

COORDINATION MODELS AND LANGUAGES FOR PARALLEL PROGRAMMING

PAOLO CIANCARINI

University of Bologna, Italy
E-mail: ciancarini@cs.unibo.it

THILO KIELMANN

Vrije Universiteit, Amsterdam, The Netherlands
E-mail: kielmann@cs.vu.nl

Most conventional approaches to parallel programming are based on some basic kinds of synchronized mechanisms and related models of concurrency control: shared variables, message passing, and remote procedure calls. Whereas these paradigms suffice to program parallel applications, they hardly provide abstractions adequate for programmers or language designers. Hence, parallel programming using these models becomes tedious and error-prone. Another issue is that most modern software engineering methods and techniques exploit component-based software architectures usually designed using some object-oriented approaches, which require specific support for concurrency and/or distribution of software components and architectures. Programming languages based on a formally defined *coordination model* and encompassing a notion of autonomous software components, or *agents*, are emerging as an effective paradigm for designing parallel systems. In this paper we describe the basic ideas behind some research in the field of coordination models and languages for designing and controlling the software architecture of parallel applications.

1 Introduction

For several years programming has been synonymous for sequential programming; the very idea of algorithms was based on the concept of instruction sequences. Non-sequential programming became a research topic when system programmers realized that an operating system could be better designed and built as a collection of independent cooperating processes. This idea gave birth to the concept of concurrent programming languages able to express interactions among independent control threads.

With the introduction of distributed systems, the scope of concurrent settings became broader. The absence of shared memory as well as the presence of significant communication latencies, communication errors, and possible system heterogeneity added a lot of complexity to concurrent programming. Traditional concurrent programming languages are based on three basic kinds of mechanisms and corresponding models of concurrent programming: shared variables, message passing, and remote procedure calls. Whereas these paradigms suffice to program distributed systems, they hardly provide adequate abstractions. Hence, programming using these models becomes tedious and error-prone. These problems as well as the requirements of open systems with dynamically changing configurations make programming models

with suitable abstractions inevitable.

Gelernter introduced a fourth basic model called *generative communication*.¹¹ Concurrent languages based on this concept initiated the research area of *coordination*. Today, the interaction between active entities is typically investigated based on the notion of coordination, and by introducing a variety of non-conventional computing models.

This paper presents a perspective on current research issues in coordination models and languages. It has the following structure: Sect.2 introduces some basic concepts concerning coordination. Sect.3 defines our notions of coordination models and languages. Sect.4 puts in a design perspective coordination and software integration. In Sect.5, we discuss some relevant coordination languages. In Sect.6 we draw our conclusions.

2 What is coordination

Coordination as the key concept for modelling concurrent systems is discussed in a wide range of publications. Due to its fundamentality, this notion has a lot of facets it covers. In the scope of this paper we see coordination from the viewpoint of designers of programming models and languages. The following quotations introduce some important issues within this respect.

Here, coordination is:

- *“the additional information processing performed when multiple, connected actors pursue goals that a single actor pursuing the same goals would not perform.”*¹⁶
- *“the integration and harmonious adjustment of individual work efforts towards the accomplishment of a larger goal.”*²⁰
- *“the process of building programs by gluing together active pieces.”*⁶

As can be seen, coordination is concerned with managing the communication which is necessary due to the distributed nature of a system, with the expression of parallel and distributed algorithms, as well as with all aspects of the composition of concurrent systems. In an effort to give a concise definition, we need to define a number of notions:

Agent Agents are active, self-contained entities performing actions on their own behalf.

Action Actions by agents can be divided into two different classes:

1. *Inter-Agent actions.* These actions perform the communication among different agents. They are the subject of coordination models.
2. *Intra-Agent actions.* These are all actions belonging to a single agent, like e.g. internal computations. In the case of specialized interface agents, intra-agent actions may also comprise communication acts by an agent outside the coordination model, like primitive I/O operations or interactions with users.

Configuration We call a collection (or a system of) interacting agents a configuration. Configurations usually have some structure, that is the *software architecture* of the system of agents.

Coordination Coordination is managing the inter-agent activities of agents collected in a configuration.

3 Coordination models and languages

Coordination of agents can be expressed in terms of coordination models and languages. In the following, we try to clarify these two different notions. For coordination models we prefer the following intuitive definition: “A *coordination model is the glue that binds separate activities into an ensemble.*”⁶ In other words, a coordination model provides a framework in which the interaction of individual agents can be expressed. This covers the aspects of creation and destruction of agents, communication among agents, spatial distribution of agents, as well as synchronization and distribution of actions over time.

A more constructive approach to describe coordination models is to identify the components out of which they are built:

1. **Coordination Entities.**
These are the building blocks which are coordinated. Ideally, these are the active agents.
2. **Coordination Media.**
These are the media enabling the communication between the agents. Also, coordination media can serve to aggregate a set of agents to form a configuration.
3. **Coordination Laws.**
These laws describe how agents are coordinated making use of the given coordination media.

A coordination model can be embodied in a (software) coordination architecture or in a coordination language. Examples of coordination architectures are the client-server architecture, the software pipeline, or the blackboard: Software components

are arranged in some special, well-defined structures that enact specific cooperation protocols. Usually parallel programmers implicitly design a software architecture using low-level communication primitives. Due to this rather intuitive approach to software design, there is the need for high-level coordination languages to simplify the implementation of such systems.

A *coordination language* is “*the linguistic embodiment of a coordination model.*”⁶ Thus, a coordination language should orthogonally combine two models: one for coordination (the inter-agent actions) and one for (sequential) computation (the intra-agent actions). The presumably most famous example of a coordination model is the Tuple Space in Linda which encountered several linguistic embodiments like C-Linda or FORTRAN-Linda, both on workstation networks and on massively parallel architectures.

4 Coordination as software integration

Open systems are systems in which new active agents may dynamically join and later leave, i.e. evolving self-organizing systems of interacting intelligent agents.^{2,7} More precisely, open systems can be defined as being composed of software components which are *encapsulated* and *reactive*.²² Components are called encapsulated if they have an interface that hides their implementation from clients; they are called to be reactive if their lifetime is longer than that of the atomic interactions (e.g. messages) which they execute. The fundamental property of open systems is their ability to cope with incremental adaptability, where encapsulation captures spatial incrementality by controlled propagation of local state changes and reactivity enables temporal evolution by incrementally executing interactions.

A related important notion is the one of *open distributed systems*. It is defined in the ISO reference model of open distributed processing (RM-ODP).¹³ In the RM-ODP definition, *distributed systems* have to cope with *remoteness* of components, with *concurrency*, the *lack of a global state*, and *asynchrony* of state changes. In addition, *open distributed systems* are characterized by *heterogeneity* in all parts of the involved systems, *autonomy* of various management or control authorities and organizational entities, *evolution* of the system configuration, and *mobility* of programs and data.

ODP identifies the following properties of systems capable to deal with the above characteristics: *Openness* for new kinds of systems, *integration* of heterogeneous systems, *flexibility* with respect to system evolution, *modularity* of system structure, *federation* for combining separate systems to compound ones, and *transparency* of distribution and heterogeneity.

The requirements of open distributed systems on programming models can be characterized by the notions of heterogeneity in the following senses:

1. Heterogeneity of hardware.

Open distributed systems may consist of computers of various architectures, from different vendors, with different data representations and capabilities. Furthermore, there may be very different interconnection media and topologies, including dial-up lines, different types of local area networks, and even specific interconnections in tightly coupled systems like parallel computers.

As a result, coordination models for open distributed systems cannot rely on specific data representations or communication structures.

2. Heterogeneity of software.

Computers in open distributed systems of course run different operating systems, like MS-DOS, various UNIX-flavours, or some special systems for mainframes, depending on their hardware platform. Also, programming languages in use may vary depending on the purpose of every system in use, ranging from Cobol for business programs, over Fortran in numerical processing to C for text processing or Prolog for knowledge based applications.

Consequently, coordination models trying to integrate such systems cannot be bound to specific language interfaces or communication protocols.

3. Heterogeneity of configurations over time.

Besides the different kinds of the involved systems, the most challenging property of open distributed systems is their dynamic nature. Examples are situations in which additional machines will be brought into the system, dial-up lines connect and disconnect, new machines replace older ones, new operating systems or communication protocols have to be integrated etc.

Thus, agents in an open distributed system must be allowed to appear and disappear completely on their own behalf. This forbids coordination models to rely on specific (central) units or to make use of communication schemes based on static connections or specific identifiers (addresses).

Hence, composition of open distributed systems by modelling the interaction of encapsulated and reactive components, covers concurrency aspects of the more general notion of *software composition* which has recently emerged as an independent research area.¹⁸

4.1 Open object systems

Encapsulation and reactivity as central requirements of agents in open systems directly leads to object-based modelling. Objects are by their very nature open interactive systems. They can not (completely) be described algorithmically because they

interact while computing. Hence, specification of object-based systems is inherently incomplete and hence reflects openness.²³

The RM-ODP model which conceptually provides the basis for commercially available systems uses object-based modelling too; also because of the principal object properties of encapsulation and reactivity. RM-ODP focuses on interaction between objects based on the client/server architecture: "They (objects) embody ideas of services offered by an object to its environments, that is, to other objects."¹³ In RM-ODP, coordination between objects takes place via centralized instances, so called *traders*; which are repositories of service type definitions, used to identify offered and requested services.

Presumably the most prominent commercial system for open, object-based systems is the Common Object Request Broker Architecture (CORBA).¹⁹ Its central component, the Object Request Broker (ORB) acts as a trader in the sense of RM-ODP. Like other traders, the ORB provides references to server objects which in case of dynamically changing configurations may quickly turn into void ("dangling") references causing problems in open configurations.

Another well-known commercial system is Microsoft's *Object Linking and Embedding* (OLE). It is a set of so-called *component objects* for advanced document processing. It is based on the Component Object Model (COM),¹⁷ which provides a binary interface standard between possibly heterogeneous application components. Like with CORBA, new interfaces can be introduced at runtime to a simplified form of a trader, called *component object library* (COL).

Today, client/server architectures are seen as the current intermediate step on the way from mainframe-oriented to collaborative (peer-to-peer) computing.¹⁵ Nevertheless, service-oriented communication is an important paradigm for open distributed systems¹ and must hence be captured by coordination models. But because client/server communication is restricted to the exchange of request/reply pairs, other communication forms like e.g. for group communication can not be modelled adequately. Hence, coordination models for open systems need to be more general in their applicability.

Generative communication as initially introduced in¹¹ is based on an abstractly shared data space, sometimes also called *blackboard*, in which data items can be stored ("generated") and later retrieved. This kind of communication model inherently uncouples communicating agents: a potential reader of some data item does not have to take care about it (e.g. as with rendezvous mechanisms) until it really wants to read it. The reader even does not have to exist at the time of storing. The latter point directly leads to the other major advantage of generative communication: agents are able to communicate although they are anonymous to each other.

This uncoupled and anonymous communication style directly contributes to the design of coordination models for open systems: uncoupled communication enables

to cope with dynamically changing configurations in which agents move or temporarily disappear. Anonymous communication allows to communicate with unknown agents. Hence it allows communication with incomplete knowledge about the system configuration which is a crucial demand of open systems. Due to this fact, coordination models based on generative communication are superior to message passing or trader-based schemes because these both rely on knowledge about a receiver's or server's identification.

Based on this observation, the LAURA model²¹ has been developed in order to introduce generative communication into the RM-ODP model. In LAURA, agents using and offering services share a so-called *service space*. Here, *offer* and *request* forms are matched by LAURA's service type system which replaces RM-ODP's trading function. This model introduces uncoupled and anonymous communication into RM-ODP, but it does not help to overcome the rather restrictive communication scheme of request/reply pairs. Hence, a general-purpose coordination model for open systems needs further improvements.

4.2 Coordination requirements for open object systems

As we have seen so far, open (distributed) systems impose special requirements over coordination models in order to achieve functionality as well as abstractions suited to cope with very large and complex systems. In the following, we investigate properties necessary to fulfil those requirements.

We have identified the following group of properties being *necessary* for building open distributed systems. Primarily, the requirements of heterogeneity and dynamics are reflected here.

Dynamics It is a central requirement of open distributed systems to allow agents to enter or leave a configuration at any time. Hence, coordination laws must not rely on the existence of specific agents.

Decentralization As a consequence of dynamics, it must be possible in a coordination model to specify an agent's behaviour in a separate program. By definition of open distributed systems, there is no overall compile time. Hence, it must be possible to program new agents during the runtime of an already existing system.

This property not only requires a certain degree of dynamics in the coordination model. Moreover, it forces a particular implementation to be "open-ended" in order to allow new agents to enter a running system.

Generativeness With *generative communication* (meanwhile also known as *uncoupled communication*) as introduced so far, all communication is performed

by generating and consuming separate entities, usually contained in a specific computational space. (In the original work,¹¹ the tuple space of Linda served for this purpose.) The existence of separate communication entities enables the modelling of communication in open distributed systems, because it uncouples communication from the existence of specific agents: Communication may simply be performed by generating a communication entity and putting it into the computational space. This allows agents to enter and leave an open system whenever they want without prohibiting their ability to communicate.

Furthermore, generativeness enables reasoning about the communication of a specific configuration, because it is solely based on the state of the communication entities involved in it.

Interoperability The necessity of coordinating agents which are programmed in different languages and operating on different platforms requires that coordination models, and especially their linguistic embodiments, must not rely on properties of specific programming languages or communication media like data types or their representations.

4.3 *Coordination requirements for scalable systems*

By identifying the following group we reflect the aspects of open distributed systems concerning scalability to significantly large sizes. Although such systems could be built with coordination models lacking the properties listed below, they are essential in order to build really large but still maintainable systems.

Homogeneity A model suited for coordinating large systems must be as simple as possible. Hence, all agents should be modelled in a uniform way.

Hierarchical Abstraction For the purpose of really large systems, it is vital to divide the overall configuration into smaller subconfigurations. Hence, it must be possible to treat entire configurations like single agents in a more abstract coordination level. This requirement unifies the notions agent and configuration, enabling agents to be composed out of multiple agents. Additionally, this property enables communication based on groups of agents.

Encapsulation In order to maintain a well-defined, consistent state it must be possible for a configuration to be protected from undesired interactions with agents outside.

Separation of Concerns Inter-agent actions have to be cleanly separated from intra-agent actions in order to distinguish between the concerns of coordination on one hand and of computations on the other.

Rigorous Semantics The model should be based on a rigorous formalism in order to allow reasoning about the interactions to be coordinated. Formal semantics should also offer a basis for automatic optimizer tools, which are able to exploit the different features of different hardware available in heterogeneous systems.

5 Sample coordination languages

We will now present some existing coordination languages. Therefore, we identify their coordination entities, media, and laws. We also discuss their suitability for software integration purposes by evaluating them against the requirements identified in the previous section.

5.1 Linda

The Linda coordination model has been introduced to incorporate the idea of generative communication.¹¹ In Linda, processes (the coordination entities) communicate by writing or reading tuples (in the mathematical sense; consisting of basic data items like numbers and strings) into the so-called “tuple space”. The basic communication acts are creating a tuple (by the *out* operation), and reading or removing tuples from it (by the *read* and *in* operations). Synchronization is performed by letting processes wait until a suitable tuple to be read has been inserted into the tuple space. Furthermore, new processes can be invoked by putting active tuples into the tuple space (by the *eval* operation) which are in turn evaluated. Active tuples produce results in form of passive tuples to which they are converted on termination of their computation.

Typically, in the tuple space the main coordination law defines how tuples are selected to be read from the tuple space itself. The potential reader specifies a template for a tuple he wishes to obtain. The tuple space performs a matching operation in order to find an appropriate tuple. Both tuples and templates may consist of actual fields (values) and formal fields (placeholders for specific data types). A tuple matches a given template if the arities of both correspond and if each actual field matches one of the same type and value or a formal field of the corresponding type.

Although it is not provided by current implementations, the Linda model itself allows dynamic and decentralized concurrent systems. Linda is of course generative and it also enables interoperability on the basis of simple data types being the building blocks for tuples. The separation of concerns between computation and coordination is achieved in an orthogonal way, because the tuple space operations can be added to any given language without interferences with other language properties.

Because Linda fulfils all of these properties, it is a possible model for building open distributed systems. Unfortunately, Linda fails being homogeneous by introducing active and passive tuples, templates, and a tuple space. The original Linda model neither provides hierarchical abstractions nor encapsulation, because there is

only one global tuple space. Hence, distributed systems based on Linda are either open or encapsulated, but not both.

For making Linda completely suitable for integrating open distributed systems, we have recently introduced the Objective Linda coordination model.¹⁴ Here, tuples are replaced by encapsulated objects. Object matching is performed based on objects types and interfaces. Objective Linda provides hierarchies of object spaces and a matching-based mechanism for relating object spaces to each other. A formal semantics for Objective Linda as well as its application to component-based parallel programming has been presented recently.¹²

5.2 ActorSpaces

The actor model unifies objects and concurrency.² Actors are autonomous and concurrently executing objects which operate asynchronously. Actors may send messages to each other. In response to receiving a message, an actor may take the following types of actions: It may send messages to other actors, it may create new actors, and it may specify its own new behaviour (state) to be used when processing the next incoming message.

In the *ActorSpace* model,³ coordination entities are the actors, whereas the coordination laws are stated in terms of pattern-directed message passing: The sender of a message directs it to a set of receiving actors by denoting a pattern which has to be matched by the receivers. So, ActorSpaces serve as the coordination media, acting as passive containers for actors, providing a context for matching patterns on actors and their attributes.

ActorSpaces are dynamic, because the creation of new actors and ActorSpaces is allowed in the model. It is also possible to implement decentralization, because actors and ActorSpaces can be realized as self-contained entities. Unfortunately, ActorSpaces are not generative, because communication is based on messages which have no observable state. This leads to semantical problems with messages sent to actors which might not yet exist or may already have left a system, a phenomenon which has also been described elsewhere.³

The ActorSpace is quite homogeneous, because there is only one kind of actor, whereas communication is always based on terms of ActorSpaces. It is also possible to overlap or even nest different ActorSpaces which in turn enables the design of large systems using hierarchical abstractions. The latter is strengthened because ActorSpaces protect the actors they contain because their visibility to the outside is determined by properties of the ActorSpaces. So, it is possible to achieve encapsulation.

The concerns of computation (the intra-agent actions) and coordination (the inter-agent actions) are separated in a very simple manner: Computation is imple-

mented inside the actors themselves while communication is described in terms of ActorSpaces only. Formal semantics for ActorSpaces have already been developed, so reasoning about coordination in ActorSpaces got a rigorous foundation.

5.3 *Obliq*

Obliq⁵ has been introduced as a scripting extension to Modula 3 rather than as a coordination model. Nevertheless, it has some interesting features which contrast the *ActorSpaces* model introduced above.

In Obliq, multiple sites contain objects and threads as their coordination entities. These sites can communicate over a local or even world-wide network. An Obliq site corresponds to an address space, objects contain data in form of their state, and threads perform communication. Threads and objects can move across sites. Sites communicate via globally known name servers which provide locations for given object names.

Every action performed by an Obliq computation (such as method invocation, delegation, object updating or cloning) is performed on a per-object basis. Obliq provides no abstractions for dealing with groups or sets of objects. Hence, threads (special ones acting as so-called “execution engines”) can be seen as active agents whereas there is not really a notion of configuration: In Obliq only exist local sites and the whole system of participating sites. As a result, it is impossible to build hierarchical abstractions. Furthermore, it is also difficult to identify coordination media; name servers, sites, object interfaces, and communication channels play this role.

Synchronization can be performed on a per-object basis, too. Operations on objects can be synchronized by monitor-like constructs, namely mutual exclusion and condition variables. Deadlocks caused by recurrent method invocations are avoided using the notion of self-inflicted operations which circumvent the object’s mutual-exclusion protection.

Obliq’s coordination laws are based on communication channels which are initially set up using the name servers. This allows new agents to enter an active configuration simply by registering them at a name server. This enables decentralized programming. But as a valid channel is needed for communication, it is impossible for a specific agent to leave a configuration at any time he wants to, so dynamics is prohibited. Additionally, communication in Obliq is not generative, because it is based on remote method invocation. As a consequence, there are problems with agents eventually leaving a configuration.

Obliq isn’t homogeneous, because the notions of passive objects and active threads are separated, causing the necessity of introducing synchronization constructs in the Obliq language. The encapsulation of objects is performed in two interesting ways. On one hand is Obliq’s lexical scope which introduces the necessity of a valid handle

in order to access an object. On the other hand, it is possible to implement encapsulation explicitly by so-called “protected objects”.

Because Obliq computations are performed by sequences of method calls, both local computations and remote invocations are performed in a unified manner. Unfortunately, this prohibits the separation of concerns between computation and coordination purposes. The lexical scope which is basically used by Obliq computations enables to simply introduce distributed semantics based on the notions of sites, locations, values, and threads. But due to the currently experimental status of Obliq, there already are no formally given semantics, yet.

5.4 *Embedding Java in a coordination model*

The Java programming language recently attracted manifold interest as a platform for software integration purposes, especially with open distributed (e.g. Web-based) systems. Although Java includes some mechanisms for concurrency control based on monitors and critical regions, it can be combined with a higher level coordination model to support related component techniques. In fact, we have recently introduced a coordination language kernel called Jada,⁹ which extends Java with Linda-like coordination primitives.

Jada has been introduced to design object-oriented applications distributed over the WWW, however, especially for applications which need support for rule-based coordination, workflows, and logically distributed transactions, Jada is too simple because it is based on singleton tuple transactions, whereas in several cases programmers need to test, delete, or replace multisets of tuples. Several other projects pursue a similar goal using a similar technology.

Interestingly, the idea of combining Java with Linda-like coordination has been pursued by some important software industries. In fact, both Sun’s JavaSpaces¹⁰ and IBM’s TSpaces²⁴ provide mechanisms for distributed persistence and data exchange for code written in Java. Their basic coordination medium consists of flat multiple tuple spaces. In JavaSpaces, a tuple is called an *entry*, defined as typed group of objects. An entry can be written in a JavaSpace, to be later searched or deleted by lookup operations. JavaSpaces support a transaction mechanism to build atomic multiple operations across multiple JavaSpaces. Moreover, active objects can ask a JavaSpace to notify when an entry is written that matches a given template. The actual implementation of JavaSpaces is part of the Jini software architecture.⁴ TSpaces exploit a similar coordination model; however such a system has been developed using a relational database management system as support for tuple spaces, so it is clearly more suited for applications where data management is an important issue.

6 Conclusions

Parallel as well as concurrent programming is typically performed using ad-hoc and low-level constructs like message passing, shared variables, or remote procedure calls. Those constructs hardly provide adequate abstractions, making the construction of large parallel software systems tedious and error-prone. The research area of *coordination* studies interaction between active entities or *agents*, aiming at the provision of suitable models for managing the complexity of such systems.

In this paper we identified and discussed the notion of *coordination* itself, as well as of *coordination models* and *languages*. We investigated the role of coordination as a paradigm for software integration in concurrent systems, focusing on open distributed (e.g. Web-based) systems. Here we discussed the importance of objects and of component-based software architectures, and we identified requirements for coordination models suitable for those systems. Finally, we discussed some coordination models and languages and their suitability for open distributed systems.

References

1. R. M. Adler. Distributed Coordination Models for Client/Server Computing. *IEEE Computer*, 28(4):14–22, 1995.
2. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. M. I. T. Press, Cambridge, Massachusetts, 1986.
3. G. Agha and C. J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. In *Proc. of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, pages 23–32, San Diego, Ca., 1993. Published in SIGPLAN Notices, Vol. 28, No. 7, 1993.
4. K. Arnold, B. O’Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison Wesley, 1999.
5. L. Cardelli. Obliq: A Language with Distributed Scope. Research Report 122, Digital Equipment Corporation, Systems Research Center, 1994.
6. N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.
7. P. Ciancarini. Coordination Languages for Open System Design. In *Proc. of IEEE Intern. Conference on Computer Languages*, New Orleans, 1990.
8. P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models*, number 1061 in Lecture Notes in Computer Science, Cesena, Italy, 1996. Springer. Proc. COORDINATION’96.
9. P. Ciancarini and D. Rossi. Jada: Coordination and Communication for Java agents. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag, Berlin, 1997.

10. E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces, Principles, Patterns, and Practice*. Addison Wesley, 1999.
11. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
12. T. Holvoet and T. Kielmann. Behaviour Specification of Parallel Active Objects. *Parallel Computing*, 24:1107–1135, 1998.
13. ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing. International Standard ISO/IEC 10746–1 to 10746–4, ITU–T Recommendation X.901 to X.904, 1995.
14. T. Kielmann. Designing a Coordination Model for Open Systems. In Ciancarini and Hankin⁸, pages 267 – 284. Proc. COORDINATION’96.
15. T. G. Lewis. Where is Client/Server Software Headed? *IEEE Computer*, 28(4):49–55, 1995.
16. T. Malone and K. Crowstone. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.
17. Microsoft Corporation. The Component Object Model. *Dr. Dobbs Journal*, 12 1994.
18. O. Nierstrasz and D. Tsichritzis, editors. *Object–Oriented Software Composition*. Prentice Hall, 1995.
19. Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG Document Number 93.12.43, 1993.
20. B. Singh. Interconnected Roles (IR): A Coordinated Model. Technical Report CT–84–92, Microelectronics and Computer Technology Corp., Austin, TX, 1992.
21. R. Tolksdorf. Coordinating Services in Open Distributed Systems with LAURA. In Ciancarini and Hankin⁸, pages 386–402. Proc. COORDINATION’96.
22. P. Wegner. Tradeoffs between Reasoning and Modeling. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object–Oriented Programming*, pages 22–41. MIT Press, Cambridge, Mass., 1993.
23. P. Wegner. Interactive Foundations of Object–Based Programming. *IEEE Computer*, 28(10), 1995.
24. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.