Coordination Tools for the Development of Agent-based Systems *

Enrico Denti
DEIS, Università di Bologna
Viale Risorgimento, 2
40136 Bologna, Italy
email: edenti@deis.unibo.it

Andrea Omicini
DEIS, Università di Bologna
Via Rasi e Spinelli, 176
47023 Cesena (FC), Italy
email: aomicini@deis.unibo.it

Alessandro Ricci
DEIS, Università di Bologna
Via Rasi e Spinelli, 176
47023 Cesena (FC), Italy
email: aricci@deis.unibo.it

Abstract

Development and deployment tools are required to effectively support the engineering of multi-agent systems (MAS). In particular, tools are needed to monitor and debug *inter-agents* aspects, such as interaction protocols, coordination policies and environment constraints. By assuming interaction as a first-class issue, this paper aims at identifying the main aspects of support tools for an effective agent infrastructure, and takes tuple-based coordination infrastructures as its reference. The role of support tools is discussed in a simple case study: the development and deployment of a well-known agent interaction protocol, the Contract Net.

1 Coordination Issues in Agent-based Development

A *multiagent system* (MAS) is more than just a sum of individual agents: how agents interact with other agents and the environment, and how they are organised in an ensemble, cannot be reasonably handled as individual issues. In this respect, *interaction* should be considered a fundamental design dimension – the dimension of the social aspects that determine collective behaviour. Interaction is then something that can not be "added on" an already-designed system: instead, it is a critical design issue, to be accounted for since the earliest design phases.

But what does "taking interaction into account" actually means? Just defining and implementing a set of interaction protocols – whatever mechanism is used – is not enough, since nothing would ensure that the agent ensemble behaves as desired, respecting global constraints and ensuring the achievement of the desired global system. Instead, exploiting interaction as a crucial design dimension requires the ability to *govern* it in a precise, well-specified, and controlled way. Yet, mastering interaction is much a less established area [Wegner, 1997].

*This work has been partially supported by MIUR, and by Nokia Research Center, Burlington, MA, USA.

Constraining and governing interaction is precisely the purpose of *coordination* [Gelernter and Carriero, 1992]: in the same way as agent languages and architectures provide abstractions and tools to design and build the single agents, coordination models, languages and architectures provide abstractions and tools to manage and rule the interaction space. Effectively governing interaction basically means the ability to specify and possibly enforce goals, constraints, and desired properties that are not peculiar to a given agent, but to an ensemble of agents, and also the ability to adapt / modify this "social glue" dynamically, at run-time – all, without affecting agent autonomy. While computation languages express the inner working algorithm of an agent, coordination languages express the agent's observable behaviour – what is needed to design its interaction protocol.

Dependencies in MAS can be classified in two types [Schumacher, 2001], *objective* and *subjective*. Objective dependencies refer to *inter-agent* dependencies, and typically concern the configuration of the system in terms of the basic interaction media, agent generation / destruction, and environment organisation; subjective dependencies, instead, refer to the *intra-agent* dependencies towards other agents. The management of subjective dependencies is called *subjective coordination* and is essentially concerned with intra-agent aspects. The management of objective dependencies, instead, is called *objective coordination*, and is essentially concerned with inter-agent aspects, since the dependencies are external to agents. So, in the subjective case, coordination comes from the attitudes of each individual towards the organisations/societies it belongs to, while objective coordination prescinds from the subjective view of the coordinated agents, and promotes instead the separation between the individual perception of coordination and the global coordination issues. As a result, objective coordination enables the modelling and shaping of the interaction space independently of the interacting entities.

However, just providing a suitably-expressive (objective or subjective) coordination model is not enough to effectively support the engineering of Internet-based systems – suitable coordination technologies and run-time support should be made available as well. An efficient design, development and

deployment process requires suitable *infrastructures*, providing designers and developers with a reliable support for commonly-required services: among these, for the above reasons, coordination services should be included as an essential item. To work with the widest range of hardware devices and software platforms, including PDAs and embedded systems, and to support the widest set of applications, a *coordination infrastructure* for agent-based Internet applications should be easily deployable, light-weight, and both statically and dynamically configurable. In addition, it should come along with effective *methodologies* and *tools* supporting the *design*, *development*, and *debugging / monitoring* of interaction.

To be effective, such methodologies and tools must be related to the coordination model embedded in the infrastructure: in particular, the support tools for system development should be inspired by, and modelled after, the metaphors and abstractions introduced by the coordination model – otherwise, the gap between theory and practice would likely make the design, development and debugging task simply too hard and complex. We claim that the definition of such methodologies and tools is a fundamental research issue: it is the researchers' responsibility to study and indicate how the coordination model's concepts and run-time model should transpose onto suitable methodologies and tools, so as to ensure fundamental properties for system development – incremental development, system verification and debugging, system maintenance, etc.

2 Deployment Tools for Interaction

Here we focus on tools and environments for the support of deployment (from debugging to testing, from monitoring to analysis and run-time verification) of the interaction-related aspects of multi-agent systems. While literature on tools and toolkits for the design and development of agent systems and agent infrastructures abounds, very little seems to be available concerning deployment tools [Ndumu *et al.*, 1999]. Perhaps this is due to the fact that (quoting [Van Liederkerke and Avouris, 1995]) *"designing and building MAS is a complex process: the integrated system debugging phase (...) presents particular difficulties (...) from the well-documented limitations that a human faces when observing and monitoring a collection of distributed concurrent processes"* [Garcia-Molina *et al.*, 1994]. Yet, the need for tools supporting the deployment of complex systems is clearly recognised: [Rover *et al.*, 1998] proposes a classification of the most typical tools, categorised in debugging, testing, verification tools, etc.

2.1 Debugging Tools in Concurrent / Parallel Programming

Debugging a MAS is known to be a complex activity, aimed at identifying *structural* and *functional* errors [Ndumu *et al.*, 1999]: structural errors concern the organisation, structure and relationships between agents, while the second class involves errors in the logic of the tasks to be performed. There is, however, one third class of errors, the *coordination errors*, that occur when individual agent behaviour is structurally and functionally correct, but the emergent behaviour of the overall system is incorrect: this is precisely where a suitable coordination infrastructure, providing *ad hoc* tools, can be of help.

Typically, traditional debugging tools in concurrent and distributed programming provide features such as *breakpoints* on the single process, *traps* to catch process exceptions, single-step *tracing*, and explicit read/write of selected memory location or CPU registers; others go further, enabling traps on inter-process communication, the ability to add/remove inter-process messages, and the chance of controlling timeouts [McDowell and Helmbold, 1989; Garcia-Molina *et al.*, 1994]. The most recent and advanced debugging/monitoring tools are event-based [Bates, 1995]: by catching event sequences, they make it possible to post-analyse and post-process event histories, so as to reconstruct the distributed system behaviour.

However, it is known [McDowell and Helmbold, 1989] that observing the execution of concurrent processes is not a neutral operation, since it can interfere with the system behaviour (*Probe Effect*). This problem can be easily understood, since the addition of any spy, print or trace operation alters the timing of a process. So, for instance, critical races, which could be the cause of the system's malfunctioning, and the very reason for debugging, may disappear in the new "debugging-oriented" system configuration [Ndumu *et al.*, 1999].

These problems are due to the lack of separation between computation and interaction: the *state of communication* is not explicitly represented, and so can not be observed without affecting the actors of communication – which is precisely what changes irremediably their previous behaviour. As a general principle, observing a given level with no strong interferences calls for an explicit representation of that level – a *meta level* observer, in fact. This does not mean to avoid interferences at all: it does, however, reduce their impact, since they can occur only at the meta level.

2.2 Mediated Interaction

In mediated interaction models, interaction never occurs directly between interacting entities: communication occurs via some kind of *communication abstraction* explicitly introduced for the purpose of enabling communication. A typical form of mediated interaction is *blackboard*-based interaction: there, interacting entities write/read messages to/from the blackboard only. This approach makes it possible to achieve some fundamental properties, such as *name uncoupling*, *space uncoupling* and *time uncoupling* – that is, interacting entities do not necessarily have to know each other explicitly, nor to be in the same place, not even to co-exist at the same time, in order to communicate. Moreover, the set of messages in the blackboard represents at any time the *state of communication*, which can therefore be quite simply observed.

In tuple-based interaction [Rossi *et al.*, 2001], originated by the Linda model [Gelernter and Carriero, 1992], the communication medium is the *tuple space*, that is, a multi-set of *tuples*, which are ordered collections of data chunks. Agents still communicate by adding (*out*), removing (*in*) and reading (*rd*) tuples to/from the tuple space, but access is associative, based on a pattern-matching (or similar) mechanism.

The existence of explicit communication media makes interaction a first-class issue: from the model to the implementation, interaction is raised to deserve explicit abstractions and places (blackboards or tuple spaces) devoted to it – and to its own handling. This key difference with respect to the previous approaches is what makes the communication state not only explicitly available, but also observable *without affecting* the communicating entities: in fact, since all communication acts occur in given place(s), putting such places under observation requires no change at all on the previous system configuration, avoiding the probe problem *at that level* a-priori.

2.3 Coordination and Tuple Centres

While communication means enabling interaction, coordination means constraining and ruling interaction [Wegner, 1996]. In Linda tuple spaces, the coordination laws are built-in, since the primitive operations are pre-defined, as their semantics. If such laws are inadequate to the application, changing the tuple space behaviour is impossible: the desired overall behaviour must be built upon the fixed tuple space behaviour, charging agents to "fill the gap" – thus violating the separation between computation and coordination.

To overcome this limitation, one must either add new primitives to the basic tuple space model, or change the semantics of the existing operations as needed [Rossi *et al.*, 2001]. *Tuple centres* [Omicini and Denti, 2001b] are tuple spaces whose behaviour can be defined by means of *reactions* to communication events, expressed in the Turing-equivalent ReSpecT language [Omicini and Denti, 2001a]: a tuple centre can then be programmed so as to feature potentially any desired behaviour. Tuple centre behaviour is expressed in terms of ReSpecT *specification tuples*: correspondingly, a tuple centre can be thought as made of two parts – the tuple space, holding ordinary data tuples, and the *specification space*, holding specification tuples. So, both the state of communication and the *state of coordination*, expressed uniformly as logic tuples, are fully observable and modifiable.

Formally, given the set S of specification tuples, a tuple centre state is described at any time by the set T of ordinary tuples, the set W of the currently-pending queries, and the set Z of triggered reactions, waiting to be executed.[1] If the set S is empty, the tuple centre behaviour defaults to the standard tuple space behaviour. These sets provide different relevant views over interaction: in particular, the current communication state is captured by sets T (*data-oriented* view) and W (*control-oriented* view), while the current coordination state is represented by the sets S (static view) and Z (dynamic view) completed with the execution step of the tuple centre virtual machine.

[1] For details on tuple centre and ReSpecT semantics, we forward the interested reader to [Omicini and Denti, 2001b] and [Omicini and Denti, 2001a], respectively.

2.4 Deployment with Mediation

When interaction is mediated by some kind of explicit communication media, observing interaction no longer requires changes to the interacting entities, since the communication state is represented and observable in the communication medium. This chance does not mean, however, that any interference with the system is automatically avoided: in standard tuple spaces, for instance, the insertion of a given tuple can only be observed by explicitly checking this kind of event by having an agent performing a suspensive matching *rd*. So, observation still takes place at the same level where communication occurs: this holds even if the whole tuple set is retrieved via a sort of *rdAll* or *copyAll* bulk primitive, which provides a snapshot, at that given time, of the tuple space state. So, the interaction histories of this communication medium change with respect to the case where the observer agent is not present. Moreover, some communication events may not be observable by definition: in standard tuple spaces, once again, *rd* operations cannot be observed, since they cause no change to the communication state (i.e., to the set of tuples).

Though mediated interaction does not guarantee absence of interferences, yet it is the key to reduce such interferences, bounding them inside an upper (meta) level. In tuple centres, in particular, both the communication and the coordination state are explicitly represented, from the static and the dynamic viewpoints. Since the tuple centre model is *inherently able* to perform operations (reactions) that are *invisible* to the interacting entities, there is a degree of openness that makes it possible to add observation inside the communication media *without affecting the communication level* itself – that is, *(i)* no extra operations must be added in order to achieve observability, and *(ii)* all communication events are *fully* observable, whether they change the tuple space state or not.

This feature provides a fundamental new tool, enabling us to monitor, debug and test interaction in a configurable way. So, it is possible to implement virtually any (computable) debugging policy – possibly, providing some frequent debugging rules as recurring debugging patterns. In particular, since any condition can in principle be intercepted by some suitable reaction, and handled accordingly, tuple centre-based debugging suffers no a-priori limitations. This is a much more flexible approach than many commercial packages': TupleScope by Scientific Computing's Linda implementation (described in Linda user manual), for instance, allows only a pre-defined set of conditions to be captured and monitored – those that TupleScope's designers judged to deserve such "special treatment".

At the meta level, tuple centres endorse a *static* and a *dynamic* observation level. Statically, the set S of specification tuples can be observed, even though with

the same limits discussed above for the tuple space. From the dynamic viewpoint, tuple centres enable in principle the set Z of triggered reactions to be observed, along with the current state of the tuple centre virtual machine. Observation problems have not disappeared, indeed: however, they have moved one level up, to the meta level, where their impact can be better handled.

3 Tools and Technologies

The tuple centre model has been exploited in the TuCSoN [Omicini and Zambonelli, 1999] and LuCe [Denti and Omicini, 2001] coordination infrastructures and related technologies: some IDE tools have been provided to support the deployment of the agent-based applications, engineered upon tuple centre coordination media. The simplest IDE tools provide a GUI to access tuple centres by means of Linda-like communication acts (i.e., coordination primitives) such as *out*, *in*, *rd*, *inp*, and *rdp*. Moreover, the *Inspector*, the most powerful IDE tool, enables the observation, inspection and debugging of the relevant information and activities related to both the tuple centre's communication and coordination states (some views are shown in Figure 1, used for the case study developed in Section 4). Since the (inter-)actions between the Inspector and a tuple centre do not occur at the same level as interactions between a tuple centre and agents, but at the infrastructure (*meta*-)level, the communication state can be observed without incurring in the probe problem.

With respect to the communication state, the Inspector supports the dynamic observation of, and the action on, both the set of tuples and the set of the pending queries of a given tuple centre. By opening the *tuple view*, the set T of the logic tuples stored in the tuple centre (see Subsection 2.3) is displayed: the Inspector provides the means to observe and change such a set, along with facilities to tune inspection according to one's needs. In particular, observation can be both *real-time* and *on demand*: in the first case, each change occurring in the set is immediately notified to Inspectors and shown by the tuple views, while in the second case a snapshot of the tuple set is taken only when explicitly requested via the Inspector's GUI. It is also possible to filter the displayed tuples by specifying a tuple template, as well as to log and save the (possibly filtered) tuple set shown, so as to track the evolution as a sequence of tuple set snapshots. Moreover, the full content of the tuple set can be replaced in one shot, by editing the list of displayed tuples. The Inspector also makes it possible to observe the set W of the pending queries: the *pending query view* displays the list of the pending `in` and `rd` queries, along with related information such as the agent issuing the request, and the (tuple centre local) time when the request itself was received. Filtering and logging of the set content work as above.

The ability to specify the reactive behaviour of the coordination medium can be exploited also to define custom observation and debugging policies in terms of reactions triggered by selected communication events: external communication events are related to agent communication acts, while internal ones are referred to tuple medium's actions on the tuple set. The Turing-equivalence of ReSpecT provides the flexibility to express computations for tracing, monitoring, and debugging purposes, which can be particularly useful when debugging complex, multi-stage interaction protocols involving different agent roles. In these situations, a tuple centre can be programmed so as to trace protocol evolution, making it easier to identify faults, wrong / malicious agents' interaction behaviour, etc. For instance, the following rule monitors the communication acts of agents providing an information (a product price) and generates the related history:

```
reaction( out(price(Info) ), (
  current_agent(Id), current_time(Time),
  out_r(price_history(time(Time),who(Id),content(Info)))).
```

The rule below, instead, handles statistical information about service requests:

```
reaction( in(service(Descr) ),( pre,
   in_r(service_counter(Descr,N)), N1 is N + 1,
   out_r(service_counter(Descr,N1)) )).
```

Such a flexibility can be very useful for infrastructure maintenance, since it simplifies both interaction monitoring and interaction problem detection. Coordination laws can also be exploited to prevent interactions that are invalid / undesired in the current interaction context. For instance, a tuple representing a vital information can be protected from unwanted removal (i.e., from `in` operations) by a reaction like:

```
reaction( in(vital_info(Info) ),( pre,
   out_r(vital_info(Info)) )).
```

In the same way, insertion of critical information could be excluded or just restricted to trusted sources:

```
reaction( out(price(Who, Info) ),(
   no_r(source_authorised(Who)), in_r(price(Who, Info)) )).
```

More generally, although coordination laws can be exploited for security purposes, security issues should better be faced at a different (meta)level (for details please see [Cremonini *et al.*, 1999]).

Inspectors enable the coordination state of a tuple centre to be observed and acted upon, too, by providing access to both the set S of reaction specifications and the set Z of triggered reactions. The *specification view* supports the editing of the ReSpecT code: classical facilities such as syntax highlighting, syntax checking, loading and saving source code, etc. are also provided. In this way, the Inspector can be used to dynamically inspect and change the coordination laws ruling the interactions (of course, the same could be done by means of the infrastructure API). The Inspector's ability to observe and log the dynamic behaviour of the coordination media in terms of triggered reactions is crucial to debug coordination activities: in fact, tracking successful and failed reactions simplifies the understanding of the dynamics of the coordination flow inside the tuple centre.

Finally, the Inspector enables the management of tuple centres as *coordination virtual machines*, providing step-by-step tracing of the machine behaviour, as traditional debuggers do. Since the tuple centre virtual machine manages interactions as first class en-

```
1  out(announcement(Task))
2  wait(ExpireTime)
3  in(bids(Task,BidList))
4  Bid ← selectWinner(BidList)
5  out(award(Task,Bid))
6  in(result(Task,Result)
```

```
1  rd(announcement(Task))
2  MyBid ← evaluate(Task)
3  in(bid(Task,MyBid,Answer))
4  if (Answer==’awarded’) {
5      Result ← perform(Task)
6      out(result(Task,Result)}
```

Table 1: Agent behaviour in the CNP: *Manager* (left) and *Contractor* (right)

```
1 reaction( in(bid(Task,MyBid,Answer )), ( pre,
    out_r(contractor(Task, MyBid)),
    in_r(bids(Task, L)),
    out_r(bids(Task, [MyBid|L])) )).

2 reaction( out(announcement(Task) ), (
    out_r(bids(Task,[])) )).

3 reaction( in(bids(Task,L) ), ( post,
    in_r(announcement(Task)) )).

4 reaction( out(award(Task,TheBid) ), (
    in_r(award(Task,TheBid)),
    in_r(contractor(Task,TheBid)),
    out_r(bid(Task,TheBid,awarded)),
    out_r(refuse_others(Task)) )).

5 reaction( out_r(refuse_others(Task) ), (
    in_r(refuse_others(Task)),
    in_r(contractor(Task,TheBid)),
    out_r(bid(Task,TheBid,’not-awarded’)),
    out_r(refuse_others(Task)) )).
6 reaction( out_r(refuse_others(Task) ), (
     in_r(refuse_others(Task)),
     no_r(contractor(_,_)) )).
```

```
% tracing new contractor
reaction( in(bid(Task,MyBid,Answer) ), (pre,
  current_time(T),
  out_r(history(time(T),
              new_bid_arrived(Task,MyBid))) ))

% tracing manager task announcement
reaction( out(announcement(Task) ), (
  current_time(T),
  out_r(history(time(T),
              new_task_announcement(Task))) )).

% tracing the end of the bidding stage
reaction( in(bids(Task,L) ), ( pre,
  current_time(T),
  out_r(history(time(T),
              task_announcement_expired(Task))) )).

% tracing the announcement of the winner
reaction( out(award(Task,TheBid) ), (
  current_time(T),
  out_r(history(time(T),
              contract_awarded(Task,TheBid))) )).
```

Table 2: ReSpecT coordination rules of the CNP protocol (left); reactions added to provide the interaction history of the CNP run (right)

tities, it is also possible to trap (and trace) instructions involving the acceptance of input communication events, the production of output communication events, the collection of triggered reactions, the execution of reactions, etc. This is an very interesting aspect, to be better explored in the future.

4 Case study

For sake of concreteness, we briefly illustrate the deployment of the Inspector tool for the development and debugging of a simple version of a well-known agent interaction protocol – the Contract Net –, clarifying some of the benefits of the coordination tools. The Contract Net Protocol (CNP) is generally used to solve the so-called *connection problem*: to find the appropriate agent to work on a given task. According to the terminology in [Huhns and Stephens, 1999], *Manager* agents have tasks to be solved, while *Contractor* agents are able to solve tasks. The involved coordination aspects, described as coordination laws of a tuple centre, are expressed by the ReSpecT code shown in Table 2 on the left, while the behaviour of both Managers and Contractors is described in the pseudo-code in Table 1: agent communication acts are put in particular evidence. It should be clear that such code implements a very basic CNP version, whose purpose is just to show how coordination occurs – not to constitute a full-featured implementation of that protocol.

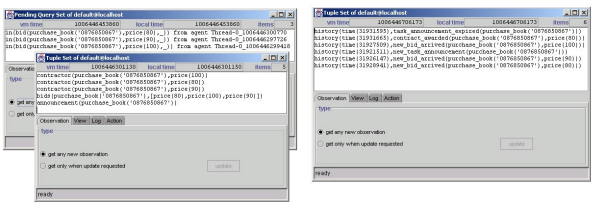

Figure 1: Tuple and pending query views during a CNP run (left), and tuple views at the end of the CNP run (right) in the case with reactions enabling history.

In Table 1, first Manager (left) announces a task to be performed (line 1) and waits for bids: when the related timeout expires, it retrieves the list of bids from potential contractors (line 3), selects the best bid according to its evaluation (line 4), awards the contract (line 5), and receives the result of the performed task (line 6). On the other side, a Contractor (Table 1, right) waiting for possibly-interesting task announcements (line 1) evaluates its capability to respond to this request (line 2), issues its bid, and starts waiting for an answer. If the bid is accepted by the Manager (answer `awarded`), the Contractor performs the task (line 5) and outputs the computed result (line 6). Coordination rules are expressed by the ReSpecT code in Table 2 (left): note that the level of coordination, implemented as tuple centre reactions (based on `in_r`, `out_r`, etc.), is separated from the level of agent interaction acts (performed via `in`, `out`, `rd`, etc.).

When a Manager issues a task announcement, reaction 2 is triggered and coordination data are set up in the tuple centre (Figure 1). Each time a Contractor makes a bid, reaction 1 records the new proposal and updates the bid list. When the Manager eventually collects the bids, reaction 3 removes the task announcement: so, no more bids are considered. Finally, when Manager awards the contract, reaction 4 emits the tuple `bid(Task, TheBid, awarded)` to notify the specific contractor waiting for that answer, then triggers reaction 5 and 6: the first places the `refuse_others` tuple, used to collect and remove the information about other bidders (tuples `contractor`), while reaction 6 notifies the other contractors emitting

the `bid(Task, TheBid, 'not-awarded')` tuple.

To show the power of tuple centre based tools, let us consider now the run-time inspection of the tuple centre during the enactment of the contract net protocol among a set of agents. By the tuple view and the pending query view, it is possible to observe the evolution of the protocol and get information about the involved participants. For instance, the tuple and pending views in Figure 1 (left) show that a task announcement was issued for the assignment of the task `purchase_book(isbn('0876850867'))`, that three bids were proposed, and that the related Contractors are waiting for an answer.

By *incrementally* extending the tuple centre behaviour (Table 2, right), we can document the protocol evolution explicitly, tracing each relevant step and enabling the full history of the protocol interactions to be easily observed. The tuple view in Figure 1 (right) shows the result of a complete run of a CNP, including the above task announcement and bids, as well as the award from the Manager to one of the bidders. Despite its simplicity, the case study puts in evidence the positive impact that tools such as the Inspector could have in managing the complexity of coordination, providing views to observe/act upon agent interactions and means for creating suitable monitoring/debugging policies according to the needs.

5 Lessons Learnt

Several proposals face the issue of suitable tools for the debugging and testing of agent-based systems: yet, rather than focusing on interaction as an independent dimension, most of them try to bring computation-specific metaphors and tools towards interaction. Recent works [Graham *et al.*, 2001] emphasize the need of suitable infrastructure-based tools, providing support for interaction inspection, monitoring and debugging.

One lesson learned is that debugging is much easier when some form of mediated interaction (blackboards, tuple spaces, tuple centres, etc.) is adopted instead of point-to-point communication – whether they are low-level sockets or high-level abstractions such as KQML or FIPA ACL. There are basically two reasons for this: *(i)* the time / space uncoupling properties of generative communication (as in the case of tuple spaces), and *(ii)* the ability to define the behaviour of the communication media (as in the case of tuple centres). Also, mediated interaction enables specific monitoring and debugging tools to be associated to the communication media, thus bringing the metaphors used to model the agent interaction space up to the development and deployment phases.

From the software engineering viewpoint, we could experiment the benefits of the separation between agent computation and coordination. Interaction specifications and protocols can be tested and debugged independently of agents, either using special debugging agents, or simulating them via suitable tools – as our Inspector. In particular, the role of tools seems crucial in helping engineers harnessing the intrinsic complexity of agent interaction, by providing them with the most effective views on the state and evolution over time of interaction within a MAS.

References

[Bates, 1995] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behaviour. *ACM Transaction on Computer Systems*, 13(1):1–31, February 1995.

[Cremonini *et al.*, 1999] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. Multi-agent systems on the Internet: Extending the scope of coordination towards security and topology. In Francisco J. Garijo and Magnus Boman, editors, *Multi-Agent Systems Engineering*, volume 1647 of *LNAI*, pages 77–88. Springer-Verlag, 1999.

[Denti and Omicini, 2001] Enrico Denti and Andrea Omicini. LuCe: A tuple-based coordination infrastructure for Prolog and Java agents. *Autonomous Agents and Multi-Agent Systems*, 4(1/2):139–141, March 2001.

[Garcia-Molina *et al.*, 1994] Hector Garcia-Molina, Frank Germano, Jr, and Walter H. Kohler. Debugging a distributed computing system. *IEEE Transaction on Software Engineering*, 10(2):210–218, March 1994.

[Gelernter and Carriero, 1992] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[Graham *et al.*, 2001] John Graham, Daniel McHugh, Micheal Mersic, McGeary Foster, Victoria Windly, David Cleaver, and Keith Decker. Tools for developing and monitoring agents in distributed muti agent systems. In *Proc. of Autonomous Agents 2000 - Workshop on Infrastructures for Scalable MASs*, 2001.

[Huhns and Stephens, 1999] Micheal Huhns and Larry Stephens. Multiagent systems and societies of agents. In Gerhard Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 79–120. The MIT Press, 1999.

[McDowell and Helmbold, 1989] Charles McDowell and David Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4), December 1989.

[Ndumu *et al.*, 1999] Divine Ndumu, Hyacinth Nwana, Lyndon Lee, and Jaron Collis. Visualizing and debugging distributed multi-agent systems. In *Proceedings of Autonomous Agents 1999*. ACM, 1999.

[Omicini and Denti, 2001a] Andrea Omicini and Enrico Denti. Formal ReSpecT. In Agostino Dovier, Maria Chiara Meo, and Andrea Omicini, editors, *Declarative Programming – Selected Papers from AGP'00*, volume 48 of *Electronic Notes in Theoretical Computer Science*, pages 179–196. Elsevier Science B. V., 2001.

[Omicini and Denti, 2001b] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, November 2001.

[Omicini and Zambonelli, 1999] Andrea Omicini and Franco 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.

[Rossi *et al.*, 2001] Davide Rossi, Giacomo Cabri, and Enrico Denti. Tuple-based technologies for coordination. In Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, editors, *Coordination of Internet Agents*, chapter 4, pages 83–109. Springer-Verlag, March 2001.

[Rover *et al.*, 1998] Diane Rover, Abdul Waheed, Matt Mutka, and Aleksandar Bakic. Software tools for complex distributed systems: Toward integrated tool environments. *IEEE Concurrency*, 6(2):40–54, 1998.

[Schumacher, 2001] Michael Schumacher. *Objective Coordination in Multi-Agent System Engineering – Design and Implementation*, volume 2039 of *LNAI*. Springer-Verlag, April 2001.

[Van Liederkerke and Avouris, 1995] Marc Van Liederkerke and Nicholas Avouris. Debugging multi-agent systems. *Information and Software Technology*, 37(2):103–112, 1995.

[Wegner, 1996] Peter Wegner. Coordination as constrained interaction. In Paolo Ciancarini and Chris Hankin, editors, *Proceedings of the 1st International Conference on Coordination Languages and Models*, volume 1061 of *LNCS*, pages 28–33. Springer-Verlag, April 15–17 1996.

[Wegner, 1997] Peter Wegner. Why interaction is more powerful than computing. *Communications of the ACM*, 40(5):80–91, May 1997.