Coordination Technology for the Development of Multi-Agent Systems on the Web

Enrico Denti, Andrea Omicini, and Vladimiro Toschi
LIA - Laboratorio di Informatica Avanzata
DEIS - Università di Bologna
Viale Risorgimento, 2 40136 – Bologna, Italy
{edenti,aomicini,vtoschi}@deis.unibo.it

Abstract. Multi-agent systems are rapidly becoming one of the most relevant success stories in the area of Artificial Intelligence, in that agent-oriented engineering is foreseen as the next dominant paradigm for the design and development of complex software systems. This introduces in the DAI field new issues concerning methodologies and enabling technologies, and calls for new metaphors and abstractions specifically supporting the engineering of MAS. In particular, the crucial point in MAS engineering is building agent societies. So, coordination models and languages are required, which allow the agent interaction space to be managed so as to rule social behaviours and accomplish social tasks. In this context, we argue that an effective coordination technology is needed, which not only implements a coordination model, but also supports its metaphors at the development system level, and enables MAS designers and developers to work at any time at the abstraction level they need.
In this perspective, this paper presents the LuCe coordination system for the design and development of Web-based MAS. Based on the full integration of Java and Prolog technologies, LuCe implements a coordination model based on the notion of logic tuple centre, a coordination medium which is also exploited as the core of the development system.
The power of the LuCe coordination technology is first discussed in general, then shown in the context of a simple MAS: a TicTacToe game among intelligent software agents and human players on the Web.

Keywords: Multi-Agent Systems, Web Agents, Coordination Models, Coordination Technology, Tuple Centres

1 Introduction
Multi-agent systems (MAS henceforth) are rapidly becoming an ubiquitous paradigm for building complex software applications, particularly in the case of Internet-based systems. On the one hand, such a widespread acceptance is making DAI one of the most important success stories in the history of Artificial Intelligence. On the other hand, it introduces in the AI field new issues coming from the Software Engineering area, such as models and methodologies for the design and development of MAS, and the availability of enabling technologies [4].

In particular, this convergence results in a new research area, called multi-agent system engineering [8]. There, the emphasis is put on task-oriented design, where the
global system goals are delegated to the responsibility [18] of either individual agents (individual tasks) or agent societies (social tasks) [21, 12]. As agent architectures and languages [24] support the design and development of individual agents [13], coordination languages and models [11, 2, 9, 19, 20] are meant to provide for the metaphors and abstractions required to build agent societies [1]. In particular, coordination media should work as the core for agent societies accomplishing social tasks.

Speaking in general, coordination deals with managing the interaction among components [14]; so, in the context of MAS, it addresses the issue of how agents interact. When looking at MAS as at complex software systems to be engineered, coordination models alone cannot face the complexity intrinsically related to building agent societies, especially in the case of the Internet-based ones. More generally, no model, no abstraction can actually help in the development of highly interactive systems [23], like MAS, unless supported by both a suitable development system and a flexible and usable administration environment. In this perspective, we argue that the full exploitation of a coordination model requires a suitable coordination system working as the enabling technology for the engineering of MAS social behaviours.

So, this paper focuses on the issues of coordination technology [1] and of its full exploitation for the engineering of Web-based MAS. Since the design process in both MAS and Web-based systems is typically as critical as their maintenance and incremental development, the availability of a suitable coordination technology turns out to be as essential as the choice of the coordination model. From this viewpoint, we claim that the mere implementation of a coordination model is not enough for a coordination system. Instead, apart from obviously supporting all the abstractions and mechanisms provided by its coordination model, a coordination system should (i) fully exploit the model's metaphors to provide specialised development tools, and (ii) enable developers to choose in any phase of a MAS' life cycle the abstraction level they need.

In this context, this paper presents the LuCe coordination system (from Logic Tuple Centres) for the design and development of Web-based MAS. LuCe exploits a coordination model based on logic tuple centres, a coordination medium which is also exploited as the core of the development system.

This paper is structured as follows. Section 2 presents the coordination model on which LuCe is based, by discussing the core notion of logic tuple centre. Section 3 illustrates the LuCe development system, and discusses its support tools. Section 4 presents a simple application where some logic-based intelligent agents play against human players interacting through the LuCe system on the Web.

2 Coordination in LuCe

In LuCe, agents interact by exchanging logic tuples through a multiplicity of communication abstractions, called tuple centres. The LuCe communication language is based on first-order logic: a tuple is a Prolog ground fact, any unitary Prolog clause is an admissible tuple template, and unification is the tuple matching mechanism. Agents perceive a LuCe tuple centre as a logic tuple space, which can be accessed through the typical Linda-like [10] read/write/consume operations over logic tuples (out, in, rd, inp, rdp). Each tuple centre is uniquely identified by a ground Prolog term, and any ground term can be used to denote a tuple centre. Since LuCe is coordination transparent [15], agents do not have to care about tuple centre existence and creation.

What makes a tuple centre different from a tuple space is the notion of behaviour specification, which defines how a tuple centre reacts to an incoming/outgoing com-
munication event. From the agent’s viewpoint, a tuple centre looks like a tuple space, since it is accessed through the same interface; however, unlike a tuple space, the tuple centre’s behaviour in response to communication events can be defined so as to embed the coordination laws into the coordination medium [5]. In particular, LuCe adopts the tuple centre model defined in [6], where reactions in response to communication events are defined by means of first-order logic tuples, called specification tuples.

So, a logic tuple centre is conceptually structured in two parts: the tuple space, containing ordinary communication tuples, and the specification space, containing specification tuples. This distinction suggests two levels of abstraction over the space of agent interaction: the communication and the coordination viewpoints. By representing at any time the current state of agent interaction, the state of the space of (ordinary) tuples provides for the communication viewpoint. Instead, the space of the specification tuples provides for the coordination viewpoint, since the behaviour specification of a tuple centre governs inter-agent communication, and specification tuples actually define agent coordination rules. On the other hand, since both spaces are collections of unitary logic clauses, they may in some sense be taken as theories of communication and coordination, respectively [16].

LuCe provides agents with a set of predicates (outSpec, rdpSpec, inpSpec, and so on) to access the specification space with the same conceptual protocol used for the space of tuples. Since agents can access both the tuple space and the specification space in a uniform way, they can at any time choose to adopt either the communication or the coordination viewpoint over the multi-agent system they belong to. So, the LuCe LP-based approach enables in principle agents to reason on the system behaviour by taking both the communication and the coordination theories into account, and to possibly refine or change the coordination laws by acting on the specification space.

A logic tuple centre is then a coordination medium which encapsulates both the state of inter-agent communication and the rules of agent coordination expressed in terms of specification tuples. As a result, (i) agents can be designed around their individual tasks, tailoring their interaction protocol to the agents’ desired perception of the interaction space, and (ii) coordination rules can be designed so as to accomplish social tasks, while bridging amongst the different agents’ perceptions and ontologies. The clear distinction between the coordinating (tuple centres) and the coordinated entities (agents) enables multi-agent systems to be designed in a modular way, neatly separating individual and social tasks. So, both system designers and intelligent agents might in principle change the social behaviour of a MAS by properly modifying the specification tuples, or reuse the same coordination rules for problems belonging to the same class by exploiting the same behaviour specification (the same theory of coordination) in different domains.

3 Coordination Technology in LuCe

A tuple centre is characterised by three sets – the set $T$ of its tuples, the set $W$ of its pending queries waiting to be served, and the set $S$ of its reaction specifications. So, the LuCe system comes with a set of specialised LuCe agents to view, edit and control tuple centres both from the data, the pending query and the specification viewpoints: the $T$, $W$ and $S$ Inspectors. By enabling tuples’ observability, the $T$ Inspector provides for the abstraction level of communication in a data-oriented fashion, while the $W$ Inspector, supporting communication events’ observability, provides for the abstraction level of
communication in a control-oriented fashion. Finally, by making specification tuples observable, the S Inspector provides for the abstraction level of coordination.

The T Inspector. The T Inspector is the tuple tracer/editor, activated by issuing a \texttt{tinspect(\textit{Insp}, \textit{TC})} command from the LuCe cell console (Fig. 1), where \textit{Insp} is the name of the inspector agent, and \textit{TC} is the tuple centre to be inspected. As a tracer, it can capture all the state transitions of a tuple centre, and show the current state of the space of tuples. As an editor, it allows the user to add, read or remove tuples by means of the standard LuCe communication primitives \texttt{out}, \texttt{in}, \texttt{rd}, etc.

Since different phases of a MAS’ life cycle may require different views over the MAS, the tracer behaviour can be configured accordingly. In fact, (i) tracing has a cost and (ii) a too fine-grained trace may result confusing, making it difficult to find the relevant events among a plethora of tracing messages. For this reason, the \texttt{Freq} field allows the user to define the refresh ratio (that is, how often the view should be updated, in terms of observable tuple centre transitions), while the \texttt{Pattern} field allows the user to specify a filter template, so that only matching (unifying) tuples are shown. Independently of the refresh ratio, the user is free to force an update of the tracing view at any time, by means of the \texttt{Update} button.

As a tuple editor, the T Inspector allows the user to directly perform tuple operations: the involved tuple is first typed in the \texttt{Request} field, then the proper operation is triggered by pressing the proper button (\texttt{Out}, \texttt{Rd}, \texttt{Rdp}, \texttt{In}, \texttt{Inp}): the \texttt{Answer} field shows the operation result, if any. For instance, Fig. 1 shows the result of a \texttt{rd(p(X))} operation. User-triggered operations are observationally indistinguishable from any other agent-triggered operations, and activate the related reactions, if any, according to the current behaviour specification. This feature allows the tuple centre’s behaviour to be tested separately from agents, so that social tasks can be developed separately from individual ones.

The W Inspector. The W Inspector is the pending query tracer. This tool is activated from the console with a \texttt{winspect(\textit{Insp}, \textit{TC})} command, and works similarly to the T Inspector, allowing tuple centre’s incoming communication events to be fully monitored. Moreover, by means of the \texttt{Out}, \texttt{Rd}, \texttt{Rdp}, \texttt{In}, and \texttt{Inp} buttons, the W Inspector allows new communication events to be triggered, so that the behaviour of a tuple centre can be tested by observing the evolution of the pending query queue.
The S Inspector. The S Inspector is the specification editor, and is launched from the LuCe console by issuing a `sinspect(Insp,TC)` command. Unlike the T and W Inspectors, this tool does not provide the Out, Rd, Rdp, In, Ins buttons for specification tuples. Instead, it means to capture the typical programmer’s “way of doing”, based on writing programs (reactions) in a file, edit them, and reload the new specifications when done. Accordingly, it provides the programmer with controls to Consult, Edit, Update and Save a behaviour specification, along with syntax checking capabilities.

4 TicTacToe in LuCe

In this section we show an example of Web-based MAS design and development, by describing in short a simple LuCe-coordinated MAS implementing the classical TicTacToe game. In its basic version, the game is played on a grid of $3 \times 3$ cells by two players, identified by circles (o) and crosses (x). Each player aims at putting three pieces of his/hers so as to fill a line of the grid (vertically, horizontally or diagonally), while trying to prevent the other player from doing the same. The application scenario is a Web-based TicTacToe arena, where any human player from anywhere on the Web can at any time ask to enter a game, and is ensured to find a game and an opponent – a human one, if available, or a software agent, if needed.

Despite of its simplicity, this application demonstrates how the LuCe coordination model and technology impacts on the design and development of Web-based MAS, by making it possible to design and develop individual and social tasks separately, and to connect them through straightforward communication protocols.

4.1 Design by Tasks

The design of a MAS should be driven by the definition of individual and social tasks [13]: the former to be delegated to single agents, the latter to be charged upon agent societies. In a LuCe-coordinated MAS, social behaviours are built around tuple centres, suitably ruling the agent interaction space [12].

Our MAS’ tasks can be summarised as follows: (i) concurrently manage several TicTacToe games, enabling any human player (ii) to enter a game and always find an opponent, either a human or a software one, (iii) to ask for suggestions, if needed, and (iv) to leave the game at his/her will.

Correspondingly, our MAS needs GUI agents handling human interaction and representing human players in the system, and expert agents owning TicTacToe logic for playing, suggesting, and validating moves. More precisely, our MAS is composed by (i) one GUI agent (written in Java) for each human being currently playing, (ii) one expert agent (written in Prolog) for each game currently played, and (iii) a single master agent in charge of starting the MAS and activating a new expert agent for every newly-created game. Agent interaction is handled by the tictactoe tuple centre, which stores the information about TicTacToe games, bridges between different agent representation of the application domain, and embodies the laws of agent coordination.

4.2 Individual Tasks and Interaction Protocols

Once agents are assigned a task, their interaction protocols can be defined according to simple information-oriented criteria: which information can be made available, and
when, which information is needed, and when, how information is to be represented and accessed, and so on. Since any interaction occurs via the tictactoe tuple centre, designing the agent protocols in our MAS amounts at defining how agents exchange tuples with tictactoe.

**The Master Agent.** The master agent is simply in charge of (i) initialising the tictactoe tuple centre and (ii) starting a new expert agent for every newly-created game. Thus, it first initialises tictactoe by emitting an initialise tuple, then it starts its working cycle, by repeatedly synchronising on a newGame(ID) tuple, representing the need for the creation of a new TicTacToe game. Whenever such a tuple is found, the master agent consumes it, and activates a new expert agent for the newly-created game ID.

**The Expert Agent.** Each expert agent is dedicated to a single TicTacToe game, and is conceptually in charge of three tasks: (i) validating human players’ moves, (ii) playing in place of a human player, if needed, and (iii) suggesting moves, if required. However, since the logic needed for both making and suggesting a move is the same, only tasks (i) and (ii) are actually delegated to expert agents: so, these tasks are designed as individual ones, by definition. Instead, task (iii) is identified as a social one, and so delegated to the agent ensemble constituted by the GUI agent(s) representing the human(s) playing a game and by the corresponding expert agent, ruled by the tictactoe coordination medium.

Expert agents are written in Prolog and have no GUI: they just perform a polling loop, which repeatedly looks for expertTask/3 tuples to consume (which represents tasks to be accomplished by the expert), and ends when no gameOn(ID) tuple is found (which corresponds to the end of the game handled by the expert). More precisely:

- an expertTask(ID,Status,validate(X,Y)) tuple represents the request to validate a move on cell (X,Y). The expert answers by emitting either an invalid(ID, on(X,Y)) tuple if the move is not valid, or a valid(ID, on(X,Y),NewStatus) tuple otherwise, where NewStatus is either an s/9 or an end/2 term, in the same way as the Status argument in the gameStatus/4 tuple (see Subsection 4.3).
- an expertTask(ID,Status,play(Role)) tuple makes the expert agent evaluate game Status and respond by emitting an expertMove(ID,Role,on(X,Y),NewStatus) tuple, representing its move (X,Y) as a Role player.

**The GUI Agent.** A GUI agent is started by a human player willing to enter a game, typically via an Internet browser. Its tasks are quite obvious: (i) mapping human player’s commands (move, suggest, leave the game) onto logic tuples, and (ii) providing human player with his/her specific view of the game status.

So, on startup, the GUI agent gets a joinGame(ID,Role) tuple which assigns role Role in game ID to the new player. Then the agent’s main loop begins, getting the game status (tuple gameStatus(ID,Status,Next)), and showing it to the human player.

If Next is not Role, it is the opponent’s turn to move. The GUI agent restarts its main loop, unless the player quits the game closing the GUI window and causing a leaveGame(ID,Role) tuple to be output and the GUI agent to quit. If, instead, Next is Role, it is the player’s turn to move, and three commands are made available:

- try to put Role’s sign on cell (X,Y) of the game ID grid. The request for a humanMove(ID,Role,on(X,Y),OK) tuple signals the user’s desired move and makes the agent wait for validation by the expert agent. If the move is accepted, OK is true in the answer tuple, and the agent’s main loop restarts. Otherwise, OK is false, the game status is unmodified, and the GUI agent waits for the next player’s command.
Fig. 2. A portion of the tictactoe behaviour specification.

reaction(out(initialise), ( inᵦ(initialise),
          outᵦ(id(1)), outᵦ(initialStatus(s(n,n,n,n,n,n,n,n,n))))).

reaction(inᵦ(id(1)), ( post, current_tuple(id(OldID)), NewID is OldID+1, outᵦ(id(NewID)) ))).

reaction(inp(gameOn(ID)), ( pre, outᵦ(gameOn(ID)),
          rdᵦ(gameStatus(ID,gameStatus(ID,ID,Status,Next)),
          outᵦ(gameStatus(ID,Status,Next)) ))).

reaction(rd(gameStatus(ID,Status)), ( pre,
          rdᵦ(gameStatus(ID,Status,Next)),
          outᵦ(gameStatus(ID,Status,Next)) ))).

reaction(in(joinGame(ID)), ( pre, noᵦ(freeRole(ID)),
          inᵦ(id(ID)), outᵦ(newGame(ID)) ))).

reaction(in(joinGame(ID)), ( pre, inᵦ(freeRole(ID,Role)),
          outᵦ(joinGame(ID,Role)) ))).

reaction(in(newGame(ID)), ( post, current_tuple(newGame(ID)),
          outᵦ(freeRole(ID,x)), outᵦ(freeRole(ID,o)),
          inᵦ(freeRole(ID,Role)), outᵦ(joinGame(ID,Role)),
          rdᵦ(initialStatus(IS)), outᵦ(gameStatus(ID,0,IS,Role)) ))).

- ask for a suggestion. A suggest(ID,Role,SuggestedMove) tuple is required, and the answer tuple contains the move on(X,Y) suggested by the expert agent. The suggested move is visualised (as an ‘S’) by the GUI agent on the grid, then the GUI agent keeps waiting for the player’s new command.
- quit the game. An out(leaveGame(ID,Role)) operation is performed, then the GUI agent quits.

4.3 Social Tasks

Tuple centres work as the core for the engineering of agent societies, to which MAS’ social tasks are delegated. In particular, a tuple centre is in charge of (i) representing knowledge about the domain state, (ii) mediating amongst the different agent’s perceptions of the domain, and (iii) ruling agent interaction in order to carry out the social tasks. In the following, we discuss the design of the tictactoe tuple centre, and show some relevant portions of the tictactoe’s behaviour specification (partially reported in Fig.2). For the full specification, we forward the reader to the URL http://www lia.deis.unibo.it/Research/LuCe/SampleApp/TicTacToe/.

Domain Representation. All the information about TicTacToe games is stored in tictactoe in form of logic tuples. Games are labelled progressively with an integer value, so that a gameStatus(ID,MoveNo,Status,Next) tuple in tictactoe represents the current Status of the game ID after move MoveNo has been performed, while waiting for the next move from player Next (x or o). Status is either an s/9 term representing the state of the 9 grid’s cells by means of n (no sign yet), x or o constants, or an end/2 term denoting the game end.

A move(ID,MoveNo,Player,Role,on(X,Y)) tuple records move MoveNo of game ID performed by player Player acting as Role and putting its sign (x or o) on the grid
cell \((X, Y)\). A freeRole\((ID, Role)\) tuple in tictactoe indicates that no human player is currently playing as Role \((x\ or\ o)\) in game ID: so, a human player asking to enter a game can freely join game ID, and play as Role. The tuple id\((ID)\) records the label for the next game to be started, which is initialised as id\((1)\) and then incremented at any new game creation (Reaction 2 in Fig. 2). The term representing the initial status for any TicTacToe game is recorded by the initialStatus\((s(n,n,n,n,n,n,n,n,n))\) tuple. Both the initial id/1 and initialStatus/1 tuples are produced by Reaction 1 in Fig. 2 in response to the master agent’s emission of the initialise tuple.

**Agent Perception.** The design of the agent interaction protocol is based on the agent perception of the interaction space, determined in LuCe by the tuple centre behaviour specification. In our sample MAS, for instance, expert agents perceive tictactoe as containing gameOn\((ID)\) tuples, denoting that game ID is still going on. Analogously, GUI agents perceive tictactoe as containing gameStatus\((ID, Status, Next)\) tuples. However, these tuples are not part of the domain representation: instead, gameOn/1 and gameStatus/3 tuples are how the gameStatus/4 tuples actually contained in tictactoe are perceived by expert and GUI agents, respectively, thanks to Reactions 3–4 in Fig. 2.

**Social Behaviour.** Since a tuple centre is the space where interaction occurs, and is programmable, it is the most natural place where to embed coordination laws: so, its behaviour can be used to rule agent mutual interaction.

For instance, in our sample system, a GUI agent asking for a free role in a game by consuming a joinGame\((ID, Role)\) tuple may either involve a new game creation, or, alternatively, allow the human player to replace an expert agent via the GUI agent in an open game. This coordination policy is achieved by means of Reactions 5–7 in Fig. 2, where freeRole/2 tuples are used to denote roles played by expert agents in open games. When no freeRole/2 tuples are available, a new game is created: so, a new game ID is obtained, and a newGame/1 tuple is inserted, making the master agent trigger a new expert agent for the newly-created game.

All the other social tasks, such as handling game end, validating and suggesting moves, enabling human players to enter and quit games transparently, are implemented analogously. (See the LuCe home site for details.)

### 4.4 Development in LuCe

**Agents.** Given the uncoupling of agents induced by tuple-based coordination [11, 7], T and W Inspectors enable agents to be developed separately. The T Inspector allows the content of tictactoe to be detected and changed by the developer so as to accomplish the interaction protocol defined for the specific agent under development. In conjunction with the W Inspector, it also makes it possible to trace and check the agent observable behaviour.

**Agent societies.** Once agent interaction protocols are defined, the tictactoe behaviour can be developed and tested independently of agents, using S, T, and W Inspectors. T and W Inspectors are used to simulate the agent interaction protocols defined, thus supporting their separate testing. The S Inspector is obviously exploited to program and verify the behaviour of the tictactoe tuple centre.

### 5 Related Works and Conclusions

Like LuCe, PageSpace [3] focuses on combining coordination technology and the Web, and relies on a tuple-based model for MAS coordination. PageSpace is a reference
architecture for Web agents, defining several agent classes. In particular, agents like kernel agents are used in PageSpace to provide for global system features, unlike LuCe, which exploits for the same purpose the behaviour of the coordination media.

Integrating logic-based and object-oriented technology for Web-based coordination is the starting point of Jinni [22], too. Jinni [22] (Java Inference Engine and Networked Interactor) is a light-weight, multi-threaded, pure logic programming language to be used as a scripting tool to glue Java objects and knowledge processing components together in distributed MAS. The Jinni model is based on the metaphors of things, places and agents, completed with a shared blackboard with Linda-style primitives as the communication and coordination support. Like LuCe, Jinni exploits logic programming as the basic glue for the design and development of Web-based MAS, and integrates it with Web technology by putting logic programming on top of Java.

With respect to all the above approaches, the LuCe coordination system (freely available at http://www.lia.deis.unibo.it/Research/LuCe/) focuses on exploiting coordination technology to effectively support the design and development of Web-based MAS. In particular, we claim that simply supporting a coordination model is not enough to make coordination technology appealing and actually fruitful. Since the issues of application maintenance and incremental development in Web-based MAS are as critical as the design one, the LuCe coordination system provides specialised development tools (the inspectors) based on the coordination model’s metaphors (the tuple centres), which enable developers of Web-based MAS to choose at any time the desired abstraction level over the agent interaction space.

In cooperation with other research groups, we are currently addressing some of the LuCe open problems in the context of the work on the TuCSoN coordination model [17], which is meant to provide for a single, coherent framework where coordination is taken as the unifying paradigm accounting for mobility, authentication, authorisation, and support for intelligent agent exploration.

References


