

Engineering Multi-Agent Systems in LuCe

Enrico Denti and Andrea Omicini

LIA - Laboratorio di Informatica Avanzata
DEIS - Università di Bologna
Viale Risorgimento, 2 40136 – Bologna, Italy
{edenti,aomicini}@deis.unibo.it

Abstract. Logic languages have already proved to be effective in developing single agents and enabling inter-agent communication in multi-agent systems. In this paper, we show how LP can also be exploited for multi-agent coordination, by *ruling* inter-agent communication so as to build social behaviours. For this purpose, we present the LP-based LuCe model and technology for the coordination of Internet agents, based on the notion of *logic tuple centre* as the core of the agent interaction, and show its impact on the design and development of multi-agent systems.

Keywords: Multi-Agent Systems, Logic Programming, Coordination Models, Software Engineering, Logic Tuple Centres.

1 MAS & LP

In the early age of object-oriented systems, logic programming exhibited its weakness in supporting programming-in-the-large, mainly due to its lack of features in terms of structuring capabilities [24]. Several attempts to develop coherent models and languages for object-oriented logic programming were then made [14, 17], and several LP systems tried to add OO capabilities to their languages [21].

However, only the shifts from the object-based to the component-based, and then to the agent-based paradigm finally made LP a viable tool for effectively building complex software systems. In component-based systems, the technology used for developing single components is no longer bound to be unique, and to be the same used to make component interact. So, LP can be naturally exploited for fast development of small, specialised components, which can be easily refined in an incremental way. In turn, agent-based systems stress autonomy of software components, whose deliberative capabilities can fruitfully exploit both the declarative representation of the world and the inferential capabilities featured by LP. This led to several proposals of LP-based frameworks for the development of multi-agent systems, notably Jinni [22] and CaseLP [13].

Nevertheless, the design of multi-agent systems can not merely focus on single components and their subsequent composition in an ensemble. Instead, the holistic (non-compositional) view of multi-agent systems currently promoted by several recent research efforts [20, 11] recognises that social rules, far from being a mere limitation of the behaviour of individuals, constitute one further source

of intelligence in multi-agent systems, going beyond that provided by individual agents. This view suggests that, in the context of multi-agent systems, engineering social aspects is at least as much relevant as engineering single agents. In the same way as agent languages and architectures [25] provide for abstractions and tools to build up agents, *coordination languages*, *models*, and *architectures* [3] provide for the abstractions and tools required to rule the space of agent interaction, thus supporting the engineering of social tasks in multi-agent systems [4].

Furthermore, when considering real-world multi-agent systems, the Internet is today the framework of election [12]: there, multi-agent technology is typically exploited to enable autonomous and heterogeneous components to be combined so as to bring intelligence into Web-based applications. First, this requires any LP-based MAS development framework to be rooted in the standard Web models and technologies, namely the Java language and architecture. Moreover, since the design process in Web application development is typically as critical as application maintenance and incremental development, the availability of a suitable *coordination technology* turns out to be as essential as the choice of the coordination model.

The aim of this paper is then to show how LP can support the engineering of multi-agent systems not only in providing for languages for single agent development (as computation languages) and inter-agent communication (as communication languages), but also in

- building social behaviours, by expressing the laws for the coordination of agent societies, and
- enabling intelligent agents to reason about the state of the agent interaction space, and possibly refine coordination laws so as to accomplish social tasks.

For this purpose, Section 2 introduces the LP-based LuCe system (from Logic Tuple Centres) for the coordination of Internet agents, which

- fully integrates Prolog and Java within the Internet platform, by exploiting a light-weight Prolog engine entirely written in Java,
- adopts logic tuples for inter-agent communication,
- exploits *logic tuple centres* as the coordination media, providing developers with specialised development tools,
- enables social tasks to be accomplished by specifying the behaviour of the coordination media in terms of logic tuples, and
- allow intelligent agents to modify the behaviour of a multi-agent system by acting on the specification logic tuples.

Then, Section 3 shows the impact of LuCe on the design and development of Web-based multi-agent systems, by presenting some examples of LuCe-coordinated systems.

2 Coordination in LuCe

The adoption of Prolog for inter-agent communication has been already successfully experimented, particularly in the context of tuple-based coordination

models [15] such as Shared Prolog [1], ESP [2], Linda Interactor [23], and *ACCT* [16]. The LuCe communication language, too, 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 [15].

What is new in LuCe is the adoption of *logic tuple centres* as coordination media which allow LP to be exploited not simply to enable inter-agent communication, but also so as to rule it. Agents perceive a LuCe tuple centre as a logic tuple space, which can be accessed through the typical Linda-like [10] read/write/consume predicates (**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. For a formal description of the LuCe communication and coordination languages, we forward the interested reader to [15]: there, the semantics of a logic-based coordination models is defined, whose communication and coordination languages are identical to the LuCe ones.

What makes a (logic) tuple centre different from a (logic) 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, 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].

The behaviour of a LuCe tuple centre is defined through the ReSpecT specification language [6], a logic-based language where *reactions* in response to communication events are defined by means of first-order logic tuples, called *specification tuples*. Given a communication event generated by an incoming operation $\mathcal{Op}$, a specification tuple $\text{reaction}(\mathcal{O}, R)$ associates a reaction Re to the event, where $Re = R\theta$ and $\theta = mgu(\mathcal{O}, \mathcal{Op})$. A ReSpecT reaction is then executed as a sequence of *reaction goals*, which can access properties of the triggering event and perform tests on terms. Moreover, reaction goals can operate on the space of the tuples through the **out_r**, **in_r**, **rd_r**, and **no_r** predicates, which in turn may trigger further reactions in a chain. The main ReSpecT predicates for reactions are reported in Table 1.

Reaction goals are executed sequentially. A reaction as a whole either succeeds or fails depending on whether all its reaction goals succeed or not, and is executed with a transactional semantics: so, a failed reaction has no effect on the tuple centre state. All the reactions triggered by a communication event are executed before serving any other event: so, agents perceive the result of serving the communication event and executing all the associated reactions altogether as a single transition of the tuple centre state. More details on the ReSpecT language and its computational model can be found in [6].

As a result, the effect of a communication primitive is no longer limited to adding, reading, or removing a single logic tuple, as in the case of logic tuple spaces. Instead, it can be made as complex as needed in order to uncouple the agent view of the space of tuples from its actual state, and to relate them so as to embed coordination laws into a new observable behaviour of a tuple

<i>Tuple space access and modification</i>	
out_r (T)	succeeds and inserts tuple T into the tuple centre
rd_r (TT)	succeeds, if a tuple T unifying with template TT is found in the tuple centre, by unifying T with TT; fails otherwise
in_r (TT)	succeeds, if a tuple T unifying with template TT is found in the tuple centre, by unifying T with TT and removing T from the tuple centre; fails otherwise
no_r (TT)	succeeds, if no tuple unifying with template TT is found in the tuple centre; fails otherwise
<i>Communication event information</i>	
current_tuple (T)	succeeds, if T unifies with the tuple involved by the current communication event
current_agent (A)	succeeds, if A unifies with the identifier of the agent that triggered the current communication event
current_op (Op)	succeeds, if Op unifies with the descriptor of the operation that produced the current communication event
current_tc (N)	succeeds, if N unifies with the identifier of the tuple centre performing the computation
pre	succeeds in the <i>pre</i> phase of any operation
post	succeeds in the <i>post</i> phase of any operation
success	succeeds in the <i>pre</i> phase of any operation, and in the <i>post</i> phase of any successful operation
failure	succeeds in the <i>post</i> phase of any failed operation

Table 1. Main ReSpecT predicates for reactions.

centre. In particular, since ReSpecT is Turing-equivalent [6], any computable coordination law can be in principle encapsulated into the coordination medium. Two examples of ReSpecT specifications for a LuCe tuple centre are presented in Tables 2 and 3.

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 LP-based approach of LuCe 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.

From a technology viewpoint, the LuCe development framework fully supports the coordination model based on logic tuple centres, and provides for specialised tools for system development, verification, and maintenance. In particular, as more deeply discussed in [8], a tuple centre is characterised by three sets: the set T of its logic 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 logic 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*. Further details on the LuCe can be found in [8].

3 MAS Design with LuCe

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 ReSpecT logic tuples. As a result,

- agents can be designed around their individual tasks, tailoring their interaction protocol to the agents' desired perception of the interaction space, and
- 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 (logic 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 multi-agent system 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. To show how LuCe provides for the above properties, in the rest of this section we present two examples of LuCe-coordinated multi-agent systems, along with two variants.

In Subsection 3.1, we present a tuple centre version of the well-known Dining Philosophers problem [9], aimed at showing (claim *(i)*) how an approach based on tuple centres impacts on the design of the agents and of their interaction

protocol, while ensuring the required overall system properties, such as deadlock avoidance and fairness. Then, we show how a variant of the same problem, with different resources and new concepts, such as different chopstick sets to be used for breakfast, lunch, and dinner, may be implemented (claim (ii)) by simply updating the coordination laws stored in the tuple centre, without affecting agents at all. In Subsection 3.2 we discuss (claim (iii)) how the same tuple centre behaviour specification, expressed in terms of ReSpecT logic tuples, can be re-used within two different multi-agent systems, by sketching the implementation of two different games (TicTacToe and Reversi) governed by the same coordination rules.

These examples have obviously been chosen here for their simplicity. We forward the reader asking for more complex and realistic examples of MAS built around ReSpecT logic tuple centres to [18, 19].

3.1 Dining Philosophers

In the classical Dining Philosopher problem, N philosopher agents share N chopsticks and a spaghetti bowl. Each philosopher needs two chopsticks to eat, but each chopstick is shared by two adjacent philosophers: so, the two chopsticks have to be acquired atomically to avoid deadlock, and released atomically to ensure fairness. Despite of its simplicity, which makes it nearly a toy example, this problem is interesting because it is still a little beyond the basic Linda capabilities: in fact, the above properties cannot be enforced by the coordination model, but must be ensured by properly programming some higher-level protocol on top of it.

First, let us consider the issue of knowledge representation. In a tuple space, chopsticks could be represented either singly (**chop**(i) for the i -th chopstick) or as pairs (**chops**(i, j) for the two adjacent chopsticks i and j). The first choice is the most natural for the domain, but can easily lead to deadlock if a philosopher is not enabled to get atomically the two chopsticks he needs. The second choice solves the deadlock problem, but introduces the problem of ensuring a coherent domain representation: for instance, once **chops**(3, 4) has been taken, also **chops**(2, 3) and **chops**(4, 5) should be no longer available.

The typical way to address this problem is to charge the agents of the coordination load, embedding the required coordination policies *within* the agents, building specialised interaction protocols on top of the basic ones. This solution leads to “coordination-aware” agents, which have to take care of both their tasks and their coordination with other agents. Computation and coordination issues, still separated at the agent language level, are then undesirably merged at the agent design level, making agent design and implementation unnecessarily complex [5, 6]: even the least modification to the system policies implies a re-design both of the agent code and, possibly, of the resource representation.

In the Dining Philosophers case, if chopsticks are represented singly (as **chop**(i) tuples), ensuring deadlock avoidance when using a standard tuple space as a coordination medium requires philosophers to agree on a locking protocol,

```

reaction( out(chops(C1,C2)), ( in_r(chops(C1,C2)),
    out_r(chop(M,C1)), out_r(chop(M,C2))  ) ).
reaction( in(chops(C1,C2)), ( pre, out_r(required(C1,C2))  ) ).
reaction( in(chops(C1,C2)), ( post, in_r(required(C1,C2))  ) ).
reaction( out_r(required(C1,C2)), (
    in_r(chop(C1)), in_r(chop(C2)), out_r(chops(C1,C2))  ) ).
reaction( out_r(chop(C)), ( rd_r(required(C1,C)),
    in_r(chop(C1)), in_r(chop(C)), out_r(chops(C1,C))  ) ).
reaction( out_r(chop(C)), ( rd_r(required(C,C2)),
    in_r(chop(C)), in_r(chop(C2)), out_r(chops(C,C2))  ) ).

```

Table 2. Coordination of the Dining Philosophers.

such as a semaphore tuple to be taken from the tuple space before getting chopsticks, and to be released just after. However, such an approach calls for a global agreement among agents, which does not cope well with the required openness of Internet-based multi-agent systems: it is actually impractical to require agents spread over the network to negotiate such an agreement, or have to embody a sort of “plug-in” subroutine. Moreover, since there is no way to enforce the laws of coordination, there are no means to ensure that a philosopher will always adhere to the required locking protocol: a philosopher agent could try to get chopsticks without synchronising on the semaphore tuple first.

Tuple centres, instead, allow designers to invert the approach (claim (i)): first agent interaction protocols may be defined around the desired agent perception of the communication space, then the most adequate knowledge representation may be chosen, based on the application domain, finally interaction rules are defined so that agents have the desired perception of the domain, based on the actually available knowledge.

In our case, the most natural agents’ viewpoint is to view chopsticks as pairs to be acquired/released, while the application domain suggests that chopsticks are represented singly. Accordingly, a philosopher willing to eat should acquire the chopstick pair he needs by means of a single `in(chops(i, j))` operation, and release it after eating by means of a single `out(chops(i, j))` operation. Of course, the result of such operations should be the atomic removal / insertion of both `chop(i)` and `chop(j)` tuples from the tuple space, transparently to the performing agent. This can be achieved by embedding the required coordination laws into the tuple centre used as the coordination medium, thus also waiving agents from directly handling coordination. The corresponding **ReSpecT** code is reported in Table 2.

Computation and coordination issues are then neatly separated in the design of the philosopher agent, promoting the separation of concerns between the design and implementation of individual agent and agent societies. Individual tasks (like interleaving eating and thinking) are charged upon single agents,

```

reaction( out(chops(C1,C2)), ( in_r(chops(C1,C2)),
    in_r(used(M,C1,C2)),
    out_r(chop(C1)), out_r(chop(C2))  ) ).
reaction( in(chops(C1,C2)), ( pre, out_r(required(C1,C2))  ) ).
reaction( in(chops(C1,C2)), ( post, in_r(required(C1,C2))  ) ).
reaction( out_r(required(C1,C2)), (
    rd_r(timefor(M)), out_r(used(M,C1,C2)),
    in_r(chop(M,C1)), in_r(chop(M,C2)), out_r(chops(C1,C2))  ) ).
reaction( out_r(chop(M,C)), ( rd_r(required(C1,C)),
    rd_r(timefor(M)), out_r(used(M,C1,C)),
    in_r(chop(M,C1)), in_r(chop(M,C)), out_r(chops(C1,C))  ) ).
reaction( out_r(chop(M,C)), ( rd_r(required(C,C2)),
    rd_r(timefor(M)), out_r(used(M,C,C2)),
    in_r(chop(M,C)), in_r(chop(M,C2)), out_r(chops(C,C2))  ) ).

```

Table 3. Coordination of the Dining Philosophers with labelled chopsticks.

while social tasks ensuring global system properties (like deadlock avoidance and fairness) can be embodied into the coordination media, where they conceptually belong, instead of being improperly distributed among all the interacting agents.

Variant. One key feature of the tuple centre approach is that it promotes the incremental design and implementation of coordination policies: in principle, the laws of coordination (and perhaps the knowledge representation adopted, if this is the case) can be modified without affecting the interacting agents at all.

In [5], a variant of the Dining Philosophers problem is introduced, where three different chopstick sets are to be used at breakfast, lunch, and dinner time. Consequently, the resource representation changes so as to embody the meal specification, replacing `chop(i)` tuples with `chop(meal,i)` tuples, where *meal* is either `breakfast`, `lunch`, or `dinner`. Moreover, new knowledge is added to represent the current day time, under the form of a `timefor(meal)` tuple. However, nothing changes for the agents, which can still interact using the very same chopstick acquisition/release defined above, provided that the tuple centre is re-programmed so as to reflect the new desired coordination policy. The corresponding ReSpecT code is reported in Table 3.

3.2 Playing Games

In [8] we discussed the design and development of a multi-agent system in LuCe implementing a distributed, Internet-based version of the TicTacToe game. There, the analysis of the individual tasks led to identify three kinds of agents: *GUI agents*, written in Java, providing human players with a graphic interface to the system, intelligent *expert agents*, written in Prolog, encapsulating knowledge the game rules and strategies, and one *master agent*, which may be written in any language supported by the LuCe system, supervising the system.

Several social tasks were identified, too, and delegated to the `ticTacToe` tuple centre, adopted as the medium for agent interaction. Among the others, the behaviour of `ticTacToe` is designed so as to ensure that

- any human willing to play has an opponent, whether it is another human or an expert agent
- a request for suggestion by a human player is handled transparently to the expert agent, which needs not to implement two different logics and interaction protocols for playing and suggesting
- at any time, there is one expert agent in the system for each open game

Since the corresponding behaviour specification is quite complex, we forward the interested reader to [8]. However, what is relevant here is to understand how such a complex “interaction component” (the logic tuples representing the `ticTacToe` behaviour specification) may be re-used in a different domain, so as to implement the same coordination policies in a different context.

To this end, we selected another game, namely Reversi, which is apparently quite different from TicTacToe:

- the play area is a larger grid (we considered a 4×4 grid for the sake of simplicity)
- the grid state no longer evolves monotonically, since pieces may be flipped as a result of a single move
- the game rules to play, win (and lose...) are completely different - in particular, a single line of pieces of a same colour no longer leads to victory

and so on.

Nevertheless, both individual and social tasks characterising this multi-agent system turned out to be the same as TicTacToe’s. So, the Reversi system still features GUI, expert and master agents, with the same conceptual tasks and interaction protocols, even though their inner logic may be quite different. For instance, the GUI agent has to deal with a larger grid and a new command (the *Pass* one – see Figure 1), while the expert agent obviously incorporates the game rules and strategies for a different game. Instead, the logic of coordination is the same: so, we simply took the `ReSpecT` logic tuples defining the behaviour of the `ticTacToe` tuple centre, and re-used them *as they are*, with no changes, for the specification of the behaviour of the `reversi` tuple centre, exploited here as the core of the Reversi system.

We encourage the interested reader to check the two systems at [7], where the `LuCe` system is also freely available.

4 Conclusions

Several research proposals cited in the paper [22, 1, 2] already exploited logic languages for the coordination of multi-agent systems. The main original contribution of `LuCe` is the notion of logic tuple centre as the core for the coordination of Internet agents, and the corresponding technology for the development,

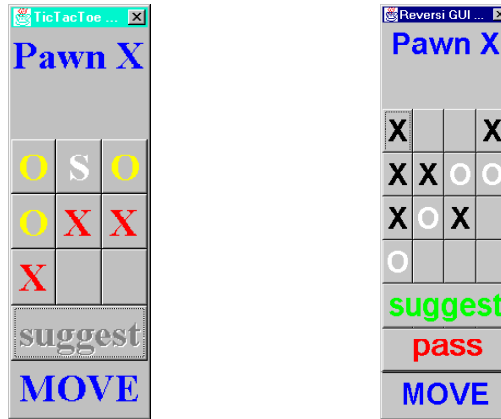


Figure 1. GUI agents for TicTacToe and Reversi

verification and maintenance of multi-agent systems. LuCe logic tuple centres exploit the logic-based ReSpecT specification language not only to enable communication, but also to rule it. This makes it possible to exploit LP also in the coordination of agent societies, and to show its impact on the engineering of social behaviours in multi-agent systems.

References

1. A. Brogi and P. Ciancarini. The concurrent language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1), January 1991.
2. P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12(3):251–284, May 1994.
3. P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2):300–302, June 1996.
4. P. Ciancarini, A. Omicini, and F. Zambonelli. Coordination technologies for internet agents. *Nordic Journal of Computing*, 1999. To appear.
5. E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In D. Garlan and D. Le Métayer, editors, *Coordination Languages and Models – Proceedings of the 2nd International Conference (COORDINATION’97)*, volume 1282 of *LNCS*, pages 274–288, Berlin (D), September 1–3 1997. Springer-Verlag.
6. E. Denti, A. Natali, and A. Omicini. On the expressive power of a language for programming coordination media. In *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC’98)*, pages 169–177, Atlanta (GA), February 27 - March 1 1998. ACM. Track on Coordination Models, Languages and Applications.
7. E. Denti, A. Omicini, and V. Toschi. This paper’s home page. <http://www.lia.deis.unibo.it/Research/D0-MAS99/>.
8. E. Denti, A. Omicini, and V. Toschi. Coordination technology for the development of multi-agent systems on the Web. In E. Lamma and P. Mello, editors, *Proceedings of the 6th AI*IA Congress of the Italian Association for Artificial Intelligence (AI*IA’99)*, pages 29–38, Bologna (I), September 14–17 1999. Pitagora Editrice.

9. E. W. Dijkstra. *Co-operating sequential processes*. Academic Press, London, 1965.
10. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
11. F. Hattori, T. Ohguro, M. Yokoo, S. Matsubara, and S. Yoshida. Socialware: Multiagent systems for supporting network communities. *Communications of the ACM*, 42(3):55–61, March 1999. Special Section on Multiagent Systems on the Net.
12. A. Joshi and M. P. Singh. Multiagent systems on the net. *Communications of the ACM*, 42(3):38–40, March 1999. Special Section – pages 41–77.
13. M. Martelli, V. Mascardi, and F. Zini. Applying logic programming to the specification of complex applications. In M. Falaschi, J. J. Navarro, and A. Policriti, editors, *Proceedings of the APPIA-PRODE-GULP 1997 Joint Conference on Declarative Programming (AGP'97)*, pages 491–499, June 1997.
14. F. G. McCabe. *Logic & Objects*. International Series in Computer Science. Prentice Hall, London (UK), 1992.
15. A. Omicini. On the semantics of tuple-based coordination models. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99)*, pages 175–182, San Antonio (TX), February 28 - March 2 1999. ACM. Track on Coordination Models, Languages and Applications.
16. A. Omicini, E. Denti, and A. Natali. Agent coordination and control through logic theories. In M. Gori and G. Soda, editors, *Topics in Artificial Intelligence – Proceedings of the 4th AI*IA Congress of the Italian Association for Artificial Intelligence (AI*IA'95)*, volume 992 of *LNAI*, pages 439–450, Firenze (I), October 11–13 1995. Springer-Verlag.
17. A. Omicini and A. Natali. Object-oriented computations in logic programming. In M. Tokoro and R. Pareschi, editors, *Object-Oriented Programming – Proceedings of the 8th European Conference (ECOOP'94)*, volume 821 of *LNCS*, pages 194–212, Bologna (I), July 4–8 1994. Springer-Verlag.
18. A. Omicini and F. Zambonelli. Coordination of mobile information agents in TuC-SoN. *Internet Research: Electronic Networking Applications and Policy*, 8(5):400–413, 1998.
19. A. Omicini and F. Zambonelli. Coordination for Internet application development. *Journal of Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999. Special Issue: Coordination Mechanisms for Web Agents.
20. M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):55–61, December 1998.
21. Swedish Institute of Computer Science, Kista, Sweden. *SICStus Prolog Library*, 1998.
22. P. Tarau. Jinni: a lightweight Java-based logic engine for Internet programming. In K. Sagonas, editor, *Proceedings of Post-JICSLP'98 Workshop and 1998 Compulog Net Area Meeting on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*. Manchester (UK), June, 15–19 1998. Invited talk.
23. P. Tarau, V. Dahl, and K. De Bosschere. A Logic Programming infrastructure for remote execution, mobile code and agents. In *Proceedings of the 6th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'97)*, pages 106–112, Boston, June 1997. IEEE Computer Society.
24. P. Wegner. Object-based versus logic programming. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 225–229. ICOT, 1992.
25. M. Woolridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.