langle Id_{s_c}, \tau_{s_c} \rangle))$

  Graphically this could be represented as seen in Figure 3.5.

- **Agent Definition**
  In the light of the ambient addition, there are slight variations in the interpretation of the

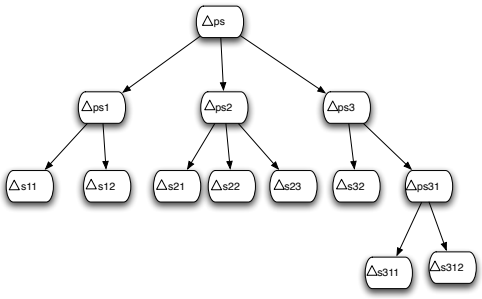

Figure 7: Hierarchical representation of Nested Ambients

other categories. A $Framework$, which still bounds an interaction now comprises an ambient and a set of clauses. Each $Clause$ corresponds to an agent in an ambient. An agent is redefined in the new language as an entity having a unique identifier $\mathcal{A}$ enacting a role $\mathcal{R}$ in an ambient Δ and is represented as $a(Id_{\mathcal{A}}, Id_{\mathcal{R}}, Id_{\Delta})$. They are subsequently defined as agents enacting certain roles in specific ambient instances, or carrying out actions, upon satisfying certain conditions (preconditions C) and has certain effects (post conditions E).

The interactions that the agent must perform are given by a definition Def. These definitions may be composed as sequences (*then*), choices (*or*), or in parallel (*par*). Additionally the definitions are optionally constrained by a precondition and effected by a post condition.

- **Action Definition**
  In the new definition, the fifth syntactic category is replaced by an enclosing category namely *Action*. There are two kinds of *Actions*, $Messages$ form one category used as an interaction mechanism among agents. Another category is the *ambient operations* that an agent can perform to function effectively within the *ambient* framework. The actual interactions between agents are given by $Message$ definitions. Messages involve sending ($\Rightarrow$) or receiving ($\Leftarrow$) of terms M from another agent and the process is made generic by the definition $[E \leftarrow]Def[\leftarrow C]$. In this context, C stands for the preconditions that an agent needs to satisfy before sending any message. C is further defined to accommodate most possible variations in preconditions. There is a specific operation $get(term, V)$ to access the execution variables of the ambient instance which will be explained later in the section. The other possible preconditions can include the use of external functions all of which is encapsulated by the generic definition of $term$ referring to a *prolog term*.

  E represents in general the effect of an action that an agent has performed. This can be a set of actions that needs to be performed as the effect of the previous action such as $put(V, Term)$, or a set of states that becomes true such as $get(Term, V)$. Thus syntactically C (preconditions for an action) and E (post effects of an action) have similar structure, though they conceptually stand for different entities.

  The definition also includes the possibility of combining more than one condition in a conjunctive or disjunctive manner. In summary the different kinds of message operations in Ambient LCC remain the same as that in LCC, with the only difference that they all must be carried

out in the context of an ambient now.

The actual ambient operations are:

- Creating an ambient type and instance at the same time *new* $(\langle Id_\Delta, \tau_\Delta \rangle, Id_\delta)$,
- Creating an ambient instance of a predefined ambient type *new* (Id_Δ, Id_δ),
- Moving into an ambient instance *in* $Id_\delta(r)$
- Moving out of an ambient instance *out* $Id_\delta(r)$
- Opening an ambient instance *open* Id_δ.

In the above syntax, Id_δ represents the unique identifier for the ambient instance (execution environment).

new $(\langle Id_\Delta, \tau_\Delta \rangle, Id_\delta)$ is two operations encapsulated into one construct. *new* first creates a new ambient and declares its type as τ_Δ. Then the id, Id_Δ is returned as the id of the newly created ambient. Using this new ambient, an instance of the same is created and finally the id of the instance Id_δ is returned. Thus in this case, both the ambient id and the instance id are returned as the result of the *new* operation. Whereas the *new* (Id_Δ, Id_δ) takes an existing ambient Id_Δ and creates an instance of this ambient, and returns the id Id_δ of the newly created instance.

The *in* and the *out* operations manage the movement of the agent across the ambients. Upon satisfying a set of preconditions, as specified in τ_{Delta}, *in* $Id_\delta(r)$ will let the agent executing the operation, inside an ambient instance. Here the mention of the agent role, r, is included as there may be a role change between the execution of subsequent protocol bits. However only the agent executing the *in* operation can move into the ambient instance. That is the agent identifier needs to remain the same. The minimum criteria to access variables inside an ambient instance with identity Id_δ is to be a member of Id_δ, which is precisely what the *in* construct provides. *out* $Id_\delta(r)$ is similar. *open* Id_δ destroys the ambient instance δ. This is to provide a logical end of an ambient execution environment. The intuition behind this *open* is that after a protocol bit is run to termination, which was contained in the ambient instance, there is little meaning in the existence of the container ambient. When new agents want to execute the same protocol, a new container ambient will be created again and get initialized with a fresh set of variables and participating agents. In ambient LCC the *open* construct differs a bit from the open construct in ambient calculus. In ambient calculus, an open operation lets all the processes *and* the data into the surrounding environment. Where as here, the data becomes non existent after the open, and the agents remain in the surrounding ambient with a potential role change.

- **Access primitives**
  The access primitives are there to support the actions that an agent wants to execute. There are two primitive operations provided to access variables (in general Terms), $get(Term, V)$ $put(V, Term)$. This can be used to access and update variables in the various levels of the ambient hierarchy. This is achieved through typing the primitives with the ambient identifier Id_δ. For example, $get(Term, V) : \delta_1$ will get the value of the term from the ambient instance δ_1 and will make V bound to it. The variables are generalized to a *prolog term* to include the possibility of containing complex expressions. Ambient LCC is a strongly typed language,

and as far as possible typed variables are used. τ can be an ambient type, an ambient instance id, or any other previously defined type at specification time.

The *Timeout* is used to introduce a minimal time management into the language. Interaction models normally constrain many agent actions to happen within a time interval. That is to say, agents can no longer perform a certain operation once a time interval is elapsed.

3.6 A few methodological issues

So far what we have discussed is the syntax of Ambient LCC. There are a few concepts, which are not specifically part of the language syntax, but nevertheless methodologically important as they provide additional structure and ease of understanding.

- **Ambient execution environment**
  An ambient Δ consists of two logical components, the ambient specification τ_Δ which we have previously dealt with, and the ambient instance, also called the ambient execution environment δ. Though Id_δ appeared many times in the above discussion, there was no mention of the execution environment as such denoted by δ. The following describes the notion of δ and elaborates on what constitutes its structure. The ambient execution environment is a runtime environment of the ambient Δ of type τ_Δ. For any ambient Δ of type τ_Δ, there can be an infinite enumeration of its runtime counterpart. While the ambient specification specifies a blueprint of the ambient, including the structure, norms for access, and other necessary information related to the type of the ambient, the execution environment deals with the execution variables, the current state the ambient is in and so on. It is defined in such a way that the execution variables comply with the specification requirements of τ_Δ. δ is not part of the formal syntax of the ambient LCC, and this omission is intended so as not to have to repeat the execution parameters in the protocol specification. Nevertheless, the variables have reserved names and can be accessed by these names scoped by the ambient identifier. As with τ_Δ definition, the δ definition varies with the interaction model under consideration.

  The recursive definition of ambients is also valid at the ambient instance level. That is, the recursive definition specifies the ambient hierarchy as explained previously. This is a τ_Δ specification at the highest level, and hence taken as the blueprint while creating the execution ambients. The structure of δ is created at runtime based on the *in* and *out* operations. Thus these operations should respect the ambient hierarchy defined at the type level. Thus the following statements holds for nested ambients at the instance level.

  - To move into a sub ambient of an enclosing ambient, it is required to be in the enclosing ambient. That is, given the spec $\langle Id_{\Delta_1}, \tau_{\Delta_1}\rangle(\langle Id_{\Delta_2}, \tau_{\Delta_2}\rangle)$ where Δ_1 encloses Δ_2 and the operations $new(Id_{\Delta_1}, Id_{\delta_1})$ and $new(Id_{\Delta_2}, Id_{\delta_2})$, for an agent $a \in \mathcal{A}$, with role $r \in \mathcal{R}$, *in* $Id_{\delta_2}(r)$ is valid only if $a \in \delta_1$ due to a prior operation, *in* $Id_{\delta_1}(r)$ that moved a into the enclosing ambient.
  - After *open* Id_{δ_2} is executed, a is assumed to be in the surrounding ambient, δ_1. *open* moreover destroys the ambient Id_{δ_2}.
  - Similarly, after *out* $Id_{\delta_2}(r)$ is executed, a is assumed to be in the surrounding ambient, δ_1

- **Domain Type**
  It is also useful to divide τ_Δ into two parts, that of τ_I and τ_D. While τ_I holds the interaction model specification variables, τ_D holds the domain variables which are specific to a domain. The separation between τ_I and τ_D is brought in to keep the domain independent specification from the domain dependent variables. τ_D is the place holder for all the variables that are part of the domain. By including τ_D in the specification, the execution environment variables are forced to adhere to the τ_D specification. This also provides the possibility of automatic verification.

4 Mapping Electronic Institutions into Ambient LCC

Ambient LCC can be used as a general distributed execution language. In the context of the Open-Knowledge project we will use it to implement electronic institutions in a distributed environment (i.e. without any central infrastructure) while preserving the semantics of scenes and performative structures. From an Electronic institution perspective, an ambient is an entity which can represent a performative structure, a scene or an agent. It has a nested structure as well, so we can have a performative structure ambient containing a collection of scene ambients, and a scene ambient containing a collection of agent ambients. The level of nesting corresponds to the nesting in the performative structure. The ambient processes can be *agent processes* that run in an agent ambient, scene ambient or a performative structure ambient, each of which are expanded below.

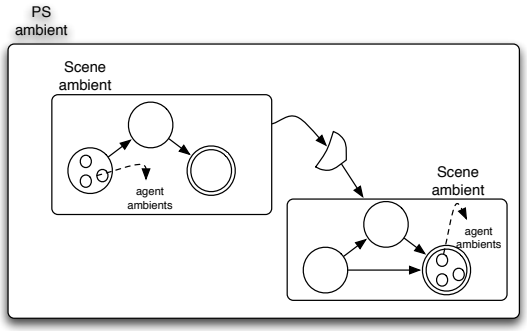

Figure 8: The ambients mapping an EI

4.1 Structural mapping

The mapping between EIs and Ambient LCC is based on the following structural translations:

- **Performative structure ambients**

  An institution's top performative structure is mapped into the highest level ambient, that we call *rootambient*, which includes other performative structure ambients, scene ambients and

agent ambients. The nesting is achieved by the recursive definition of the ambients. As in the original electronic institution specification, the transitions which connect the scenes are the main components to achieve synchronization, and will be treated as special types of scene ambients. Thus, scene ambients together with transition ambients construct the performative structure ambient. Any agent process entering an institution, first enters the root ambient, becoming an agent ambient (i.e. an ambient with a process running the decision making processes and a set of parameters to maintain the private state of the agent), and until it exits the institution resides in the root ambient (or in any ambient in the nested structure). In the process of enacting different scenes and transitions, the agent ambient enters and exits various sub ambients, at various layers of the nested structure. This is in sync with a performative structure definition which contains any number of nested performative structures and scenes.

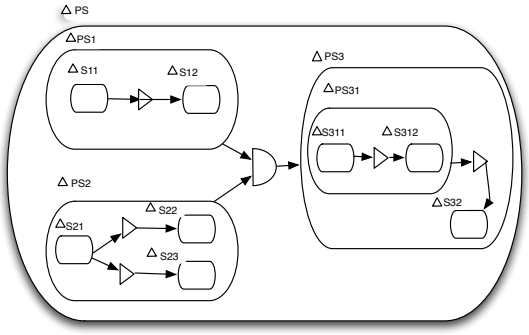

Figure 9: Nested ambients implementing an electronic institution

- **Scene ambients** Scenes are mapped into ambients. As for any ambient in Ambient LCC, a scene ambient is composed of a scene ambient specification and an execution environment. The specification of a scene in an EI is mapped into a number of parameters, the ambient type, that keep as values the different components of a scene specification (e.g. the states of the state machine, the arcs, the allowed illocutions, etc.) The execution environment parameters keep the context of the scene in execution and provides an environment for enacting the scene ambient specification. By this we mean, the scene ambient execution environment contains, for instance, the names of the participating agent processes at any given point in time, their roles, and the state of the computation at that instant of time. The participating agents are aware of the ambients they are in, and can communicate among themselves by referring to their identification within the ambient. Scene ambients are the most dynamic in the sense that most of the computation occurs within a scene ambient. The capabilities of movement defined in the language is expressive enough to model the movement of agents within an EI scene.
- **Transition Ambients** Transitions are considered as special kinds of scenes in electronic institutions. Owing to their similarity with the scenes and following similar conventions, it is fair enough to treat them similarly in ambient LCC too. Thus, transitions are mapped into ambients and derive all the properties of ambients as defined in the language. Transitions

in ambient LCC facilitate the initiation of scene ambients. They also execute the agent movements into succeeding scenes. Apart from these, transitions provide the settings for a scene ambient to start the execution, or for the agents to start their protocols.

- **Agent Ambients** We map agents into agent ambients. As mentioned before, an agent ambient contains a process (for internal thinking) and the parameters that represent the internal state of the agent. We don't map the functionality of governors. A governor provides an abstraction that facilitates the interaction between external agents and an institution. They ensure that the institution norms are followed as all agent messages go through the governors. Agent ambients provide the agent specific context of computation, but do not quite ensure that agents follow the norms of the institution.

4.2 Parameter mappings

4.2.1 Ambient Type

We map the specification of every single component of an institution into a number of parameters (accessible variables within an ambient) of the ambient that implements the component. Thus, for instance, if we recall from Section 2.1 the definition of a scene as $s = \langle R, DF_S, W, w_0, W_f, (WA_r)_{r\in R}, (WE_r)_{r\in R}, \Theta, \lambda, min, Max\rangle$ contains each component of a scene as a parameter. Any process within an ambient can then know the states of the dialogues by simply accessing the parameter W, or know what illocutions are legal from a given state by using the function λ.

4.2.2 Execution environment

The ambient execution environment δ when mapping EIs has the following structure.

$\langle \mathcal{A}, \rho, \sigma, w\rangle$ where

- $\mathcal{A}$ is the set of participating agents in the ambient δ
- $\rho : \mathcal{A} \rightarrow \mathcal{R}$ maps the agents to their respective roles $\mathcal{R}$ in the ambient
- σ contains all the illocution variable bindings at each moment during the ambient execution
- w is the current state of the ambient (in the context of electronic institutions, w is the current state of the scene)

4.3 Synchronization

This section outlines the synchronization issues that need to be solved when a centralized interaction model is specified in a distributed protocol language, and then maps the general issues to implementation decisions when *ambient LCC* is used as the protocol specification language. Electronic Institutions are assumed to be the underlying interaction model whenever scenes, transitions and illocutions are mentioned in the following discussion.

4.3.1 Requirements

The following are the general requirements for an appropriate electronic institution activities synchronisation:

- Event ordering

  In a distributed interaction model, a multi party conversation protocol is to be viewed from the perspectives of each of the participating agents. Here the holistic view can only be obtained from the interaction of partial views. Given this context, synchronization is required for the partial views to be consistent with the semantics of the original multi-party conversation. Thus, event ordering refers to the need for a correct ordering of scenes, transitions, and illocutions while maintaining the level of concurrency specified in the centralized model. Within a scene, for example, without *explicit* mechanisms for synchronization, the agents will not know when to enact their part in the scene, that is, when to speak. This can lead to potential conflicts such as more than one agent trying to move into different scene states by uttering different illocutions at the same time, which would violate the fundamental integrity constraint of the scene, that of all the agents in a scene are at the same conversation state at any given point of time. Higher level entities such as Performative Structures or transitions, can be synchronized with just some minor additions to the basic mechanism of scene synchronization.

- State consistency

  All participating agents *are* at the same state of execution of an EI at any given point in time. That is, the state of a conversation is in one and only one of the possible states of the finate state machine of the scene. And we need to preserve this. State consistency can be total or partial. For example, in a scene, in a *totally* consistent view, all the participating agents must know what the conversation state is at all times. In a *partially* consistent view, the idea is that as long as an agent has nothing to say it does not need to get synchronized. As soon as it has something to say, it needs to know the current state of the conversation. The concept of state consistency within a scene has different counterparts in other components.

- Concurrency

  We aim at achieving a similar level of parallelism of the centralized EI model. If scenes can go on in parallel in the performative structure, then the same should be valid in the distributed model as well. That is, the synchronization techniques to be introduced should stay at the optimum level so as to guarantee the correct event ordering, and maintain state consistency, but should not become overly restrictive by removing the parallelism present in the centralized model.

From these general requirements we can illustrate the variety of synchronisation problems that electronic institutions present with a list of examples. Later we can see the proposed generic solution and how it is operationalised as algorithms that are generated and embedded into the code generated by the translation algorithm.

4.3.2 Example Scenario

- **Scene** As outlined above, the event ordering synchronization requirement in a scene arises when more than one participating agent can utter illocutions and take the scene to potentially different states. The following protocol segment illustrates this possibility. In the figure, at $state0$, agent $?a$ uttering illocution $?L$ would take the scene to $state2$ whilst agent $?b$ uttering illocution $?M$ would take the scene to $state1$.

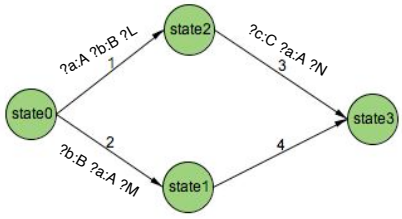

Figure 10: A scene scenario

The requirement for state consistency can be made clear with an example: if agent $?a$ has managed to utter $?L$ moving the scene state to $state2$ and agent $?b$ is not aware of this change, then it might still try to make the state transition from $state0$ to $state1$ by uttering $?M$ which is an invalid illocution from state $state2$. Similarly, if an agent $?c$ wants to say $?N$ from $state2$, where c is neither the sending agent nor the receiving agent in the previous illocution before arriving at $state2$, then agent $?c$ has no way of knowing what state the scene is in, and whether can it say its illocution $?N$ or not. In other words, $?c$ has to be informed about the evolution of the state of the scene.

- **Transitions** Transitions act as the gateways to scenes. So the synchronization aspects related to the entry into the scene are naturally managed at the transition nodes. There are a number of synchronisation issues here, for instance the transition types, that permit an agent to join one or more than one target scene, or the participating agent may need to wait for other agents to follow one or more edges leading to scenes. The scene specification in the corresponding ambient τ_Δ will contain the necessary information on the eligibility conditions for any role to move into a scene. Thus, at any given point, the synchronization can be ensured by looking at the parameters of the transition ambient and checking for correctness. The agents will have to wait in the ambient until the specification constraints to move are satisfied, in which moment the operations to move into the target scene ambient can be granted.

4.3.3 Solution Concepts

In the cases described above, the problems arise due to lack of communication and coordination. So the solution we propose is to introduces both these elements into the ambient LCC code im-

plementing the interaction models. The ambient will act as a host for particular *request*, and *acknowledgment* protocols for communication.

The synchronization for a scene starts from the preceding transition. That is, as soon as an agent a arrives at the preceding transition of a scene S, it sends out a multicast message to all agents in the performative structure ambient, requesting for the entry into the scene that it wants to enter. If a receives an accept message from a $b \in Id_s$ (where Id_s is the ambient execution environment for scene S. Also S and s is differentiated here to refer to ambient type and ambient instance respectively. That is Id_S and τ_S will refer to scene ambient Id, and scene ambient type, where as Id_s will refer to the Id of the corresponding ambient instance), then it executes the *in* operation and enters the scene, and awaits for an opportunity to speak. If a receives a wait request from b, then it waits for the accept message to join the scene ambient if a does not receive any acknowledgment, then that means that there is no other agent in the scene ambient, and it has to start a new execution of the scene. So a creates a new instance Id_s of an ambient type τ_S with initial parameter values and move into the ambient. Creation of a new scene has to be always in agreement with the performative structure specification τ_{PS}

Agent as a scene manager

As long as an agent holds the ambient, it is also designated to act as the scene manager. The scene manager a does the following tasks.

- If the scene is at the initial state w_0, a checks whether the required number of agents are present
  - if it is the case, it allows message processing and the scene execution is actually started
  - if not, waits for entry requests from other agents
- a looks at its cache for the request messages received from other agents wanting to join Id_s
  - When a finds a request for entry from b, a first checks whether agent b of role r is already in the ambient
  - if yes, a updates the cache by removing this request and goes on to process the subsequent requests.
  - if not, a looks at the scene specification τ_Δ to check whether current scene state w_i is one of the access states for b of role r.
  - If the check succeeds, then a checks for the cardinality constraints on the role r.
  - if succeeds, then a sends an accept message to b and wait for the acknowledgment from b. [A number of repetitions are done to make sure that b has a fair chance to join the ambient]
  - when the acknowledgment is received from b, a gives the necessary access permission to move into the ambient. a then updates the ambient.
- a looks at its cache for the request messages received from other agents wanting to exit Id_s
  - When a finds a request to exit from b, a first checks whether agent b of role r is in the ambient

- if no, a updates the cache by removing this request and goes on to process the subsequent requests.
- if yes, a looks at the scene specification τ_Δ to check whether current scene state w_i is one of the exit states for b of role r.
- if succeeds, then a sends an accept message to b and wait for the acknowledgment from b. [A number of repetitions are done to make sure that b has a fair chance to exit the ambient]
- when the acknowledgment is received from b, a gives the necessary permission to move out of the ambient. a then updates the ambient.

- a checks the current state and utters the illocution that is valid for the current state. It then updates the current state to w_j from w_i. a also updates the variable bindings and other changes provoked by the state transition.
- If there is no other pending tasks, then a checks requests for the value of the *current state* by other agents. Alternatively, a can also send the updated *current state* value to all the participating agents.
  - a sends the ambient Id_s to the agent $b \in A$ who is the first to request for the ambient
  - if no request is received, a waits in a loop.

Agent wanting to join a scene

An agent wanting to join a scene does the following

- send a multi cast entry request to all agents in a higher level ambient.
- wait for access permission from the scene manager
- if it receives access permission to join the scene
- then moves into the ambient
- else waits and starts a new scene instance

Agent wanting to utter an illocution

An agent in the ambient wanting to utter an illocution does the following

- send a request message for the *current state* of the scene to the scene manager
- if the current state is a valid state for uttering the illocution, then it sends another request to hold the ambient. (Alternatively the number of messages could be reduced by making the scene manager handle the state matching.)
- if the agent receives the ambient to hold, then it executes the scene manager procedure
- else waits in a loop (or thinks more)

4.3.4 Algorithm steps

The above algorithm schemes can be translated into more precise algorithm steps as described in algorithms 1–4. For illustrative purposes, only the scene manager part is shown here.

Algorithm 1 Main Procedure - Scene manager

- **while** $requiredNumAgts \notin Id_s$ **do**
  $entryRequestProc()$
  end while
  while $sendAmbient \neq null$ **do**
  $entryRequestProc()$
  $exitRequestProc()$
  $execIllocution()$
  $ambRequestProc()$
  end while

Algorithm 2 entryRequestProc()

if $requestedAgt \notin Id_s$ **then**
  if $w_i \in accessStates(R_{requestedAgt})$ **then**
  if $cardinality(R) \in Id_s \leq Max \in Id_S$ **then**
  send accept message
  else
  continue
  end if
  end if
  end if

Algorithm 3 exitRequestProc()

if $requestedAgt \in Id_s$ **then**
  if $w_i \in exitStates(R_{requestedAgt})$ **then**
  send accept message
  else
  continue
  end if
  end if

4.3.5 Example

Consider the scene s of Figure 4.3.5

The scene specification parameters $\tau_{I_s} = \langle R, DF_S, W, w_0, W_f, (WA_r)_{r\in R}, (WE_r)_{r\in R}, \Theta, \lambda, min, Max\rangle$ for the above scene s are:

R $=\{R1, R2\}$

Algorithm 4 ambRequestProc()

while *ambRquestQue* $\neq$ *null* **do**
  if *currentState* = *receivedState* **then**
    send ambient
    here the scene manager takes care of matching the states,and sending the ambient
    sendAmbient = 1
  end if
end while

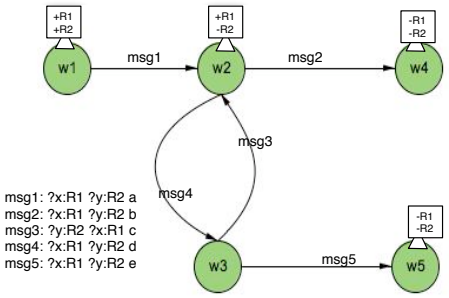

Figure 11: Scene example (note: messages don't follow the EI syntax).

$$
\begin{aligned}
DF_S &= \langle \{a,b,c,d,e\}, \_, \_, \_, \_, \_ \rangle \\
W &= \{w1, w2, w3, w4, w5\} \\
w_0 &= w1 \\
W_f &= \{w4, w5\} \\
(WA_r)_{r \in R} &= \{\{w1, w2\}_{R1}, \{w1\}_{R2}\} \\
(WE_r)_{r \in R} &= \{\{w4, w5\}_{R1}, \{w2, w4, w5\}_{R2}\} \\
\Theta &= \{(w1, w2), (w2, w3), (w3, w2), (w2, w4), (w3, w5)\} \\
\lambda &= \{(w1, w2) \rightarrow msg1, (w2, w3) \rightarrow msg4, (w3, w2) \rightarrow msg3, (w2, w4) \rightarrow msg2, \\
&\quad (w3, w5) \rightarrow msg5\} \\
min &= \{R1 \rightarrow 1, R2 \rightarrow 2\} \\
Max &= \{R1 \rightarrow 10, R2 \rightarrow 10\}
\end{aligned}
$$

- Agent a of role $R1$ arrives at the (only) transition preceding scene S.
- a sends a multicast message to everyone in the performative structure ambient (in the worst case to all agents in the network) to request an entry into Id_s
- a receives no reply
- a starts a new scene execution, by creating an ambient instance $Id_{s1}$and moving into it. The ambient instance has the entries: $\langle \{a\}, \{a \rightarrow R1\}, \emptyset, w1 \rangle$
- a checks whether there are enough participants to start the scene. τ_S specifies there should be one agent of R1 and two of R2

- waits for requests for entry to ambient Id_s
- receives a request for entry from b of role $R2$
- a checks whether b of $R2$ is in Id_s - no
- a checks whether b of $R2$ can enter the scene at $w1$ from the specification - yes
- a checks Id_s to see whether the number of agents of $R2$ is less than Max from τ_S -yes
- a sends accept request to b and waits for acknowledgment
- b sends the acknowledgment to a
- a sends the access permission message to b
- b move into the ambient and asks to hold the ambient
- a updates the ambient as: $\langle\{a, b\}, \{a \rightarrow R1, b \rightarrow T2\}, \emptyset, w1\rangle$
- a checks whether scene can be started - no
- a receives another entry request from c and repeats the procedure, updates the ambient as: $\langle\{a, b, c\}, \{a \rightarrow R1, b \rightarrow T2, c \rightarrow R2\}, \emptyset, w1\rangle$
- a checks whether the scene can be started - yes
- a now executes illocution msg1 at $w1$ and updates the ambient as: $\langle\{a, b, c\}, \{a \rightarrow R1, b \rightarrow T2, c \rightarrow R2\}, \{x = a, y = b\}, w2\rangle$
- a checks for request messages to hold the ambient
- In this case, a can execute the next illocution at state $w2$. Hence the previous request from b is ignored.
- Scene moves into next state $w2$ and proceeds similarly.

The following ambient LCC clauses translate the above synchronization algorithm.

$$
\begin{array}{lll}
a(x, \text{“}R1\text{”}, \text{“}Trans_S\text{”}) & :: & entryrequest \Rightarrow a(\_, \_, PS) \\
 & & then \;\; in \; Id_s(R1) \leftarrow accesspermission \Leftarrow a(y, \text{“}SceneMgr\text{”}, Id_s) \\
 & or & \\
 & & new(\text{“}Id_S1\text{”}, Id_s) \;\; in \; Id_s(R1) \\
a(x, \text{“}R1\text{”}, \text{“}Id_s\text{”}) & :: & accessPerm \Rightarrow a(y, \text{“}R2\text{”}, \text{“}Trans_S\text{”}) \leftarrow accessConditions \wedge \\
 & & \qquad\qquad entryRequest \Leftarrow a(y, \text{“}R2\text{”}, \text{“}Trans_S\text{”}) \\
 & or & denyPerm \Rightarrow a(y, \text{“}R2\text{”}, \text{“}Trans_S\text{”}) \leftarrow denyConditions \\
 & or & \\
 & & exitPerm \Rightarrow a(y, R2, Id_s) \leftarrow exitConditions \wedge eexitRequest \Leftarrow a(y, R2, Id_s) \\
 & or & denyPerm \Rightarrow a(y, R2, Id_s) \leftarrow denyConditions \\
 & or & \\
 & & msg1 \Rightarrow a(y, \text{“}R2\text{”}, \text{“}Id_s\text{”}) \\
 & or & \\
 & & \text{“}Id_s\text{”} \Rightarrow a(y, \text{“}R2\text{”}, \text{“}Id_s\text{”}) \leftarrow sendAmbientConditions
\end{array}
$$

4.4 Illocutions

An *illocution schema* is part of the transition definition (see section 2.1). It defines the conditions necessary for an illocution to produce a state change, and the transition's resulting side actions.

Illocutions schemas are defined as $IS = \iota((ag_s, R_s), (ag_r, R_r), \varphi)$. Where ag_s and R_s are the sending agent identifier and role, ag_r, R_r are the receiving agent's identifier and role,ι is the illocutionary particle and φ is the propositional content being sent. Illocutions may contain variables which give the engineer greater specification power.

By *illocution* we refer to those illocution schemas whose variables have been grounded. To say it in another way, an illocution is an illocution schema whose definition is composed exclusively by constants. Grounding is the process through which an illocution schema becomes an illocution by substituting the variables by their actual values.

Variables can be used for illocution schema specification. They can appear as bound or free. A free variable v_f is grounded by matching it with any value of its type. Then, the value 'c_{n+1}' used to match v_f is added as a new binding $b_i = (v_f, c_{n+1})$ to σ in the execution environment. A variable is specified as free when prefixed with '?'. On the other hand, when a schema with a variable v_b, specified as bound, is matched against an actual illocution, it can only be substituted by the value 'c_n'. Where 'c_n' is v_b's last binding $b_l = (v_b, c_n) \in \sigma$. A variable is specified as bound when prefixed with '!'. In other words, if the value being matched against '!v_b' is different from 'c_n', the illocution uttered is unacceptable.

Most parts of an illocution schema can be specified with variables: agent ids, agent roles, and propositional content. Constraints, and actions labeling arcs can also use variables although only as bound.

4.5 The Algorithm

This section describes the algorithm that converts an Electronic Institution scene into an Ambient LCC specification.

The translation generates one Ambient LCC clause for each role allowed in a scene. Each clause deals with the projection of the scene[5] to the given role. Inside the clause the LCC code refers to what an agent of the given role can do. The code is divided by states, each state in a subpart of the clause. Each subpart has the same pattern; it first checks that the state in the ambient is the correct one, then each transition out of the state is translated. If the transition's end state is a final state the scene is destroyed, otherwise the clause is called again.

The algorithm has been divided into three main parts for legibility purposes. EI2LCC (see Algorithm 5) finds the set of agent roles and states. For each pair of agent role and state, it calls

[5]For each agent variable and role in a scene specification, we can obtain its scene projection. $S_{r_1} = (R, DF, w_0, W_f, (WA_r)_{r \in R}, (WE_r)_{r \in R}, \theta_{r_1}, \lambda, min, Max)$ is the projection of scene $S = (R, DF, w_0, W_f, (WA_r)_{r \in R}, (WE_r)_{r \in R}, \theta, \lambda, min, Max)$ for agent role r_1. Where $\theta_{r_1} = \{(w_i, w_j) \in W \times W | \lambda(w_i, w_j) = \iota((\_, R_s), (\_, R_r), \_)$ *and* $((R_s = r_1)$ *or* $(R_r = r_1))\}$. The intuition behind a scene projection is that when you project a scene through a role, you end up with a 'sub-scene' where the agent role is present in all the illocution schemes. The projected scene may be an unconnected graph.

extractLCC (see Algorithm 6) which returns the Ambient LCC corresponding to that state and the transitions that an agent of the given role can take. It then combines all the so generated ambient LCC code through the OR connective and creates one clause for each agent role.

ExtractLCC is in charge of translating one scene state and its transitions for a given role into Ambient LCC. The output contains the following three parts, first the code that checks that the ambient is in the given state, second the code for the illocution schema translations, third the code for the link to the next state. ExtractLCC delegates the illocution schema LCC extraction to extractLCCFromIllocution (see Algorithm 7).

The transformation algorithm works on the following assumptions:

- Each transition must be labeled with only one illocution.
- Each agent variable must only be used for agents of one and only one role.

Previous to executing the transformation algorithm on a scene, its specification needs to be transformed to a scene that has the previous properties. The first property is attained by getting the transitions with more than one illocution schema and generate one transition for each illocution schema. All of the generated transitions will have the same start and end state, and each will have one of the illocution schemas of the original transition. The second property is attained by renaming those agent variables that are used for agents of more than one role. The renaming is done by appending the role ID to the variable name, generating as many different agent variables as agent roles it represented.

4.5.1 Main extraction

This subsection deals with the algorithm that extracts Ambient LCC from the scene for one role and state (see Algorithm 6). The algorithm is called for each state and role in the scene. And for each combination it generates a piece of Ambient LCC code describing how an agent of the given role can move through the protocol from the given state to the other accessible states (which will be Ambient LCC code returned from other calls to the function).

In order to define a message passing protocol, Ambient LCC has three operators; “then” , “par”, and “or”. The operators define sequence, parallelism, and choice respectively. Since EI scenes are played sequentially (only one agent speaks at a time), the transformation will not generate the “par” operator. The use of the other two operators is explained below.

The “or” operator is used in two places: 1) To separate code segments representing each scene state. Since a scene is sequential, it can only be in one state at a time. Therefore, the agent will go to the clause section for the actual state. Code segments representing each scene state are separated with the “or” operator. 2) To separate Ambient LCC code segments of each transition leaving from the same state. Only one transition can take place at a time, therefore the code segments representing the different transitions leaving from a state are separated with the “or” operator. In both of these cases the Ambient LCC code represents parts of the scene that are mutually exclusive, if one happens the others cannot.

The “then” connective orders events, and is also used in two places: 1) To indicate that the transitions to another state happen after the agent checks in which state the scene is in. 2) To state

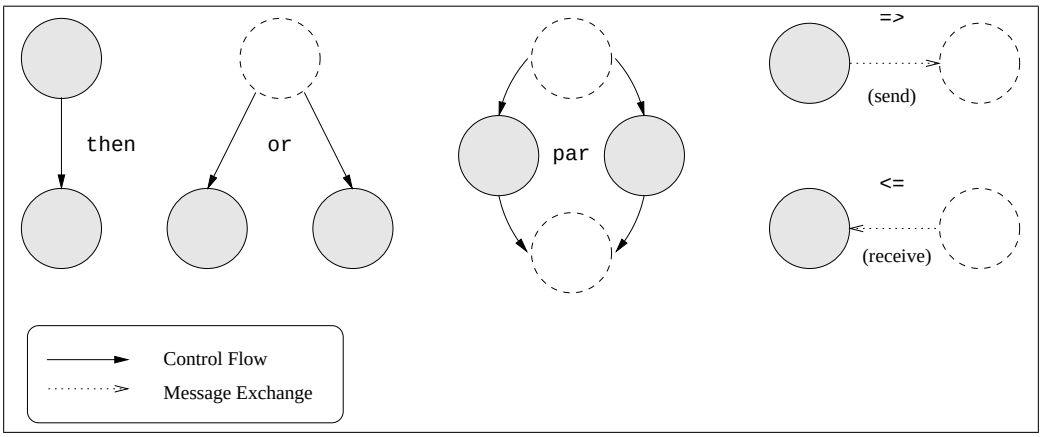

Figure 12: Ambient LCC operators

that after a transition to another state is executed, the agent may start over or leave the scene (if the scene protocol allows so).

4.5.2 Illocution schema translation

This part of the conversion is handled by algorithm 7. This algorithm will have as output different pieces of Ambient LCC code depending on whether the agent of the given role is the message sender, receiver, or neither.

The simple cases are when the agent is the receiver or neither the receiver or sender. In the first case the illocution is transformed to a simple message exchange: "$msg \Leftarrow x$". In the second case no Ambient LCC is returned because the agent does not intervene in the actual state.

The case when the agent is the sender must deal with updating the ambient, and checking that the constraints are satisfied from the ambient values. Constraint checking is done by reading the values needed from the ambient and then doing the needed operations. Some of the values that will always need to be read are the bound variables (see Section 4.5.3). The values to be updated by the actions associated to the arc of the illocution are written into the ambient. The values that will always be updated are the state (which changes after any transition, except reflexive ones), and the free variables specified in the illocution schema in the message (these will become bounded to a new value that must be saved).

4.5.3 Variables

An ambient can contain data. Ambient LCC has a way of accessing this data and modifying it. The properties of a scene, and all the variables used in the illocution schemas are mapped onto the ambient's data. When an illocution schema is translated, the variable and property values are extracted from the ambient, then the message is constructed with these values, and after the

message is sent those variables that have been modified are updated to the ambient. The Ambient LCC construct is the following:

$actions \leftarrow message \leftarrow constraints$

The constraints contain the operations to extract data from the ambient, and other arithmetic and logic operations. The actions contain operations to update ambient values, in addition to those operations that can be used in the constraints.

Algorithm 5 EI2LCC($scene$)

```
 1: lcc ← NIL
 2: for role ∈ getRoles(scene) do
 3:     clauseName ← getRoleClauseName(role, scene)
 4:     for w ∈ scene.W do
 5:         lcc_⟨role,state⟩ ← extractLcc(role, scene, w, clauseName)
 6:         lcc_role ← or(lcc_role, lcc_⟨role,state⟩)
 7:     end for
 8:     lcc_role ← newRoleClause(clauseName, lcc_role)
 9:     lcc ← lcc + lcc_role
10: end for
11: return lcc
```

Algorithm 6 extractLcc($role, scene, w_i, clauseName$)

```
    lcc_wi ← getFromAmbientLcc('w', w_i)
    for t ∈ getTransitionsFrom(w_i) do
 3:     i ← getIllocution(t)
        w_next ← getToState(t)
        lcc_i ← extractLccFromIllocution(i, w_next, role)
 6:     if w_next ∉ scene.W_final then
            lcc_i ← then(lcc_i, clauseName)
        else
 9:         openClause ← openAmbientLcc(scene)
            lcc_i ← then(lcc_i, openClause)
        end if
12:     lcc_transitions ← or(lcc_transitions, lcc_i)
    end for
    if role ∈ w_i.out then
15:     outClause ← leaveAmbientLcc(scene)
        lcc_transitions ← or(lcc_transitions, outClause)
    end if
18: lcc_wi ← then(lcc_wi, lcc_transitions)
    return lcc_wi
```

4.5.4 Entering and Leaving a Scene

Electronic Institutions define how agents can enter and leave a scene. A scene state can be marked as input or output for each role. When an agent enters a scene its id and role are added to the environment. Likewise its id and role are removed when it leaves.

There are some operations that are specific for the ambient part of Ambient LCC. These are the operations that create and destroy ambients, and the ones for getting in and out of them. The code in Ambient LCC of a scene translation only deals with what goes inside a scene. Therefore, the operations to create a scene and move into a scene are not used. Only the operations referring to the destruction (opening) of a scene and leaving a scene happen when inside a scene. The "create" and "in" operations may happen at transition scenes. The scene states in Electronic Institutions

Algorithm 7 EXTRACTLCCFROMILLOCUTION($ill, w_{next}, role$)

```
    lcc_send ← NIL
 2: if ill.senderRole = role then
        lcc_send ← ADDTOAMBIENTLCC(var, w_next)
 4:     for var ∈ ill ∧ ISFREEINILLOCUTION(var, ill) do
            lcc_send ← lcc_send + ADDTOAMBIENTLCC(var)
 6:     end for
        lcc_send ← lcc_send + ACTIONSTOLCC(ill.actions)
 8:     lcc_send ← lcc_send + MESSAGETOLCC(ill.message)
        lcc_send ← lcc_send + CONSTRAINTSTOLCC(ill.constraints)
10:     for var ∈ ill ∧ ISBOUNDINILLOCUTION(var, ill) do
            lcc_send ← lcc_send + CHECKBINDINGINAMBIENTLCC(var)
12:     end for
    end if
14: lcc_rec ← NIL
    lcc ← lcc_send
16: if ill.receiverRole = role then
        lcc_rec ← MESSAGETOLCC(ill.message)
18:     lcc ← OR(lcc, lcc_rec)
    end if
20: return lcc
```

can be defined as input or output for each role in the scene. When a state is reached that is labeled as output for R_i, all agents of role R_i can choose to leave a scene. In Ambient LCC this is translated as adding a choice to execute the "out" ambient operation when a state is reached that allows this. On the other hand, when a final state is reached, the scene must finish. Therefore the Ambient LCC translation will have an "open" ambient operation after a transition that takes to a final state.

4.5.5 Example

This section shows a scene specified using a state machine and its ambient LCC translation. The scene can be seen in Figure 4.3.5. State S1 is the initial state of the scene. The structure of the illocution template is the same as the one already introduced.

The scene in Figure 4.3.5 has the following properties: 1) Variables are used both as free and bounded. 2) There are input and output states. 3) It has two agent roles. 4) And it has choice points. It is a simple scene which has all the main problems to be found when translating to LCC. The resulting LCC when applying the transformation algorithm will be:

$a(x,$"$R1$", "$Scene1$") ::
$(get($"w", "$S1$") $then$
$put($"w", "$S2$") $\wedge$ $put($"y", $y) \wedge put($"x", $x) \leftarrow a \Rightarrow y \leftarrow y \in l \wedge get($"$allR2$", $l)$ $then$
$a(x,$ "$R1$", "$Scene1$"))
or
$(get($"w", "$S2$") $then$
$(a \Rightarrow y$ $then$ $open(sceneId) \leftarrow get($"$Id_{scene1}$", $sceneId)$ or
$c \Leftarrow y$ $then$ $a(x,$ "$R1$", "$Scene1$")))
or
$(get($"w", "$S3$") $then$
$(e \Rightarrow y \leftarrow get($"$y$", $y)$ $then$ $open(sceneId) \leftarrow get($"$Id_{scene1}$", $sceneId)$ or
$put($"w", "$S2$") $\leftarrow d \Rightarrow y \leftarrow get($"$y$", $y)$ $then$ $a(x,$ "$R1$", "$Scene1$")))

$a(y, \text{“}R2\text{”}, \text{“}Scene1\text{”}) ::$
$(get(\text{“}w\text{”}, \text{“}S1\text{”})\ then$
$a \Leftarrow x\ then\ a(y, \text{“}R2\text{”}, \text{“}Scene1\text{”}))$
or
$(get(\text{“}w\text{”}, \text{“}S2\text{”})\ then$
$(put(\text{“}w\text{”}, \text{“}S3\text{”}) \wedge put(\text{“}y\text{”}, y) \leftarrow c \Rightarrow x \leftarrow get(\text{“}x\text{”}, x)\ then\ a(y, \text{“}R2\text{”}, \text{“}Scene1\text{”})\ or$
$b \Leftarrow x\ or\ out(sceneId) \leftarrow get(\text{“}Id_{scene1}\text{”}, sceneId)))$
or
$(get(\text{“}w\text{”}, \text{“}S3\text{”})\ then$
$(e \Leftarrow x\ or$
$d \Leftarrow x\ then\ a(y, \text{“}R2\text{”}, \text{“}Scene1\text{”})))$

The example EI scene contains two roles, therefore the resulting Ambient LCC contains two clauses. Inside each clause we can appreciate a set of code chunks joined through the “or” connective. These chunks refer to one state in each scene. Notice that states ‘S4’ and ‘S5’ do not have an associated code chunk, this is because they are final states that have no transitions leaving from them. In final states the ambient is opened, and this happens at the end of the transition that ends in one of the final states.

Each of the code chunks has the same basic pattern (see Section 4.5). Lets focus on state ‘S2’ for role ‘R2’. The first part of the code checks that the actual state is ‘S2’. The next part (after the “then” connective) deals with the two transitions and the possibility of leaving the scene, which it separates with the “or” connective. In the transition from ‘S2’ to ‘S3’ the agent of role R2 sends the message, therefore it is in charge of checking the constraints and updating the ambient. The first part of the transition code are the message postconditions which must update the state value and add a new value to variable ‘y’ (it was unbounded in the scene specification). The next part shows the message to be sent to agent ‘x’. The last part gets the value of the variable ‘x’ from the ambient, the variable represents the agent to send the message to (the variable was used as bound). Finally, once the illocution is translated, the call to the initial clause is added so the next state code can be reached.

The transition from ‘S2’ to ‘S4’ is much simpler. Since the agent of role ‘R2’ is receiving the message it does not have to update the ambient or check the constraints. Also, since the end state is a final state, there is no call to the initial clause because the protocol has ended. The other LCC to take into account is the “out” ambient operation, since state ‘S2’ is an input and output state for role ‘R2’ the agent can choose to leave the scene at this point. There is no call to the initial clause here either since the protocol for this scene has ended the moment the agent leaves the ambient.

The connection between this generation algorithm and the synchronisation algorithms presented earlier is straightforward.

5 An Example of an EI expressed as an interaction model in *ambient LCC*

The interaction models considered here are those associated with an electronic institution. However it should be noted that the difference when translating from other interaction models is only in terms of the τ (the type definition of the ambient), which is normally specified in the model. The example is taken from an auction context and defines an *AuctionHouse* performative structure. The performative structure starts with a *Root* scene through which the agents enter the institution and then proceeds to execute various scenes. The external roles of this performative structure are the *Buyer*, *Seller* and the *Guest*. The internal roles are the *Staff* and the *Auctioneer*. A detailed explanation of the electronic institution entities used in the example is beyond the scope of this paper, an interested reader is referred to relevant literature on the topic. The translation of this instituti on into ambient LCC is only partial by taking sample scenes and transitions and generating the corresponding mapping into the language. Nevertheless we have tried to cover all aspects translation.

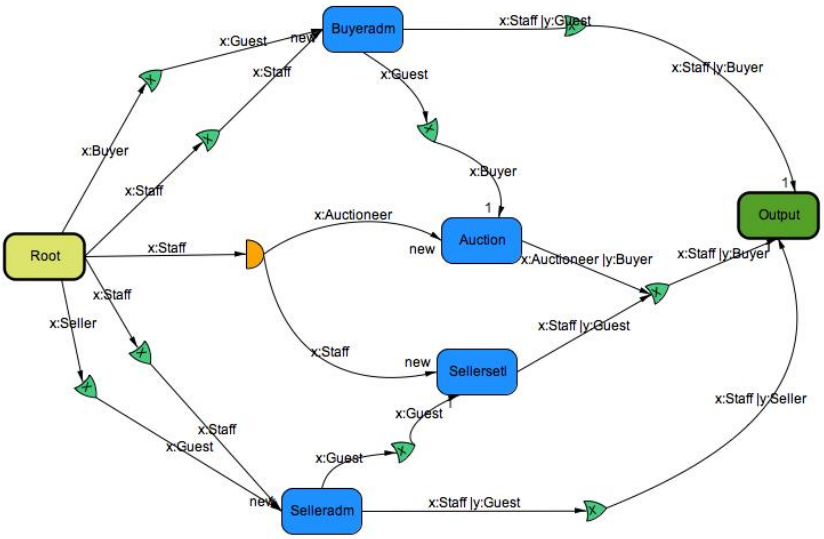

Figure 13: Upward bidding auction

5.1 A Performative Structure

- **Auction house ambient**
  The *Auction House* when treated in the ambient LCC, can be seen as an ambient consisting of scene and transition sub ambients. The ambient hierarchy for the Auction House ambient (Δ_{AH}) contains multiple layers. The top layer contains the Auction House ambient and the next layer contains the scene and the transition sub ambients. The leaf layers are the agent sub ambients which are implicitly defined immediately when agents enters into the top layer, (Δ_{AH}). Following the ambient LCC syntax, (Δ_{AH}) can be defined as below

  $\langle Id_{AH}, \tau_{AH} \rangle (Id_R, Id_{BA}, Id_A, Id_{SA}, Id_{SS}, Id_O)$

  Here τ_{AH} defines the *type* of the (Δ_{AH}) ambient. To give a flavor of what the *type* means in an auction context, we can associate *types* to different kinds of auctions such as *english*, *upward bidding*, *downward bidding*, and *dutch* which have their own specific rules of acces, behavior and hence have different *type* definitions. The performative structure specification parameters $\tau_{AH} = \langle S, T, s_0, s_\Omega, E, f_L, f_T, f_E^O, C, ML, \mu \rangle$ for the above performative structure AH are:

$$
\begin{array}{ll}
S & = \{Id_R, Id_{BA}, Id_A, Id_{SA}, Id_{SS}, Id_O\} \\
T & = \{TrXor_{BuyerAdm1}, TrXor_{BuyerAdm2}, , TrAnd_{Auction}, \\
 & \quad TrXor_{Auction}, TrXor_{SellerAdm1}, TrXor_{SellerAdm2}, TrXor_{SellerSetl}, \\
 & \quad TrAnd_{SellerSetl}, TrXor_{Output1}, TrXor_{Output2}, TrXor_{Output3}\} \\
s_0 & = Id_R \\
s_\Omega & = Id_O \\
E & = \{\langle (Id_R, TrXor_{BuyerAdm1}), (Id_R, TrXor_{BuyerAdm2}), (Id_R, TrAnd_{Auction}), \\
 & \quad (Id_R, TrXor_{SellerAdm1}), (Id_R, TrXor_{SellerAdm2}), (Id_{BA}, TrXor_{Auction}), \\
 & \quad (Id_{BA}, TrXor_{Output1}), (Id_A, TrXor_{Output2}), (Id_{SA}, TrXor_{SellerSetl}), \\
 & \quad (Id_{SS}, TrXor_{Output3}) \rangle, \langle (TrXor_{BuyerAdm1}, Id_{BA}), (TrXor_{BuyerAdm2}, Id_{BA}), \\
 & \quad (TrAnd_{Auction}, Id_A), (TrAnd_{Auction}, Id_{SS}), (TrXor_{Auction}, Id_A), \\
 & \quad (TrXor_{SellerAdm1}, Id_{SA}), (TrXor_{SellerAdm2}, Id_{SA}), (TrXOr_{SellerSetl}, Id_{SS}), \\
 & \quad (TrXor_{Output1}, Id_O), (TrXor_{Output2}, Id_O), (TrXor_{Output3}, Id_O) \rangle\} \\
f_T & = \{TrXor_{BuyerAdm1} \rightarrow Xor, TrXor_{BuyerAdm2} \rightarrow Xor, TrAnd_{Auction} \rightarrow And, \\
 & \quad TrXor_{Auction} \rightarrow Xor, TrXor_{SA1} \rightarrow Xor, TrXor_{SellerAdm2} \rightarrow Xor, \\
 & \quad TrXor_{SellerSetl} \rightarrow Xor, TrAnd_{SellerSetl} \rightarrow And, TrXor_{Output1} \rightarrow Xor, \\
 & \quad TrXor_{Output2} \rightarrow Xor, TrXor_{Output3} \rightarrow Xor\}
\end{array}
$$

  where $Id_R, Id_{BA}, Id_A, Id_{SA}, Id_{SS}, Id_O$ are the ambient Ids of the scenes *Root*, *Buyeradm*, *Auction*, *Selleradm*, *Sellersetl*, and *Output* respectively.
  Following the performative structure definition τ_{AH} has the following expansion:

  What we have specified above is a few rules that the performative structure ambients of type τ_{AH} should obey, while creating and executing new instants of this type. Some of them are the set of scene ambients that can be and should be part of the performative structure ambient (Δ_{AH}), the initial scene Id_R and the final scene Id_O. f_L, f_E^O, C, μ, ML are intentionally left unspecified as it is not relevant here. Also this could be extracted from the performative

structure diagram.

- **Initiating the AH ambient**
  In the language, it is only through the agents that a new ambient instance is created. As performative structure at the root level is the first entry into an electronic institution, an agent outside of the performative structure does not play any role, and hence cannot be defined by the language. Here we are faced with a dilemna so as how to create a performative structure ambient and initiate it. Yet, an exception could be taken in the case of the performative structure at the root level. It can be defined as follows. As the *Root* scene acts as an entry point to the performative structure, also as it has no active role, all the initialization can be performed in that scene. Given below is the initialization and agents entry into the performative structure and the root ambient. As staff agents are internal agents they are the right candidates for doing this initialization.

  $a(X, \mathit{Staff}, Id_{AH})$ :: $new(Id_{AH}, Id_{ah})$
  $then$
  $in\ Id_{ah}(\mathit{Staff})$
  $then$
  $new(Id_{Root}, Id_r)$
  $then$
  $in\ Id_r(\mathit{Staff})$
  $then$
  $a(X, \mathit{Staff}, \mathit{Root})$

  Here $new(Id_{AH}, Id_{ah})$ creates a new instance Id_{ah} of the ambient Id_{AH}. Then the *Staff* agent moves $in\ Id_{ah}$ and then creates a new instance Id_r of the *Root* scene ambient. Then the *Staff* agent moves $in\ Id_r$ and enacts the role of a *Staff* agent.

5.2 A Transition

Two transitions are picked from the example in figure 10, section 5. They are of type AND and the XOR. AND transitions require that the agents at the transition follow the succeeding arcs in parallel. XOR require that agents at the transition follow one and only one of the succeeding arcs. $TrAnd_{Auction}$ is the transition used by the *Staff* agents prior to entering the *Auction* scene. $TrXor_{Auction}$ is the transition used by the *Buyer* agents prior to entering the *Auction* scene.

Transitions act as gate keepers for scenes. And hence a number of activities related to the succeeding scene ambient are carried out at the transition. If the scene ambient instance is not already created, then it is done at the transition. Thus at $TrAnd_{Auction}$, as the *Staff* agents are responsible for creating the scene ambient instance $Id_{auction}$, $new(\mathit{Auction}, Id_{auction})$ is executed and a unique instance id $Id_{auction}$ is returned as the result of this operation. At the $TrXor_{Auction}$, as the $Buyer$ agents are only allowed to join a scene ambient already created, this operation is not performed.

The the Moving in a scene ambient is also performed at the transition. At the $TrAnd_{Auction}$ transition, the agents of role *Staff* move in both the *Auction* and the *Sellersetl* scene ambients.

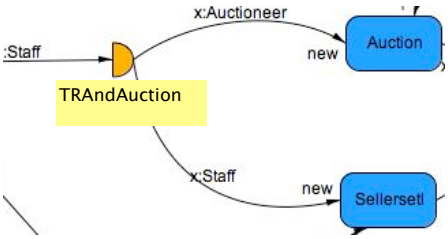

Figure 14: $TrAnd_{Auction}$ Transition

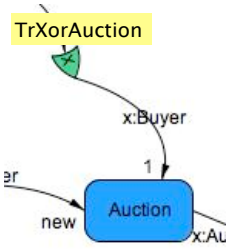

Figure 15: $TrXor_{Auction}$ Transition

At the $TrXor_{Auction}$ transition, the agents of role *Buyer* move exclusively *in* the *Auction* scene ambient.

Another task performed by the transition is, initialization of the scene to the *start* state of the scene. Thus, $put(w, `w0')$ is executed at $TrAnd_{Auction}$. As *Staff* role is responsible for this operation, it is performed only at the $TrAnd_{Auction}$ and not at the $TrXor_{Auction}$.

Finally, initiating the scene is performed by the transitions. Here at $TrAnd_{Auction}$, $a(X, Auctioneer, Auction)$ achieves this task. As with earlier reasoning, initiating the scene is done by the *Staff* agent and hence this operation is performed only at the $TrAnd_{Auction}$ transition. $TrXor_{Auction}$ only enrolls the agents of role *Buyer* into the already created, initiated and running ambient instance $Id_{auction}$).

$a(X, Staff, TrAnd_{Auction})$ :: $new(Auction, Id_{auction})$
then
in $Id_{auction}(Auctioneer)$
then
$put(w,' w0')$
then
$a(X, Auctioneer, Auction)$

par

$new(Sellersetl, Id_{sellersetl})$
then
in $Id_{sellersetl}(Staff)$
then
$put(w,' w0')$
then
$a(X, Staff, Sellersetl)$

$a(B, Guest, TrXor_{Auction})$ :: *in* $Id_{auction}(Buyer)$

5.3 An upward bidding auction scene

Here a typical scene from the above performative structure is elaborated. The translation into ambient LCC is only indicative as how the different elements of the interaction model can be expressed directly in the language. There may be missing elements such as synchronization, to declare the two to be equivalent.

- **Scene specification in Electronic Institution**
  The scene chosen for translation is the *Auction* scene. When translated into ambient LCC, the scene correspond to protocol fragments from the *Buyer* and *Auctioneer* role perspectives. In an upward bidding protocol, the auction scene starts when the *Auctioneer* declares the *startauction* followed by the start of a round selecting a particular *Good*, initial price and bidding time. Then the auction proceeds when the *Buyer* agents request to buy the *Good* for the *Price* announced. If there are one or more *Buyer* agents requesting the *Good* for the last called *Price* then the *Auctioneer* increases the *Price*. The process gets repeated until there is a *Timeout*. If there is no more *Buyer* agents requesting for the *Good* before the *Timeout* expires, then the *Good* is either sold to the last *Buyer* who requested the *Good* for the previous *Price*, or else the *Good* is withdrawn from the auction. The following are the protocol fragments(illocution schema) as specified in the original electronic institution.

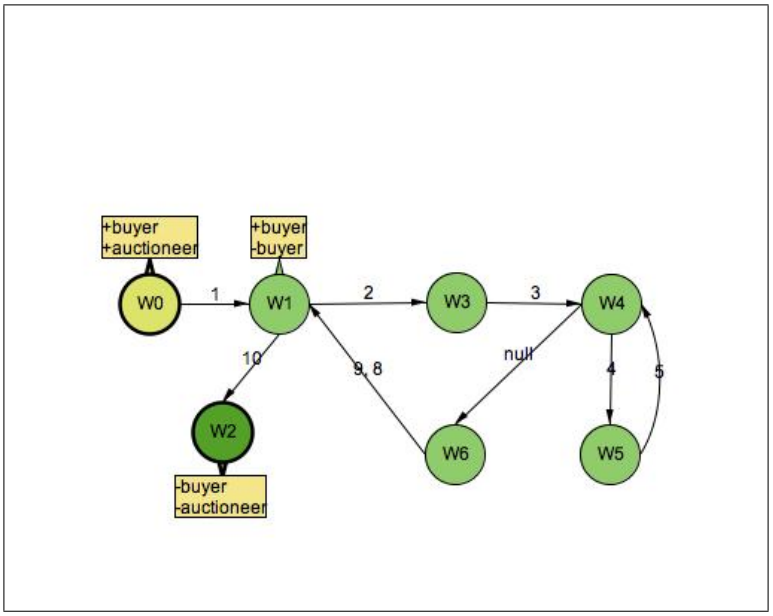

Figure 16: Upward Bidding Auction

1. $inform((?x\ Auctioneer)(all\ Buyer)(startauction\ ?a))$
2. $inform((!x\ Auctioneer)(all\ Buyer)(startround\ ?good\ ?price\ ?bidding\_time))$
3. $inform((!x\ Auctioneer)(all\ Buyer)(offer\ !good\ !price))$

4. $request((?y\ Buyer)(!x\ Auctioneer)(bid\ !good\ !price))$

   $null:\ timeout\ [!bidding\_time]$

5. $inform((!x\ Auctioneer)(all\ Buyer)(offer\ !good\ ?price))$

8 $inform((!x\ Auctioneer)(all\ Buyer)(sold\ !good,\ !price,\ !Buyer_id))$

9 $inform((!x\ Auctioneer)(all\ Buyer)(withdrawn\ !good))$

10 $inform((!x\ Auctioneer)(all\ Buyer)(close))$

- **Translation into Ambient LCC**

The *Auction scene* ambient is defined as follows.

$\Delta_A \quad = \langle Id_A, \tau_A \rangle$

Where The scene specification parameters $\tau_{I_s} = \langle R, DF_S, W, w_0, W_f, (WA_r)_{r\in R}, (WE_r)_{r\in R}, \Theta, \lambda, min, Max \rangle$ for the above scene A are:

$$
\begin{array}{ll}
R & = \{Auctioneer, Buyer\} \\
DF_A & = \langle \{(startauction\ ?a), (startround\ ?good\ ?price\ ?bidding\_time), \\
 & \quad (offer\ !good\ !price), (bid\ !good\ !price), (sold\ !good,\ !price,\ !Buyer_id), \\
 & \quad (withdrawn\ !good), (close)\}, \_, \_ \rangle \\
W & = \{w0, w1, w3, w4, w5, w6, w2\} \\
w_0 & = w0 \\
W_f & = \{w2\} \\
(WA_r)_{r\in R} & = \{\{w0, w1\}_{Buyer}, \{w0\}_{Auctioneer}) \\
(WE_r)_{r\in R} & = \{\{w2\}_{Buyer}, \{w2\}_{Auctioneer}) \\
\Theta & = \{(w0, w1), (w1, w2), (w1, w3), (w3, w4), (w4, w5), (w4, w6), (w6, w1)\} \\
\lambda & = \{(w0, w1) \rightarrow 1, (w1, w2) \rightarrow 10, (w1, w3) \rightarrow 2, (w3, w4) \rightarrow 3, (w4, w5), (w4, w6), (w6, w1)\} \\
 & \quad (w3, w5) \rightarrow msg5\} \\
min & = \{Auctioneer \rightarrow 1, Buyer \rightarrow 1\} \\
Max & = \{Auctioneer \rightarrow 1, Buyer \rightarrow 15\}
\end{array}
$$

Here τ_A specifies the rules that needs to be kept while creating and executing instances of *Auction* scene ambient. It specifies that the only roles allowed to play the *Auction* scene are the *Buyer* and the *Auctioneer* roles. It also specifies the *start* and *end* states of the scene as $w0$ and $w2$. Then it goes on further to specify that, $w0$ is the only state through which both the *Auctioneer* and *Buyer* can enter the scene ambient and $w2$ is the only state through which both the *Auctioneer* and *Buyer* can exit the ambient. It also specifies that the minimum and maximum number of agents that can be present at the ambient at any instance of time is 0 and 100 for the *Buyer* role and 0 and 1 for the *Auctioneer* role. It then specifies how the state change from one state to the next can happen through a number of parameters including λ which stands for the illocutions mentioned above.

The entry into the scene is done from outside the scene, and handled by the preceding transaction ambients. After a successful entry into the scene ambient is done, the scene protocols from the different role perspectives namely the *Auctioneer* and the *Buyer* expanded below are enacted.

There are a few calls to external functions such as $getSaleinfo(Buyer\_id, Quantity)$ and $getGoodsinfo(G, P, B)$ as part of the protocol. We don't specify them here and their meaning should be clear from their names.

$a(X, Auctioneer, Auction)$ ::

$put(\text{“}w\text{”}, \text{“}w1\text{”}) \wedge put(\text{“}GList\text{”}, G_r) \wedge put(\text{“}PList\text{”}, P_r) \wedge put(\text{“}BList\text{”}, B_r) \leftarrow$
$startauction(Id_{auction}) \Rightarrow a(\_, Buyer, Auction))$
$\leftarrow get(\text{“}w\text{”}, \text{“}w0\text{”}) \wedge getGoodsinfo(G, P, B)$

or

$put(\text{“}w\text{”}, \text{“}w2\text{”}) \leftarrow close(Id_{auction}) \Rightarrow a(\_, Buyer, Auction))$
$\leftarrow get(\text{“}w\text{”}, \text{“}w1\text{”}) \wedge GList = []$

or

$put(\text{“}w\text{”}, \text{“}w3\text{”}) \leftarrow$
$startround(Good, Price, Bidding\_time) \Rightarrow a(\_, Buyer, Auction))$
$\leftarrow get(\text{“}w\text{”}, \text{“}w1\text{”}) \wedge$
$GList = [Good|G_r] \wedge PList = [Price|P_r] \wedge BList = [Bidding|B_r]$

or

$put(\text{“}w\text{”}, \text{“}w4\text{”}) \wedge offer(Good, Price) \Rightarrow a(\_, Buyer, Auction)) \leftarrow get(\text{“}w\text{”}, \text{“}w3\text{”})$

or

$put(\text{“}w\text{”}, \text{“}w6\text{”}) \leftarrow get(\text{“}w\text{”}, \text{“}w4\text{”}) \wedge timeout[Bidding\_time]$

or

$bid(Good, Price) \Leftarrow a(Buyer(\_, Buyer, Auction)) \leftarrow get(\text{“}w\text{”}, \text{“}w5\text{”})$

or

$put(\text{“}w\text{”}, \text{“}w4\text{”}) \leftarrow offer(Good, Price) \Rightarrow a(\_, Buyer, Auction))$

or

$put(\text{“}w\text{”}, \text{“}w1\text{”}) \wedge put(\text{“}Buyer\text{”}, Buyer\_id) \wedge put(\text{“}SaleQty\text{”}, Quantity) \leftarrow$
$sold(Good, Quantity, Price, Buyer\_id) \Rightarrow a(\_, Buyer, Auction))$
$\leftarrow get(\text{“}w\text{”}, \text{“}w6\text{”} \wedge Sold\_constarints \wedge getSaleinfo(Buyer\_id, Quantity)$

or

$put(\text{“}w\text{”}, \text{“}w1\text{”}) \leftarrow withdrawn(Good) \Rightarrow a(\_, Buyer, Auction))$
$\leftarrow Withdraw\_cnstr$

or

$a(X, Auctioneer, Auction) \leftarrow get(\text{“}w\text{”}, \text{“}w2\text{”}) \wedge G_r! = []$

or

$open\ Id_{auction} \leftarrow get(\text{“}w\text{”}, \text{“}w2\text{”}) \wedge G_r = []$

$a(B, Buyer, Auction))$ ::

$put(\text{“}w\text{”}, \text{“}w5\text{”}) \leftarrow bid(Good, Price) \Rightarrow a(X, Auctioneer, Auction) \leftarrow get(\text{“}w\text{”}, \text{“}w4\text{”}) \wedge$
$offer(Good, Price) \Leftarrow a(X, Auctioneer, Auction)$

or

$sold(Sale) \Leftarrow a(X, Auctioneer, Auction) \leftarrow get(\text{“}w\text{”}, \text{“}w6\text{”})$

6 Discussion

This document contains the syntax of a new language based on process algebra concepts and specially adapted to the implementation of electronic institution models. However, we believe that the language is general enough to map other distributed computation models. In the future, the synchronization of an interaction model as a whole needs to be addressed in more detail. Here the focus was on synchronization issues that arise within a scene. And the translation algorithm will need to be extended accordingly.

The translation algorithm presented in this document shows how scenes in an Electronic Institutions can be translated into Ambient LCC. The algorithm does not explain how illocutions with constraints and actions are translated, it has been left out of the scope of the document. To further develop the algorithm, it should be able to also translate performative structures and this is still ongoing work. The limitation being that ambient LCC does not allow for the existence of agent alteroids, and therefore only performative structures without parallelism may be translated.

References

[1] L. Cardelli and A. D. Gordon. Mobile Ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computational Structures*, number 1378 in Lecture Notes in Computer Science, pages 140–155. Springer-Verlag, 1998.

[2] Guifré Cuní, Marc Esteva, Pere Garcia, Eloi Puertas, Carles Sierra, and Teresa Solchaga. Masfit: Multi-agent systems for fish trading. In *16th European Conference on Artificial Intelligence (ECAI 2004)*, pages 710–714, Valencia, Spain, August 2004.

[3] M. Esteva, J. A. Rodríguez, C. Sierra, P. Garcia, and J. L. Arcos. On the Formal Specification of Electronic Institutions. In *Agent-mediated Electronic Commerce (The European AgentLink Perspective)*, volume 1991 of *Lecture Notes in Artificial Intelligence*, pages 126–147, 2001.

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

[5] Marc Esteva, Juan A. Rodríguez-Aguilar, Carles Sierra, Josep L. Arcos, and Pere Garcia. On the formal specification of electronic institutions. In Carles Sierra and Frank Dignum, editors, *Agent-mediated Electronic Commerce: The European AgentLink Perspective*, number 1991 in Lecture Notes in Artificial Intelligence, pages 126–147. Springer-Verlag, 2001.

[6] D. Harel. Dynamic logic. In D. M. Gabbay and F. Gunthner, editors, *Handbook of Philosophical Logic*, volume II, pages 497–604. D. Reidel, 1984.

[7] C. Hewitt. Offices are open systems. *ACM Transactions of Office Automation Systems*, 4(3):271–287, 1986.

[8] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (Part 1/2). *Information and Computation*, 100(1):1–77, September 1992.

[9] Pablo Noriega. *Agent-Mediated Auctions: The Fishmarket Metaphor*. IIIA Phd Monography. Vol. 8, 1997.

[10] D. North. *Institutions, Institutional Change and Economics Perfomance*. Cambridge U. P., 1990.

[11] Juan A. Rodríguez-Aguilar. *On the Design and Construction of Agent-mediated Electronic Institutions*. IIIA Phd Monography. Vol. 14, 2001.

[12] Juan A. Rodríguez-Aguilar, Pablo Noriega, Carles Sierra, and Julian Padget. Fm96.5 a java-based electronic auction house. In *Second International Conference on The Practical Application of Intelligent Agents and Multi-Agent Technology(PAAM'97)*, pages 207–224, 1997.

[13] Y. Shoham and M. Tennenholtz. On the Synthesis of Useful Social Laws for Artificial Agent Societies. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, San Diego, USA, 1992.

[14] C. Walton. Model checking for multi-agent systems. *Journal of Applied Logic (to appear)*, 2006.

[15] C. Walton and D. Robertson. Flexible Multi-Agent Protocols. In *Proceedings of UKMAS 2002. Also published as Informatics Technical Report EDI-INF-RR-0164, University of Edinburgh*, November 2002.

[16] Petia Wohed, Wil M. P. van der Aalst, Marlom Dumas, and Arthur H. M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In *Proceedings 22nd International Conference on Conceptual Modelling (ER)*, pages 200–215, 2003.