The LuCe Coordination Technology for MAS Design and Development on the Internet

Andrea Omicini, Enrico Denti, 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. Internet-based multi-agent systems call for new metaphors, abstractions, methodologies and *enabling technologies* specifically tailored to agent-oriented engineering. While *coordination models* define the framework to manage the space of agent interaction, ruling social behaviours and accomplishing social tasks, their impact on system design and development calls for an effective *coordination technology*.
This paper presents LuCe, a coordination technology that integrates Java, Prolog and the notion of *logic tuple centre*, a programmable coordination medium, into a coherent framework. The power of the LuCe coordination technology is first discussed in general, then shown in the context of a simple yet significant system: a TicTacToe game among intelligent software agents and human players on the Internet.

1 Introduction

Multi-agent systems (MAS) are becoming an ubiquitous paradigm for building complex software applications, introducing in the AI field new issues coming from the Software Engineering area, such as the need for models and methodologies for MAS engineering, and the availability of *enabling technologies* [5]. This convergence results in a new research area, called *multi-agent system engineering* [9]. There, the emphasis is put on *task-oriented design*, where the global system goals are delegated to the responsibility of either individual agents (*individual tasks*) or agent societies (*social tasks*) [3,20,13].

While *coordination* deals in general with managing the interaction among components [16], in this context it addresses the issue of how agents interact: as agent architectures and languages [21] support the design and development of individual agents [14], *coordination models* [12,2,10,4,17,18,19] provide the metaphors and abstractions required to build agent societies [1]. In particular *coordination media*, which embed social rules as coordination laws, can work as the core for agent societies pursuing social tasks. However, coordination models alone cannot face the intrinsic complexity of building agent societies: the full exploitation of a coordination model for the design and development of multi-agent social behaviour requires a suitable *coordination system*, providing engineers with the enabling technology they need.

A. Porto and G.-C. Roman (Eds.): COORDINATION 2000, LNCS 1906, pp. 305–310, 2000.

This paper discusses how a *coordination technology* [1] like the LuCe (Logic Tuple Centres) coordination system can be exploited for the engineering of Internet-based MAS, exploiting the model's metaphors to provide ad-hoc development tools. As a case-study, we discuss a TicTacToe game among intelligent software agents and human players on the Internet, showing how the LuCe technology can effectively support the system design and development.

2 LuCe: Model and Technology

LuCe agents interact by exchanging *logic tuples* through a multiplicity of communication abstractions called *tuple centres*. The 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 [11], which can be accessed through the typical Linda-like operations (`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.

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 communication event. From the agent's viewpoint, a tuple centre looks like a tuple space, and is accessed via the same interface: however, the tuple centre's behaviour in response to communication events is not fixed once and for all, but can be extended so as to embed the coordination laws into the coordination medium [6]. In particular, LuCe adopts the tuple centre model defined in [7], where the behaviour is defined in terms of *reactions* to communication events, expressed as first-order logic tuples, the *specification tuples*.

So, a logic tuple centre is conceptually structured in two parts: the *tuple space*, with ordinary communication tuples, and the *specification space*, with specification tuples. Agents can then be designed around their individual tasks, defining interaction protocols according to their own perception of the interaction space: tuple centres' coordination laws will take care of "gluing things together".

The clear distinction between the coordinating entities (tuple centres) and the coordinated entities (agents) naturally decouples individual and social tasks: the system social behaviour can then be changed just by changing the specification tuples. Moreover, the same coordination rules can be reused for other problems of the same class, applying the same behaviour specification to a different domain.

As far as technology is concerned, the LuCe system is built around three major ingredients: the Java technology (for the system development and the Web interface), Prolog, as the *lingua franca* for communication among heterogeneous Web agents, and tuple centre technology, exploited as the core of the LuCe system. In terms of development tools, since a tuple centre is characterised at any time by the set T of its tuples, the set W of its pending queries, and the set S of its reaction specifications, LuCe supplies a set of *Inspectors* to view, edit and control tuple centres from the data, the pending query and the specification viewpoints.

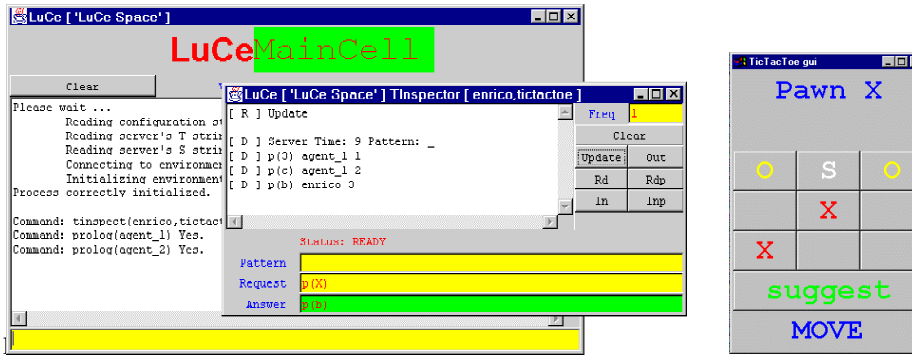

Fig. 1. A LuCe T Inspector in front of the LuCe console (left) and a GUI agent after asking for a suggestion (right)

The ***T Inspector*** is the tuple tracer/editor: as a tracer, it captures the state transitions of a tuple centre and shows the current state of the space T of tuples. As an editor, it allows the user to add, read or remove tuples via the standard communication primitives (*in*, *out*, *rd*, etc). Since user operations must be indistinguishable from agent ones, users cannot add or delete tuples directly: they can only perform communication operations via the proper buttons, so that any associated reactions are triggered. This is crucial to actually mimic agent's operations, so that tracing can be faithful. Other controls (see Fig. 1, centre) configure the tool's refresh frequency (how often the view should be updated, in terms of tuple centre transitions), and filtering options (which tuples to show).

The ***W Inspector*** is the pending query tracer, and works very much like the T Inspector: however, it provides users with the communication viewpoint in a *control-oriented* fashion, unlike the *data-oriented* fashion of the T Inspector.

The ***S Inspector*** is the specification editor: its purpose is to let users write reactions in a file, edit them, and reload the new specifications when done. So, no tuple space operations (*in*, *out*, *rd*, etc) are supported. Rather, it provides the user with buttons to *Consult*, *Edit*, *Update* and *Save* a specification set.

3 The Case Study: TicTacToe

The TicTacToe game is played on a grid of 3×3 cells by two players, namely circles (`o`) and crosses (`x`). Each player aims to put three pieces as to fill a line of the grid (vertically, horizontally or diagonally), while trying to prevent the other from doing the same. The application scenario is a Internet-based TicTacToe arena, where players can enter the game arena at any time, and must be ensured to find a game and an opponent – a human one, if available, or a software agent, if needed. Despite of its simplicity, the TicTacToe application features many of the typical issues of Internet-based multi-agent systems, yet it is simple enough to show the impact of the LuCe technology in a few pages.

3.1 Design by Tasks

As discussed in [14], the design of a multi-agent system can start by defining individual and social tasks: the former are to be delegated to single agents, the latter to the coordination media. Here, tasks include concurrently managing several games, enabling new players to enter a game (and always find an opponent), asking for suggestions, and leaving the game at their will.

The system can then be built using two agent categories: *GUI agents*, handling human interaction, and *expert agents*, owning the logic for playing, suggesting, and validating moves. More precisely, there will be one GUI agent (written in Java) for each human being currently playing, one expert agent (written in Prolog) for each game currently played, and a single *master agent* to start the multi-agent system and activate a new expert agent for every newly-created game. The `tictactoe` tuple centre stores game information and implements the desired coordination laws, bridging between the different domain representations.

3.2 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 is available / needed and when, how it is represented / accessed, etc. Given the uncoupling of agents induced by tuple-based coordination [12,8], each agent can be developed separately, using Inspectors to simulate the effects of the missing agents.

The Master Agent. The master agent's task consists of initialising the tuple centre and starting a new expert agent for every newly-created game. Initialisation is made by emitting an `init(`*`TupleList`*`)` tuple: it is the tuple centre's task to turn this into a set of single tuples. Then, the master starts repeatedly performing an `in(newGame(`*`ID`*`))` tuple, representing the need for the creation of a new TicTacToe game. Whenever such a tuple is found, the master agent activates a new expert agent for the newly-created game *`ID`*.

The Expert Agent. Each expert is dedicated to a single game: it validates human players' moves, plays in place of a human player (if needed), and suggests moves (if required). Since the logic for making and suggesting a move is the same, the expert actually needs to be able to perform only the first two tasks: turning its ability into a move or a suggestion is up to the coordination medium, in function of the current coordination state.

The expert agent is naturally written in Prolog. Basically, it repeatedly consumes `expertTask(`*`+ID,-Task`*`)` tuples, and performs the indicated *`Task`*. If *`Task`* is `validate(`*`+GridS,+Role,+Move`*`)`, the agent validates the *`Move`* and emits either an `invalid/2` or a `valid/4` tuple. If, instead, *`Task`* is `play(`*`+GridS,+Role`*`)`, the expert proposes a move according to the *`GridS`* status of game *`ID`*, emitting an `expertMove/5` tuple. Finally, if *`Task`* is `quit`, the expert just quits.

The GUI Agent. Each GUI agent represents a human player acting via an Internet browser. On startup, it tries to join the game arena by performing an `in(joinGame(`*`-ID,-Role`*`))` operation, getting its game *`ID`* and *`Role`*. Then, it

starts capturing the game status in terms of a `statusView(ID,Role,GridS,Turn)` tuple, and displays it. *Turn* may be either *opponentTurn* or *yourTurn*: in the first case the agent just restarts its main loop, otherwise three commands are enabled: making a move, asking for a suggestion, and leaving the game.

The first two tasks cause an `in(humanMove(+ID,+Role,+Move,-OK))` or an `in(suggest(+ID,+Role,-Move))` operation to be performed, respectively: the returned tuple will contain either the validation by the expert agent (i.e., *OK* is `true` or `false`), or the proposed *Move*. The suggestion is shown as an 'S' on the GUI grid (Fig. 1, right). The intention of leaving the game, instead, just causes a `leaveGame(+ID,+Role)` tuple to be output, then the agent terminates.

3.3 Social Tasks

Tuple centres store the domain knowledge, mediate amongst the different agent's domain perceptions, and rule agent interaction so as to carry out the multi-agent system's social tasks. In the following, we just sketch the main related issues: for a deeper discussion and the full specification, we forward the reader to [15].

Domain Representation. The fundamental information concerns each game's status: as long as game *ID* is running, a `gameStatus(ID,MoveNo,GridS,GameS,Next)` tuple represents the status *GameS* of game *ID* after move *MoveNo* has been performed, while waiting for the next move from player *Next* (`x` or `o`). The presence of a `freeRole(ID,Role)` tuple indicates that no human player is playing as *Role* in game *ID*, which is therefore played by the expert agent. When the game ends, the `gameStatus` tuple is replaced by either a `win(ID,Role,MoveNo,GridS)` or a `stalemate(ID,MoveNo,GridS)` tuple, respectively.

Agent Perception. Each agent interaction protocol is designed around the agent's perception of the interaction space, and is independent both of the other agents' protocols and of the actual information representation in terms of tuples in `tictactoe`. For instance, GUI agents perceive the game status as `statusView/4` tuples, though such tuples do not actually exist: they are dynamically produced in response to the agent's requests to consume them, according to `tictactoe`'s coordination laws. In the development and test phases, this portion of the `tictactoe` behaviour specification may be tested via the S and T Inspectors, using their controls to simulate the actions of the missing agents.

Social Behaviour. The tuple centre embeds the social rules to drive agents' mutual interaction. For instance, a social behaviour is to ensure that any human player can always enter the game arena and find an opponent: no single agent could take care of this task, since master and expert agents have no human interface, while GUI agents are pure interface agents. This social behaviour is achieved by means of proper reactions, which intercept the `in(joinGame(...))` operation, and exploit the `freeRole` tuple to produce the proper `joinGame` response – creating a new game if no `freeRole` exists.

All the other social tasks, including handling game end, validating and suggesting moves, enabling human players to enter and quit games transparently, are implemented analogously, as shown in [15].

References

1. P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2):300–302, June 1996.
2. P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models – Proc. of the 1st International Conference (COORDINATION'96)*, volume 1061 of *LNCS*, Cesena (I), 1996. Springer-Verlag.
3. P. Ciancarini, A. Omicini, and F. Zambonelli. Multiagent system engineering: the coordination viewpoint. In N. R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Agent Theories, Architectures, and Languages*, volume 1767 of *LNAI*, pages 250–259. Springer-Verlag, 2000.
4. P. Ciancarini and A. L. Wolf, editors. *Coordination Languages and Models – Proceedings of the 3rd International Conference (COORDINATION'99)*, volume 1594 of *LNCS*, Amsterdam (NL), 1999. Springer-Verlag.
5. M. Cremonini, A. Omicini, and F. Zambonelli. Multi-agent systems on the Internet: Extending the scope of coordination towards security and topology. In *[9]*, pages 77–88, 1999.
6. E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In *[10]*, pages 274–288, 1997.
7. E. Denti, A. Natali, and A. Omicini. On the expressive power of a language for programming coordination media. In *[17]*, pages 169–177, 1998.
8. E. Denti and A. Omicini. Designing multi-agent systems around a programmable communication abstraction. In J.-J. C. Meyer and P.-Y. Schobbens, editors, *Formal Models of Agents*, volume 1760 of *LNAI*, pages 90–102. Springer-Verlag, 1999.
9. F. J. Garijo and M. Boman, editors. *Multi-Agent Systems Engineering – Proc. of the 9th European Workshop on Modelling Autonoumous Agents in a Multi-Agent World (MAMAAW'99)*, volume 1647 of *LNAI*, Valencia (E), 1999. Springer-Verlag.
10. D. Garlan and D. Le Métayer, editors. *Coordination Languages and Models – Proceedings of the 2nd International Conference (COORDINATION'97)*, volume 1282 of *LNCS*, Berlin (D), 1997. Springer-Verlag.
11. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
12. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
13. F. Hattori, T. Ohguro, M. Yokoo, S. Matsubara, and S. Yoshida. Socialware: Multiagent systems for supporting network communities. *Comm. of the ACM*, 42(3):55–61, March 1999. Special Section on Multiagent Systems on the Net.
14. N. E. Jennings. Agent-oriented engineering. In *[9]*, pages 1–7, 1999. Invited talk.
15. `LuCe` home page. `http://lia.deis.unibo.it/research/LuCe/`.
16. T. Malone and K. Crowstone. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.
17. *Proc. of the 1998 ACM Symposium on Applied Computing (SAC'98)*, Atlanta (GA), 1998. ACM. Track on Coordination Models, Languages and Applications.
18. *Proc. of the 1999 ACM Symposium on Applied Computing (SAC'99)*, San Antonio (TX), 1999. ACM. Track on Coordination Models, Languages and Applications.
19. *Proc. of the 2000 ACM Symposium on Applied Computing (SAC 2000)*, Como (I), 2000. ACM. Track on Coordination Models, Languages and Applications.
20. M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):55–61, December 1998.
21. M. Woolridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.