CE: The Vision for the Future Generation in Research and Applications, R. Jardim-Gonçalves et al. (eds)
© 2003 Swets & Zeitlinger, Lisse, ISBN 90 5809 622 X

Task-oriented engineering of coordinated software systems

Enrico Denti*, Andrea Omicini, Alessandro Ricci
Dept. of Electronics, Computer Science and Systems (DEIS), University of Bologna, Italy

ABSTRACT: In the context of Internet-based applications where heterogeneous, legacy entities should integrate and cooperate, the efficiency of the software production process is a key problem: yet, the classical development cycle and methodologies fall short, calling for *ad hoc* abstractions, methodologies and tools.

In this work, we claim that a task-oriented approach can effectively support the design of highly-interactive applications, enabling even small development teams to afford complex software projects. By explicitly conceiving system engineering in terms of concurrent tasks plus task coordination, this approach promotes the application of innovative management techniques to improve the overall product development process, such as concurrent software engineering.

Coordination models and infrastructures for agent-based systems will then be discussed as suitable means to deliver the full potential of this approach, sketching their support to concurrent software engineering techniques and to the development of a collaborative environment for concurrent engineering.

1 SOFTWARE DESIGN AND DEVELOPMENT IN THE INTERNET ERA

Today, the limited amount of human resources often makes it necessary to charge a project upon few people, causing the traditional roles to overlap: the same persons can be at the same time designers, developers, and testers of their own creations. Also, the same designer can be asked to develop and implement different parts of the same project at the same time: so, following a clean development process (e.g., based on explicit contracts) may just be a dream.

Moreover, in today's Internet-based world even the smallest organisation cannot help but dealing with new kinds of applications, causing a sort of "silent revolution" to software developers, who are asked to operate on highly interactive, inherently-distributed applications, where their experience and known methodologies may fall short.

This is why, apart from new languages and evolving standards, properly handling *interaction* is probably among the most critical issues. In legacy applications, interaction typically assumes well-known, predictable, forms: user interfaces, peripheral I/O, inter-component messages, signals, etc. There, applications are typically designed according to the classical input/compute/output scheme, which is focused on computation: interaction is assumed to occur in pre-defined, selected points – both in terms of specific code points and of time sequence with respect to computation. Switching to a world of dynamically interacting entities (components, objects, agents) spread over the Internet implies several changes to many key issues of system design. Here are some of the questions to be answered:

- How should a single, interactive component be designed? In particular, how should its interaction protocol(s) be defined and tested?
- How can we ensure that the whole system behaves as desired? In particular, how can we get the various interaction protocols to cooperate smoothly towards the application goals?
- How can we enforce the rules coordinating the various components so as to govern the overall system behaviour and achieve the desired global system properties?
- How should we deal with the openness and dynamics of Internet applications? In particular, how can incremental system design and development be enabled and supported, so as to follow the requirement evolution over time?

While the current software development methodologies do offer an effective support to the design of components and to their interoperability via middleware services, just defining and implementing a set of

*Corresponding author. Viale Risorgimento 2, I-40136 Bologna, Italy. Voice +39 051 2093015, Fax +39 051 2093073.

computational components (on the one side) and of interaction protocols (on the other side) – whatever the mechanisms – is not enough: we need abstractions, models, design methodologies, development technologies, support tools and deployment infrastructures enabling any software development team – including small ones – to afford the design of interactive applications in the "globally distributed computing system" that the Internet is becoming to the world scale.

Moreover, the choice of a given software engineering methodology, and its related models and technology, can have a deep impact on the kind of techniques adopted to manage the software development process. For the systems considered here, the inadequacy of the classic water-fall approach and of object-oriented approaches leads to consider Concurrent Software Engineering (CSE) as an appropriate management technique to overcome the intrinsic limitations of the classical development approaches.

CSE aims to reduce time-to-market and improve productivity in product development through simultaneous performance of activities and processing of information (Blackburn, Scudder, and Van Wassenhove 2000). Activity concurrency refers to the tasks that are performed simultaneously by different groups, such as software design, coding, and testing; information concurrency refers to the flow of shared information that supports a team approach to development. So, a key issues of CSE is the coordination of concurrency across all the stages of a project, and even across projects: state-of-the-art approaches in CSE account for providing management frameworks to coordinate the application of concurrency principles throughout the software development process (Blackburn, Scudder, and Van Wassenhove 2000).

Summing up, we would need methodologies, models and technologies that are able both to support the engineering of applications as globally-distributed computing systems, and to promote the adoption of concurrent software engineering management techniques. For this to be possible, we claim that the adopted methodology/model/technology should be based on abstractions providing a high uncoupling degree, so as to promote a software development approach where design, development and testing can be performed as independently from each other as possible, enabling the concurrent application of all the engineering stages. This includes the fundamental ability to deal with interaction as a first-class issue, so that the engineering stages can be concurrently applied to the system glue, too.

2 TASK-ORIENTED SOFTWARE DESIGN

When building multi-component, distributed systems, a fundamental requirement is to *separate control* among the system components. From the software engineering viewpoint, this requirement cannot be fulfilled by just acting at the mechanism level: suitable abstraction are needed to model the control flows and provide a natural way to decouple control.

Thinking in terms of *tasks* to be pursued, instead of control flows to be achieved, provides precisely the above property, since tasks can be seen as the higher-level abstractions that drive the development of control flows, intended as lower-level mechanisms. Moreover, since each component charged of a task is an independent *locus* of control, identifying the tasks that are relevant for a given goal implies separating the control between the corresponding components. This approach reduces the degree of coupling between components, for the control logic for achieving a task is embedded within a single entity: so, each agents interacts with the others only as far as its own task needs to interact with the other tasks.

Since what needs to be coordinated is the mutual dependencies among tasks, the above view naturally reduces the need to coordinate control between components; by doing so, it also inherently decouples software designers from each other. Therefore, a software engineering methodology providing encapsulation of control, so as to identify independent and autonomous components, promotes the concurrent application of the different engineering stages, as required by CSE, task by task.

From the viewpoint of teamwork organisation, the task metaphor allows a software design team to design an interactive system by first identifying the required tasks, and then introducing as many components as needed to ensure that these tasks are carried out. Interestingly, this is well adequate to the cases where the same person is in charge of designing and implementing several components: thinking of tasks instead of control flows (such as which object methods to call, which signals to send, etc.) not only helps reducing the design and development complexity, but also provides the design flexibility that is fundamental for an effective project management (Huhns 2000). If, for instance, a new member joins the team, the task metaphor provides the adequate granularity to re-arrange the development process quickly: a new member can be given one or more tasks, while the tasks of a leaving member can be re-assigned to the other people in the team – all in much a simpler way than sharing and understanding other people's code libraries, objects, etc. Once again, this situation promotes CSE approaches, too, since the same task can be subject of the concurrent and coordinated work of multiple designers, developers and testers.

However, defining and assigning the single tasks to components is only a part of the job: the key issue is how to *design the "glue"* so that they actually behave as an *ensemble*. This is where engineering issues come out. If tasks and components were the only relevant entities, the system would be nothing more than the mere sum of its parts – a multitude of individuals – and interaction, merely rhyming communication, would be just the result of the random interleaving of the observable component behaviour.

But in complex systems, some tasks inevitably need to access distributed and shared resources, involving several components: that is, they naturally endorse a *social* nature. Such *social tasks* cannot, by definition, be assigned to the responsibility of an individual component: instead, their accomplishment requires a novel approach to system analysis, design, and development, that recognises interaction as a fundamental dimension, to be accounted for since the earliest design phases. Thus, suitably engineering interaction becomes a crucial ability to be able to engineer the collective system behaviour. This is perhaps why the design of complex, interactive applications has been considered, sometimes, out-of-reach for small design teams, while this is only the superficial effect of neglecting the true role of the interaction dimension: governing the social tasks & rules.

As a result, our task-oriented approach aims to support CSE not only from the viewpoint of the individual tasks, but also from the viewpoint of the glue – that is, of the social tasks explicitly identified and encapsulated inside such a glue –, making it easier to concurrently apply the design, development and testing activities on them.

3 COORDINATION AS THE SOCIAL DIMENSION OF INTERACTION

The above approach, which concerns all the engineering stages (though in this work we focus on design), accounts for explicitly dealing with individual tasks on the one side, and their coordination as a separate social task on the other side. What we aim to is adopting the same conceptual framework at two different levels: the system engineering level and the software development process management level. At the system engineering level, tasks are the activities that characterise the system being engineered, while at the other level the same tasks are software engineering activities, which can act concurrently on the same (individual or social) task.

3.1 *Coordination for system engineering*

From the first viewpoint, in a context where engineering the glue is at least as relevant as engineering the single agents, interaction cannot be just "added on" an already-designed system: the social behaviour must become the subject of a separate, ad-hoc design phase, which is totally independent from the single agents' design (Ciancarini, Omicini, and Zambonelli 2000). The resulting design process cannot be focused on developing individual agents and making them interact "somehow", but must lead to analyse the system so as to *(i)* identify the individual tasks to be carried out by single agents and the social tasks to be achieved by the mutual agent interaction, and *(ii)* exploit the first ones to drive the development of the single agents, and the second ones to drive the design and development of the agent *interaction space* – that is, agent interaction protocols and rules.

Managing interaction is precisely the purpose of *coordination* (Gelernter and Carriero 1992), which is orthogonal to the computation dimension: while computation focuses on algorithmic aspects (computability, correctness, efficiency, ...), coordination models and languages are aimed to shape the interaction space. In particular, while computation *languages* express the inner algorithm of an agent, coordination *languages* express the agent's observable behaviour – what is needed to design its interaction protocol.

Unfortunately, the separation between computation and coordination is not usually turned into a new approach to system design and engineering, as it should, thus failing to turn the conceptual separation into an effective separation at the analysis, design, and development levels: so, the potential benefits of orthogonality often remain unexploited. Peer-to-peer interaction, for instance, requires that all interacting entities voluntarily agree upon, and respect, some predefined interaction protocols: if an ill-behaved agent does not respect the agreement, the resulting interaction can easily go out of control, because interaction can not be *prescriptive*.

Moreover, peer-to-peer interaction implies that each agent handles its own interaction, thus charging the agent not only of its own task, as it should, but also of some "coordination burden". So, incremental changes over time, possibly to face new requirements or apply bug corrections, are not guaranteed to be restricted to some known boundaries, but may potentially concern many/all agents and several interaction protocols. In particular, changing a coordination law has a potential impact on all involved agents, making the change (unnecessarily) costly and complex.

Mediated interaction models, instead, assume that agents never interact directly, but via some kind of *coordination medium* – a blackboard, a message repository, a tuple space, or a special *middle agent* acting as a "coordinator", charged of the tasks that do not actually belong to any specific agent. In particular, mediated interaction based on explicit coordination media enables – at least in principle – interaction protocols to be engineered *separately from each other*, since no agent is ever involved in other agents's protocols. So, each protocol can evolve – or even be changed at all – with no impact on the others and, therefore, on the corresponding agents. Protocol design and development can then proceed in parallel (different agents and interaction protocols assigned to different people in the development team) or in cascade (different agents and interaction protocols built by the same person, one after the other), naturally fitting both large and small groups, while the bounded impact of evolutionary changes helps to keep the maintenance cost low.

Moreover, since all interaction occurs via the coordination media (whatever they are), mediated interaction *can* be prescriptive, since the coordination medium can be designed so as to enforce the desired interaction rules, preventing agents from violating the agreed protocols and therefore improving the system reliability and robustness. The engineering process, too, can proceed seamlessly from the the analysis and design stage, where the coordination medium is just a design abstraction, to the development and deployment stage, where it becomes a *run-time* abstraction.

As an aside, mediated interaction also brings some notable *agent uncoupling* properties – namely, space uncoupling (agents not necessarily need to be in the same place in order to communicate), name uncoupling (agents not necessarily need to know each other in order to communicate), and time uncoupling (agents may not even need to coexist in time in order to be able to interact) – which are all welcome in an open, unpredictable environment.

As a result, inside a mediated interaction framework, while the individual tasks are naturally mapped onto single agents, the social tasks can be mapped onto the coordination media, provided that these embed enough elaboration capabilities (Omicini and Denti 2001). Coordination rules are then no longer spread among agents, but stored in the place where they conceptually belong. From the software development process, this approach strongly not only reduces the design and development effort, as well as the development time, of an interactive system: it also provides engineers with a precise intervention point to control and tune the system behaviour. Moreover, the single agents and their interaction protocols can now be designed, developed and tested singly, separately from the design, development and test of the social tasks, enforcing design and development locality on a task basis.

3.2 *Coordination for concurrent engineering*

Most of the above-discussed properties for coordination issues apply to software engineering, too, as well as to the management of the software development process. In fact, conceptual frameworks based on mediated interaction can be devised in theories about the management of cooperative work such as Activity Theory and Distributed Cognition (Nardi 1996; Ricci, Omicini, and Denti 2002a) – mainly deployed in CSCW contexts – and about coordination in general (Malone and Crowstone 1994). In particular, according to Activity Theory, the social activities in any complex-enough cooperative context are always mediated by shared *artifacts*, whose role is fundamental to achieve the objectives of the system (society) as a whole. Therefore, the engineering of a (social) system must account for the explicit design, development and testing/maintenance of these coordination artifacts: the previously-discussed coordination media are precisely the coordination artifacts for agent-based societies.

So, our challenge is to apply this conceptual framework to the CSE context, and more generally to the context of Concurrent Engineering (CE) (Reddy, Sriniva, Jagannathan, and Karinthi 1993), which is heavily characterised by the need of coordinating several concurrent heterogeneous activities. Of course, this would require the engineering of suitable coordination artifacts supporting the social activities of Concurrent Engineering.

Since a well-known software engineering principle suggests the use of methodologies, models and technologies whose abstractions are as close as possible to the application domain objects in order to minimise the conceptual gaps, the engineering of effective coordination artifacts supporting CSE and CE environment could benefit from the application of methodologies, models and technologies that are explicitly based on the notions of agent and coordination medium – which is as to say, individual tasks and their coordination.

3.3 *Properties of the coordination artifacts*

In the context of software engineering, which kind of coordination media are best suited to dynamic, open environments? Coordination media designed and developed as middle agents, or as the coordination artifacts that characterise the blackboard-based (tuple space-based) approach?

Assuming the coordination medium to be an agent would bring the advantage of a conceptual and practical uniformity, since all the system entities would share the same nature. But would that really be a benefit? Since an agent is intrinsically pro-active and autonomous, the middle-agent approach brings maximum generality, but also maximum capabilities – indeed, an agent could in principle perform any action. This would call for general, possibly complex tools to manage the wider space of possible agent actions, which also means longer training times for developers, making the approach probably not very adequate to small groups, who cannot afford increasing their work just for the purpose of adopting a big, full-featured (and probably hypertrophic) tool. In one word, this approach could simply be too much for coordination purposes. Not so much paradoxically, a less general (and less powerful) coordination medium – something providing not the full power, but the *required* power – would likely be more adequate to these purposes. However, a typical blackboard or Linda-like tuple space, which might be a good candidate, does not provide the essential feature of letting designers express and embed the desired coordination laws, since interaction always occurs in a fixed, hard-coded way.

So, the coordination medium that is actually needed should be less general, and therefore thinner, than a pro-active agent, so as to be more manageable; yet, it should provide enough expressive power to let designers embed any desired social law in terms of suitable coordination laws. For such purposes, pro-activity is not necessary: what is needed is *re-activity*, embedded inside a *minimal, specialised* entity requiring just simple, easy-to-use (yet expressive) support tools, which can be easily learnt and used by even small development teams.

The last step is to make such coordination rules not hard-coded somehow, but explicitly represented inside that entity in some easy-to-handle form. By guaranteeing this property, the coordination medium becomes also *inspectable*, bringing another fundamental engineering property: the chance to incrementally refine/ specialise the system behaviour by suitably changing (only) the rules that express the social tasks of interest, without affecting the individual agents nor the other social tasks – in the same way as any change to an individual task naturally leads to modifying only the corresponding agent.

4 DESIGNING UPON A COORDINATION INFRASTRUCTURE

In order to discuss the benefits of a task-oriented approach to application analysis, design, and development, we will refer henceforth to coordination infrastructures explicitly built around a mediated coordination model – the *tuple centre* model (Omicini and Denti 2001) – which can effectively support a task-oriented design and development methodology. The LuCe and TuCSoN coordination infrastructures are both *enabling technologies* based on tuple centres: where they differ is in the topological abstractions adopted to map the network topology. Indeed, while LuCe hides the physical location of coordination media, providing for network transparency, TuCSoN is network-aware, letting the physical network structure be perceived and exploited at the agent level – as appropriate to mobile agents[1].

[1]For Further discussion on this difference, readers are referred to (Omicini and Zambonelli 1999). LuCe and TuCSoN are open source technologies, available at lia.deis.unibo.it/ research/

4.1 *Tuple centres: model and tools*

The tuple centre model is an instance of *tuple-based* coordination models (Ciancarini, Omicini, and Zambonelli 1999), characterised fundamentally by *generative communication* – communication data that survive communication acts (Gelernter 1985). As stated above, the consequent agent uncoupling property makes these models particularly suited for unpredictable environments like the Internet, where coupled interaction is hardly feasible.

Here, agents interact by writing, reading, and consuming *tuples* – ordered collections of heterogeneous information chunks – to/from tuple centres by means of simple communication operations (*out, rd, in*) which access tuples associatively. These operations allow agents to operate in a *information-driven* fashion (Papadopoulos and Arbab 1998), based on the availability of selected information represented in the form of tuples in the shared communication space.

Inspired to Linda tuple spaces (Gelernter 1985) with logic tuples, a tuple centre is perceived by agents as a standard tuple space, but may behave differently, thanks to the notion of *behaviour specification*. While the behaviour a tuple space in response to communication events is fixed and pre-defined by the model, the behaviour of a tuple centre can be tailored to the application needs by defining a suitable set of *specification tuples*, which define how a tuple centre should react to incoming/outgoing communication events. In particular, LuCe and TuCSoN adopt ReSpecT tuple centres, where specification tuples are expressed in the logic-based ReSpecT language (Omicini and Denti 2001). The effect of a communication primitive is then no longer limited to adding, reading, or removing a single tuple, but can be made as complex as desired, decoupling the agent view of the tuple space from its actual state, and relating them so as obtain a new observable tuple centre behaviour that embeds (and enforces) the required coordination laws. So, tuple centres can be used to rule inter-agent communication towards the accomplishment of social tasks. Adopting the conceptual framework of activity theory, tuple centres can be framed as general purpose customisable coordination artifacts, whose behaviour can be dynamically specified and adapted to support and automate the co-ordination stage among agents using them ().

In order for tuple centres to be effectively exploitable, however, suitable development tools, specifically modelled after the tuple centre's metaphor and related abstractions, are needed, so that the transition from the design phase to the development and implementation phases can be smooth and straightforward. Infrastructures based on tuple centres provide a complete set of support tools (Denti, Omicini, and Ricci 2002): among these, the *Inspector* enables developers to view, edit and control each tuple centre from its three fundamental viewpoints – the set of tuples, the set of pending queries waiting to be served, and the set of behaviour specifications – thus providing the abstraction levels of *communication* in a *data-oriented* fashion, *communication* in a *control-oriented* fashion, and *coordination*, respectively. Further features include logging the evolution of tuple and pending query sets, saving/restoring a (possibly filtered) view of the tuple set, as well as step-by-step tracing the tuple centre virtual machine (Denti, Omicini, and Ricci 2002).

4.2 *The design process*

The first step consists of analysing the system so as to identify *(i)* the individual tasks to be carried out by single agents, and *(ii)* the social tasks to be achieved by the mutual agent interaction. Individual tasks are then used to drive the development of the single agents, while social tasks drive the design of the agent interaction protocols and coordination rules.

Thanks to the decoupling levels provided by tuple centre coordination, the designer can freely define each agent's interaction protocol as best suited to the agent's specific task, regardless of the interaction protocols used by other agents and of the information representation adopted in the tuple centre: it is up to the coordination laws – defined separately in the development process – to "bridge the gap" between the different agent perceptions and required protocols. On the other hand, the coordination rules can be defined in an information-oriented fashion, too, independently of the single agents' working cycle, based on which information should be produced and when.

In our experience, this feature simplifies both the single agents' development and the test phase of their observable behaviour, since developers can exploit the Inspector to mimic the effect of the missing agents and of the designed (but still unimplemented) tuple centre behaviour. The development of the coordination rules can proceed in parallel, again using the Inspector to mimic missing agents and test the behaviour specification. This approach is well suited to small development teams, too, where the chance to separate the development and test of the single agents, their interaction protocols, the coordination tasks, and knowledge representation should be particularly appreciated. In principle, even one single person could do the job, splitting the project according to the above dimensions and then "putting it all together" later in a natural way – with the Inspector's views over interaction providing support in the case that problems, wrong interactions, or bugs occur. Developers and debuggers geographically separated can use tools such as Inspectors to develop and test concurrently coordination rules of the same running system, embedded in a specific tuple centre or in the set of tuple centres deployed by the distributed application.

5 CONCURRENT ENGINEERING ENVIRONMENTS UPON A COORDINATION INFRASTRUCTURE

Coordination models and infrastructures like TuCSoN can be suitable for engineering collaborative environments supporting concurrent engineering, given their natural support for the coordination of autonomous activities. The ability of dynamically balancing the coordination burden between agents and coordination artifacts makes TuCSoN suitable for designing and developing environments where the traditional automatism of workflow management systems (WfMSs) and the flexibility of computer supported cooperative work (CSCW) need to be dynamically balanced (Ricci, Omicini, and Denti 2002b; Ricci, Omicini, and Denti 2002a), bridging the gap that characterises the single approaches (Schmidt and Simone 2000).

Since task coordination is one of the main activities of any concurrent engineering approach, it is natural to think of environments where suitably-programmed tuple centres act as coordination artifacts to coordinate the activities of designers, developers and testers. The coordination laws embedded in (and enacted by) tuple centres allow interaction to be constrained and dependencies among activities to be managed, according to objectives that belong not to a specific individual task or engineering stage, but to the software development process as a whole. Moreover, a model/infrastructure like TuCSoN also accounts for the coordination of heterogeneous agents, humans – with suitable tool allowing them to act on tuple centres – as well as artificial: these could be, for instance, artificial tester agents, responsible for validating the other agents' observable behaviour or possibly even the coordination rules of the system itself.

6 CONCLUSIONS AND FURTHER WORK

In this paper, we discussed how a task-oriented methodology can impact the analysis, design and development of complex interactive applications, provided that suitable technologies and infrastructures enable the methodology's metaphors and issues – individual tasks, global tasks, coordination laws – to be straightforwardly mapped onto the application components. Moreover, we discussed the suitability of these models and infrastructures for both CSE techniques – to manage the software development process – and the engineering of cooperative environment supporting CSE and CE. Accordingly, future work will include *(i)* empowering the tools provided by our coordination infrastructure so as to improve CSE support, in particular in the case of concurrent development and debugging/testing; *(ii)* extending our exploration in the field of WfMS (Ricci, Omicini, and Denti 2002b) towards the deployment of TuCSoN for the design and development of a collaborative environment that can be suitable for concurrent engineering.

REFERENCES

Blackburn, J., G. Scudder, and L. N. Van Wassenhove (2000, November). Concurrent software development. *Communication of ACM 43*(11es), 200–214.

Ciancarini, P., A. Omicini, and F. Zambonelli (1999, Fall). Coordination technologies for Internet agents. *Nordic Journal of Computing 6*(3), 215–240.

Ciancarini, P., A. Omicini, and F. Zambonelli (2000, February). Multiagent system engineering: the coordination viewpoint. In N. R. Jennings and Y. Lespérance (Eds.), *Intelligent Agents VI – Agent Theories, Architectures, and Languages*, Volume 1767 of *LNAI*, pp. 250–259. Springer-Verlag.

Denti, E., A. Omicini, and A. Ricci (2002). Coordination tools for mas development and deployment. *Applied Artificial Intelligence*, 277–294. To appear.

Gelernter, D. (1985, January). Generative communication in Linda. *ACM Trans. Prog. Languages and Systems 7*(1), 80–112.

Gelernter, D. and N. Carriero (1992, February). Coordination languages and their significance. *Communication of ACM 35*(2), 97–107.

Huhns, M. N. (2000). Agent teams: Building and implementing software. *IEEE Internet Computing 4*(1).

Malone, T. and K. Crowstone (1994). The interdisciplinary study of coordination. *ACM Computing Surveys 26*(1), 87–119.

Nardi, B. A. (1996). *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press.

Omicini, A. and E. Denti (2001, November). From tuple spaces to tuple centres. *Science of Computer Programming 41*(3), 277–294.

Omicini, A. and F. Zambonelli (1999, September). Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems 2*(3), 251–269. Special Issue: Coordination Mechanisms for Web Agents.

Papadopoulos, G. A. and F. Arbab (1998, August). Coordination models and languages. *Advances in Computers 46* (The Engineering of Large Systems), 329–400.

Reddy, R., K. Sriniva, V. Jagannathan, and R. Karinthi (1993, January). Concurrent software engineering: Prospects and pitfall. *IEEE Computer 43*(1), 12–16.

Ricci, A., A. Omicini, and E. Denti (2002a, December). Activity theory as a framework for mas coordination. In P. Petta, R. Tolksdorf, and F. Zambonelli (Eds.), *Engineering Societies in the Agents World III*, LNAI. Springer-Verlag.

Ricci, A., A. Omicini, and E. Denti (2002b, September/December). Virtual enterprises and workflow management as agent coordination issues. *International Journal of Cooperative Information Systems 11*(3/4), 355–380. Cooperative Information Agents: Best Papers of CIA 2001.

Schmidt, K. and C. Simone (2000, May). Mind the gap! towards a unified view of CSCW. In *The Fourth International Conference on the Design of Cooperative Systems COOP 2000*.