Models for coordinating agents: a guided tour

Nadia Busi, Paolo Ciancarini, Roberto Gorrieri, and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
{busi,cianca,gorrieri,zavattar}@cs.unibo.it

Abstract. In this paper we survey and discuss a number of coordination models for agents. We define a framework general enough to be able to capture the main ideas underlying the major coordination models for agents. The framework is based on three key concepts: the *coordinables*, the *coordination medium*, and the *coordination rules*. We start modeling a simple dataspace-based model. Then we structure our discussion along three directions: more advanced coordination primitives exploitable by the coordinables, reshaping the coordination medium, and programming the coordination rules.

1 Introduction and Motivation

Coordination languages are a class of programming notations which offer a solution to the problem of specifying and managing the interactions among computing agents. In fact, they generally offer language mechanisms for composing, configuring, and controlling software architectures made of independent, even distributed, active components.

Gelernter and Carriero introduced a programming-specific meaning of the term *Coordination* presenting the following equation [CG92]:

$$\text{Programming} = \text{Computation} + \text{Coordination}$$

They formulated this equation arguing that there should be a clear separation between the specification of the components of the computation and the specification of their interactions or dependencies. On the one hand, this separation facilitates the reuse of components; on the other hand, the same patterns of interaction usually occur in many different problems – so it might be possible to reuse the coordination specification as well.

A number of interesting models have been proposed and used to design, study, and compare coordination languages. Examples include "tuple spaces" as in Linda [Gel85], various forms of "multiset rewriting" or "chemical reactions" as in Gamma [BL93], and models with explicit support for coordinators as in Manifold [AHS93]. Some of these models have been informally defined, as in the case of Linda. A major aim of this paper is to offer a coherent framework for describing and comparing coordination models for agents.

The relationship between a coordination language and its underlying coordination model is a complex one. Ideally, according to Gelernter, a coordination language "*is the linguistic embodiment*" of a coordination model. Currently, it is more and more clear that a coordination model is similar to an ontology for agent-based software design. The inventor of a coordination model is usually interested in defining a concept of computing agent, that can be mobile, autonomous, or even "intelligent", but in any case has to interact with some environment including some other agents.

Coordination languages are sometimes confused with Linda, especially because it was the first language that got such a classification. Actually coordination languages include notations which have nothing to do with Linda, thus it is difficult to precisely define what a coordination language is. However, the pragmatics of coordination languages shows that:

- A coordination language like Linda has been shown to be sufficiently general to be used for building parallel applications [CG94], for designing distributed computing platforms [WMLF98,Wal99], and for programming agent-based systems [CTV99,COZ99].
- In general, coordination languages are not fully-fledged, general purpose programming languages; rather, they are often defined as language extensions or scripting languages and they are exclusively concerned with coordination issues. In fact, a significant number of coordination models and languages are "minimalist", meaning that they are based on a small set of notions, however powerful enough to deal with the complexities of coordination. An obvious example is Linda, which offers only four coordination primitives. Other less obvious examples include Gamma [BL93], which has only a fixpoint-based coordination construct (called the Gamma operator), and the Interaction Abstract Machines[ACP93], an object-based model for pattern-based, associative coordination of either inter-agent or intra-agent entities.
- Coordination languages are especially relevant in the context of open systems like the Internet and related services, where the coordinated entities and their overall software architectures are not predefined; here they have much in common with object-based approaches. In order to operate in an open system, computing entities must be encapsulated (that is, their implementation details should be hidden from other entities) and they should persist beyond a single transaction. Moreover, in a heterogeneous system, in which the computing entities are written in different languages, the data must be stored in a common format. In the past such considerations have led to the development of object-based modelling techniques; the design of coordination languages is leading to the development of agent-oriented modelling techniques.

The designer of a coordination language has to address a number of issues:

1. *What is being coordinated?*
   The coordinated entities are usually actively computing entities – we usually call them *agents* or *processes*. The agents, or processes, may have been programmed in a variety of different languages. For instance, in the original

Linda system was not difficult to coordinate together both C and FORTRAN processes. However, in Linda coordination primitives were included in the source code of processes. An interesting issue from the viewpoint of agent-oriented programming is that the coordination of the agents should not require any reprogramming of the agents themselves; the inter-agent coordination mechanisms should act possibly as wrappers around the existing, independent agents.

2. *What are the media for coordination?*
Conventional languages for distributed programming, like for instance ADA, usually adopt a naive approach to communication, assuming the existence of channels or ports and introducing some low-level primitives, like `send` and `receive`, to transmit messages over channels. Instead, in many coordination languages, as for instance in Linda, coordination is accomplished via a shared dataspace. In such models, communication is *generative*: agents communicate by "generating" data in the shared space. This data is then available to any agent that has access to the space – this contrasts with the message-passing paradigm where communication is usually a private act between the participating agents, which have to share some channel. Messages in the dataspace can be manipulated, either by the dataspace itself, which in such a case plays a role of active channel, or by third-party agents. The dataspace itself can have a complex structure, for instance in the form of multiple nested dataspaces. Moreover, in the most recent coordination models the dataspace is complemented by event-based mechanisms.
3. *What are the protocols and rules used for coordination?*
By *coordination rules* we intend the laws which rule the relationship between the coordinables and the coordination media. These laws may be expressed in an operational way, for instance introducing some language primitives and their operational semantics, or in a more abstract, declarative way. The Linda proposal identifies a set of coordination primitives which may be used to access a shared dataspace – the primitives are normally implemented as library routines which are called from some host language such as C or FORTRAN. In contrast to Linda, many of the recent proposals have been for rule-based languages; one consequence of this shift to a more declarative view of coordination is an increased reasoning power. In either case the coordination rules provide a level of abstraction which hides much of the complexity of coordination from the programmer.

According to the three above issues, we consider key concepts for the definition of a coordination model the adopted *coordinables*, the considered *coordination medium*, and the corresponding *coordination rules*.

The main contribution of this chapter relies, on the one hand, on the proposal of a formal framework for the representation of the three concepts described above and, on the other hand, on the instantiation of this framework to a variety of coordination models and languages for agents proposed in the literature. The starting point is the basic coordination dataspace-based model of Linda; this model is then extended in three different directions in order to capture the main aspects of many other models. The three directions are briefly described as (i) more advanced coordination primitives exploitable by the coordinables, (ii) reshaping the coordination medium, and (iii) programming the coordination rules.

The chapter has the following structure: Section 2 introduce the formal framework and adopts it for the description of the basic Linda coordination model; Sections 3, 4, and 5 respectively describe the three directions of extension; finally, some conclusive remarks are reported in Section 6.

2 The starting point: the dataspace model for coordinating agents

In this section we start by providing a representation of the *coordinables*, that is of the agents. As our focus is essentially on the description of the coordination capabilities, we will abstract away from the many details of real agents and focus only on the basic features that are relevant from the coordination point of view, i.e., the basic actions that are used for the interaction with the environment where the agent is.

Our starting point is the shared dataspace model, which is at the base of Linda. Agents interact directly with the dataspace (the *coordination medium*) to insert or extract tuples/messages. Asynchronous associative inter-agent communication is realised indirectly: a message (which is a tuple, i.e., an ordered sequence of data) produced by a sender is collected in the dataspace; then a receiver can receive the message by reading or removing it from the dataspace. The access to the messages is associative in the sense that the receiver specifies via a template the kind of tuples in which it is interested, and one tuple matching the template, if available, is selected to be read or consumed from the dataspace.

When a message is emitted by an agent, it has an independent existence in the dataspace until it is explicitly withdrawn by a receiver; in fact, after its insertion in the dataspace, a message becomes equally accessible to all agents and it is bound to none. These are the basic features of the so-called Linda generative communication.

In our presentation we abstract away from the templates and the matching rules between tuples and templates; we consider three basic operations (out, in, rd) managing messages and one ($eval$) for agent generation. The output operation $out(a)$ denotes the insertion of a new instance of message a in the dataspace. The input operation $in(a)$ denotes the consumption of an instance of message a from the dataspace; if a is not available the operation is blocked. The read operation $rd(a)$ is similar to $in(a)$, but a is not removed from the dataspace. The spawn operation $eval(A)$ adds the agent denoted by A to the multiset of currently active agents.

By borrowing typical techniques from the tradition of process calculi for concurrency (e.g., Milner's CCS [Mil89]), an agent is described as a term of an algebra where the basic actions are of the four kinds listed above. To be general, we consider a generic set of messages, called $Data$, ranged over by a, b, ...

The set $Agent$ of agents, ranged over by A, A', ..., is the set of closed terms generated by the following grammar:

$$A \ ::= \ \mathbf{0} \ | \ \mu.A \ | \ \sum_{i \in I} A_i \ | \ K$$
$$\mu \ ::= \ out(a) \ | \ in(a) \ | \ rd(a) \ | \ eval(A)$$

where μ denotes an instance of one of the possible coordination primitives, and K stands for a generic element of a set $Name$ of agent names; we assume that all agent name occurrences are equipped with a corresponding (guarded) defining equation of the form $K = A$. Agent names are used to support recursive definitions as, for example, in the term $Ren_{ab} = in(a).out(b).Ren_{ab}$, which represents an agent able to repeatedly rename messages of the kind a in messages of the kind b.

Agent $\mathbf{0}$ is an agent that can do nothing. Agent $\mu.A$ is an agent that can do the action μ and then behaves like A. Agent $\sum_{i \in I} A_i$ is an agent that can behave like any of the A_i's in the summation (alternative composition).

A system configuration is composed by a multiset of active agents and by a multiset of available messages. Formally, a system configuration is a pair $(Ag, DS) \in \mathcal{M}(Agent) \times \mathcal{M}(Data)$ (where $\mathcal{M}(Set)$ is used to denote the set of the multisets of elements taken from Set). A multiset is usually represented by the classic set notation (using the brackets $\{\,\}$ which we omit in the case of singletons), but where multiplicity of occurrences is relevant. Multiset union is denoted by $\oplus$ (we will use also $\bigoplus_i M_i$ to denote the multiset union of an indexed sequence of multisets). With abuse of notation we sometimes use $\oplus$ also for set union, and the actual meaning is made clear by the context.

A system configuration evolves according to the execution of coordination operations. We formally describe this presenting a transition system defined in a variation of the CHAM style [BB92]; we define a set of rewriting rules on system configurations representing the *coordination rules*. These rules for the basic dataspace model are listed in Table 1 where we use characters in bold to emphasize the elements directly involved in the considered transitions.

The rules are very intuitive. The execution of the output operation has the effect of adding the emitted datum to the dataspace. On the contrary, the input operation removes the datum, if it is present, otherwise the rule does not apply (blocking operation). Similarly for the read operation, where however the datum is not removed. The execution of $eval(A')$ enables a new instance of agent A' to be added to the multiset of agents. The choice among the many A_i's is resolved by taking one that can do a move. Finally, an agent name K can do what its defining agent A can do.

2.1 Remarks

The first observation is that we have not modeled templates and matching rules between tuples and templates, as we have abstracted away from the actual structure of the data. However, this is not a real problem and can be easily accommodated if we extend the modeling with features like value passing in process

$$(\mathbf{out}(\mathbf{a}).\mathbf{A} \oplus Ag, DS) \longrightarrow (\mathbf{A} \oplus Ag, DS \oplus \mathbf{a})$$

$$(\mathbf{in}(\mathbf{a}).\mathbf{A} \oplus Ag, DS \oplus \mathbf{a}) \longrightarrow (\mathbf{A} \oplus Ag, DS)$$

$$(\mathbf{rd}(\mathbf{a}).\mathbf{A} \oplus Ag, DS \oplus \mathbf{a}) \longrightarrow (\mathbf{A} \oplus Ag, DS \oplus \mathbf{a})$$

$$(\mathbf{eval}(\mathbf{A}').\mathbf{A} \oplus Ag, DS) \longrightarrow (\mathbf{A}' \oplus \mathbf{A} \oplus Ag, DS)$$

$$\frac{(\mathbf{A_j}, \mathbf{DS}) \longrightarrow (\mathbf{A}', \mathbf{DS}')}{(\sum_{\mathbf{i} \in \mathbf{I}} \mathbf{A_i} \oplus Ag, \mathbf{DS}) \longrightarrow (\mathbf{A}' \oplus Ag, \mathbf{DS}')} \quad \text{if } j \in I$$

$$\frac{(\mathbf{A}, \mathbf{DS}) \longrightarrow (\mathbf{A}', \mathbf{DS}')}{(\mathbf{K} \oplus Ag, \mathbf{DS}) \longrightarrow (\mathbf{A}' \oplus Ag, \mathbf{DS}')} \quad \text{if } K = A$$

Table 1. The shared dataspace coordination model.

algebras. For a description of the typical Linda matching rules, see [CJY95] (where also different semantics for the basic calculus above are reported, using Petri nets and value passing CCS).

The second observation is that the read operation $rd(a)$ is semantically redundant as it can be equivalently modelled by an input $in(a)$ immediately followed by an output $out(a)$ of the same datum a. This result is not general; for instance, it does not hold when enlarging the basic calculus with additional operators (e.g., the test operators as in Section 3.2), nor when considering semantics that may describe simultaneous executions because in such a case one datum is enough to enable simultaneous read operations but not simultaneous input operations.

The third observation is about the existence of alternative semantics for the output operation. According to [BGZ00a], we can imagine at least three different semantics: *instantaneous*, *ordered* and *unordered*. According to the instantaneous semantics, the configuration $(out(a).A \oplus Ag, DS)$ is actually the same configuration as $(A \oplus Ag, DS \oplus a)$, i.e. the datum a is already in the dataspace. This very abstract semantics is adopted in some process calculi, e.g., the asynchronous π-calculus [HT91]. The ordered semantics is precisely the semantics outlined in Table 1: in one single step, the datum is emitted by the agent and reaches the shared dataspace. The name ordered reflects the fact that the order of emission of data is coherent with the order they reach the dataspace. On the contrary, in the unordered approach, the emission and the insertion of the datum into the dataspace form two separate steps; hence, an emitted datum can reach the dataspace with an unpredictable delay and the order of emission may be not respected by the order of arrival to the dataspace. Formally, this semantics can be described by the following a first rule which substitute the first of Table 1:

$$(\mathbf{out}(\mathbf{a}).\mathbf{A} \oplus Ag, DS) \longrightarrow (\langle\langle \mathbf{a} \rangle\rangle \oplus \mathbf{A} \oplus Ag, DS)$$

where $\langle\langle a \rangle\rangle$ denotes the datum that has been sent but not yet received; the presence of this new term requires the addition of a second rule:

$$(\langle\langle \mathbf{a} \rangle\rangle \oplus Ag, DS) \longrightarrow (Ag, DS \oplus \mathbf{a})$$

Process calculi are usually equipped with observational equivalences; their aim is to define equivalence relations which equate processes which can be considered observationally indistinguishable (see, e.g., bisimulation and weak bisimulation in [Mil89]). Despite of the fact that the three semantics above are very different at the implementation level, it can be proved that they are actually indistinguishable if we consider weak bisimulation. This result does not hold as soon as the test-for-absence operators, discussed in the next section, are introduced.

The last remark is concerned about the expressiveness of the basic calculus above; it is possible to prove that it is not even Turing complete. This result is proved in [BGZ00b], where it is shown how to map agents to finite Place/Transition nets, a formalism in which termination is a decidable property.

3 Extending the coordination primitives

In this section we consider three classes of extra coordination primitives that could be added besides the standard output, input, and read operations.

The first class consists of transaction operations which have the ability to act atomically on a group of data instead of a single data item (e.g., multiset input operations); the second class comprises operations which requires a global vision of the data actually available in the dataspace, in other words, requires a snapshot of the state of the dataspace (e.g., test for absence); the third class consists of operations which comprise both the two abilities above (e.g., the complete consumption of all the data of a certain kind).

3.1 Transaction Operations

One of the main limitations of the basic shared dataspace model is that the considered coordination primitives act atomically on one datum only. A typical problem which arises in this scenario is the realization of transaction operations involving more than one datum. A standard solution to this problem is the adoption of specific transaction protocols which ensure that the execution of the transactions preserves some minimal consistency properties.

This approach for the realization of transactions, for example, has been adopted in JavaSpaces [W$^+$98] where, any time a new transaction is started, a customized *transaction manager* is elected responsible for the preservations of the so called ACID properties:

1. **Atomicity:** This is the typical all-or-nothing condition: all the operations grouped under a transaction occur or none of them do.
2. **Consistency:** The completion of a transaction must leave the system in a consistent state; this is a semantic condition related to what the dataspace is actually representing.

3. **Isolation:** Ongoing transactions should not affect each other; more precisely, participants in a transaction should only see intermediate states resulting from the operations of their own transaction, not the intermediary states of other transactions.
4. **Durability:** The results of a transaction should be as persistent as the entity on which the transaction commits: they should be maintained until other explicit variations are performed.

An alternative approach is to extend the set of the available coordination primitives by, e.g., adding operations which atomically produce and/or consume multisets of data. For example, in TSpaces [WMLF98] the *multiwrite* primitive permits to atomically introduce inside the dataspace all the data contained in an array.

As an example of a transaction primitive we model a $rew(m_1, m_2)$ operation which atomically consumes the multiset of data m_1 and then produces the multiset of data m_2. We do not consider any ability to read multisets of data: indeed, this can be simulated by atomically consuming and producing the data to be tested for presence.

Formally, the syntax is extended by introducing a new prefix:

$$\mu \ ::= \ \ldots \ | \ rew(m_1, m_2)$$

and the semantics by adding the axiom:

$$(\mathbf{rew}(\mathbf{m_1}, \mathbf{m_2}).\mathbf{A} \oplus Ag, DS \oplus \mathbf{m_1}) \longrightarrow (\mathbf{A} \oplus Ag, DS \oplus \mathbf{m_2})$$

The considered $rew(m_1, m_2)$ primitive has been inspired us by a class of coordination languages advocating the so-called chemical reaction metaphor (see, e.g., Gamma [BL93], CHAM [BB92], and LO [AP91]). According to these languages, the items in the dataspace are seen as molecules that freely move in a chemical solution. The molecules react when they come in contact provided that they satisfy certain constraints.

Gamma is one example of coordination language inspired by the chemical metaphor which has been proposed as a means for the high level description of parallel programs with minimum explicit control. The conditional rewriting rules are defined by a pair (R, A), where R is a *reaction condition* (which is a boolean function on multisets of data) and A is a *rewriting action* (which is a function from multisets to multisets of data). When a group of molecules satisfies the reaction condition, then it can be rewritten in the way stated by the corresponding rewriting action.

For instance, a Gamma program which computes the maximum element of a non-empty multiset of integers can be defined as:

$$(R_{max}, A_{max}) \quad \text{with} \quad R_{max}(\{x, y\}) = true \ \text{ and } \ A_{max}(\{x, y\}) = max(x, y)$$

where $max(x, y)$ returns the maximum between x and y. The above program repeatedly compares pairs of numbers, and each time eliminates the smaller one;

the computation terminates when only one number remains in the dataspace (this number is the maximal one).

The Gamma notation can be mapped into our coordination model extended with the $rew(m_1, m_2)$ primitive; given a Gamma program (R, A) it is enough to define an agent $[\![(R, A)]\!]$ which is able to repeatedly perform rewriting operations (defined accordingly to the function A) on those multisets of data which satisfy the reaction condition R:

$$[\![(R, A)]\!] = \sum_{m_i \ s.t.\ R(m_i)=true} rew(m_i, A(m_i)).[\![(R, A)]\!]$$

An interesting observation is related to the atomicity of the rewriting operation $rew(m_1, m_2)$: it is not important that the emission of the multiset of data m_2 is executed atomically with the consumption of m_1. This is a consequence of the fact that the two programs below can be proved to be weak bisimilar:

$$rew(m_1, m_2).A \quad \text{and} \quad rew(m_1, \emptyset).out(a_1).\ldots.out(a_n).A \text{ with } m_2 = \{a_1, \ldots, a_n\}$$

Intuitively, this result holds because it is not possible to observe the delays in the production of data. On the other hand, as we will discuss in the following, test-for-absence operators permit to observes such delays; thus, in the presence of these operators, the above equivalence does not hold any more.

It is interesting to observe that on the contrary the atomicity on the execution of the consumption of the data is important. This problem has been discussed in [Zav00] where the expressiveness of an operation $min(m)$, which is the same of $rew(m, \emptyset)$, is investigated. Indeed, it is proved that it is not possible to provide a general encoding of the $min(m)$ operator using the standard Linda primitives; this allows us to conclude that the min operations strictly increases the expressiveness of the Linda coordination model.

3.2 Global Operations

Here we consider primitives which require, in order to be executed, a global vision of the actual state of the shared repository. Coordination primitives of this class are, e.g., tests-for-absence or operation which count the actual number of instances of a certain kind of datum. Even in this case, TSpaces provides interesting primitives: the *count* operation returns the actual number of data inside the repository which satisfy a certain condition.

A test-for-absence of data verifies that no data of a certain kind are actually available; we can model this operation in our calculus simply by considering a new operation $tfa(a)$, a blocking primitive which can be executed only if no data a are actually available. Formally, we extend the syntax by introducing a new prefix:

$$\mu \ ::= \ \ldots \ | \ tfa(a)$$

and the semantics by adding the axiom:

$$(\mathbf{tfa(a)}.\mathbf{A} \oplus Ag, \mathbf{DS}) \longrightarrow (\mathbf{A} \oplus Ag, \mathbf{DS}) \qquad \text{if } a \notin DS$$

The necessity of a global vision of the actual state of the dataspace is reflected by the side condition which considers the global multiset DS.

Some versions of the Linda coordination language comprises also other two operations which are non-blocking versions of the in and rd operations, called inp and rdp respectively. They are non-blocking as, in the case no data of the required kind is actually available, they can terminate by failing. More precisely, these operations are predicates which may return $true$ or $false$. In the case the required datum is available they behave like in and rd respectively and return $true$. On the other hand, if the datum is not available, they terminate by returning false.

In [BGZ98] the inp and rdp operators have been formalized by using terms with two possible continuations, the first chosen in the case the operation succeeds, the second chosen otherwise. In our setting we can model the same operator by exploiting the $tfa(a)$ primitive:

$$inp(a)?A\_B \;=\; in(a).A + tfa(a).B \qquad rdp(a)?A\_B \;=\; rd(a).A + tfa(a).B$$

In [BGZ00b] the expressiveness of this operation is investigated and it is proved that the simple non-Turing powerful calculus introduced in Table 1 becomes Turing complete when the inp operation is added. Even more interesting is the fact that the Turing-completeness result holds only under the instantaneous and ordered interpretations of the output operation, while it does not hold under the unordered one. Intuitively, the discrimination between the different interpretations of the output operation follows from the fact that a test-for-absence permits to observe the delay between the execution of an output operation and the effective introduction of the emitted datum inside the shared repository.

Other results which does not hold any more under the presence of test-for-absence is the fact that a read operation is the same as an input followed by the output of the same operation, and the observation regarding the fact that atomicity is not needed in the execution of the output part of the $rew(m_1, m_2)$ primitive.

A possible extension of the test-for-absence primitive is to associate to it the execution of output operations. For example, an interesting operation representing a sort of negative test&set operation is analyzed in [Zav98,BGZ00a]; there, it is proved that this kind of operation strictly increases the expressive power of the standard Linda coordination model.

Another interesting remark is that in the presence of primitives which require a global vision, monotonicity is lost. By monotonicity, we mean the fact that the addition of other data inside the shared repository does not alter the previously available computations. Formally, monotonicity holds if the following condition is satisfied for any multiset of data DS'':

$$\text{if } (Ag, DS) \longrightarrow (Ag', DS') \text{ then } (Ag, DS \oplus DS'') \longrightarrow (Ag', DS' \oplus DS'')$$

Apart from a theoretical relevance, monotonicity is a property which is particularly useful in the implementation of the coordination operations, in particular in the case the shared data are distributed across a net and not centralized in a unique storage device. Indeed, if monotonicity is satisfied, the execution of a coordination operation can be safely executed simply by considering the subset of data directly involved in the operation, without having to check the entire distributed state of the repository.

3.3 Global Transaction Operations

We now consider primitives which combine the two abilities described in this section, thus obtaining transaction operations which are able to perform tests on the global state of the shared repository. Typical operations of this class are global read or input primitives which atomically read or consumes all the data satisfying a certain condition.

In [RW98] Rowstron and Wood have proposed a *collect* primitive which removes all the data satisfying a specified pattern. Collect (and its non-destructive counterpart *copy_collect*) is proposed as a solution to the *multiple-rd* problem. of Linda. By multiple-rd, we mean the fact that two distinct read operations may return the same object. Multiple-rd becomes a problem when we need to read all the data satisfying a certain pattern inside a transaction; indeed, in the basic Linda model it is not possible to observe the difference between, on the one hand, the reading of two distinct instances of the same datum and, on the other hand, the double reading of the same instance.

We can extend our formal model with the *collect* primitive simply by extending the syntax with a new prefix:

$$\mu \quad ::= \quad \ldots \mid \mathit{collect}(a)$$

and the operational semantics with the new rule:

$$(\mathbf{collect(a)}.\mathbf{A} \oplus Ag, \mathbf{DS}) \longrightarrow (\mathbf{A} \oplus Ag, \{\mathbf{b} \in \mathbf{DS} \mid \mathbf{b} \neq \mathbf{a}\})$$

4 Reshaping the coordination media

In the previous section we have analyzed and formalized many proposals of extensions of the coordination primitives for the Linda shared dataspace model. Here, we consider extensions devoted to the reshaping of the coordination media, moving from the idea of a single shared dataspace to a collection of either named independent spaces or structured nested spaces.

Several advantages are advocated in order to justify the introduction of multiple spaces. First of all, it provides modularity in the sense that one can restrict the visibility to only data present in a particular dataspace. Moreover, it allows a network-aware style of programming, e.g., simply by allocating spaces to a particular node of the net. Finally, there are proposal for promoting dataspaces to first class citizens of the coordination languages, thus permitting, e.g., to consume, duplicate, and move entire spaces.

In our formal framework we model two different approaches for the introduction of multiple spaces. On the one hand, we consider multiple spaces, each one identified by a unique name as happens for example in Klaim [DFP98]. On the other hand, we consider a hierarchy of anonymous nested spaces as happens for example in Bauhaus Linda [CGZ95]. In the first case names are used in order to specify the space target of a particular coordination primitive; in the second case the specific structure of the dataspace is used instead. More precisely, each process is associated to a dataspace and indicates the target space by specifying its relative location (e.g., the "parent" space, that is the preceding space in the hierarchy).

4.1 Multiple Flat Dataspaces

Let *Space* be a denumerable set of space names, ranged over by $r, s, \ldots$. A system configuration is now a subset of $Conf = Space \times \mathcal{M}(Agent) \times \mathcal{M}(Data)$ with typical element denoted by $s.(Ag, DS)$; this term represents a shared space identified by s, which contains the data in DS, and such that the processes actually associated to it are exactly those in Ag. In the following we use C, D, $\ldots$, to range over system configurations. In order to ensure unique names we will consider only configurations C in which all the space names are distinct, i.e., $s.(Ag, DS) \in C$ and $s.(Ag', DS') \in C$ imply $Ag = Ag'$ and $DS = DS'$.

The syntax of the basic formalism is extended with the new versions of the coordination primitives having the associated space name. The idea is that coordination primitives with a specified space name are executed in the indicated space, while those without name are executed on the local dataspace of the agent.

$$\mu ::= \ldots \mid \ out(a)@s \ \mid \ in(a)@s \ \mid \ rd(a)@s \ \mid \ eval(A)@s \mid \ new(s)$$

A further $new(s)$ primitive is considered which permits the dynamic creation of a new space. The name s is formal, and the actual name is chosen at run-time in such a way that the uniqueness of space names is preserved.

The operational semantics for the new configurations is defined by the new transition system, with typical transition $C \longrightarrow C'$, defined in Table 2 where we do not report the rules for the operations which act on the local dataspace (i.e., without explicit indication of the target space) as they are trivial adaptations of the rules in Table 1.

In this scenario in which processes have a specific location, we could some form of mobility of processes by introducing a primitive $moveto(s)$ which moves the agent to the remote space s. This kind of primitive can be simply implemented exploiting the $eval(A)@s$ primitive which permits to move the code in A to the remote space s:

$$[\![moveto(s).A]\!] \quad = \quad eval(A)@s.\mathbf{0}$$

As already explained the addition of multiple spaces is useful because it permits some form of modularity, a network-aware style of programming, or some form of mobility. Nevertheless, formally speaking, we have that the real novelty is not in the fact that we have spaces with names that we can use in the

$$r.(\mathbf{out}(\mathbf{a})@\mathbf{s}.\mathbf{A} \oplus Ag, DS) \oplus \mathbf{s}.(Ag', DS') \oplus C \;\longrightarrow$$
$$r.(\mathbf{A} \oplus Ag, DS) \oplus \mathbf{s}.(Ag', DS' \oplus \mathbf{a}) \oplus C$$

$$r.(\mathbf{in}(\mathbf{a})@\mathbf{s}.\mathbf{A} \oplus Ag, DS) \oplus \mathbf{s}.(Ag', DS' \oplus \mathbf{a}) \oplus C \;\longrightarrow$$
$$r.(\mathbf{A} \oplus Ag, DS) \oplus \mathbf{s}.(Ag', DS') \oplus C$$

$$r.(\mathbf{rd}(\mathbf{a})@\mathbf{s}.\mathbf{A} \oplus Ag, DS) \oplus \mathbf{s}.(Ag', DS' \oplus \mathbf{a}) \oplus C \;\longrightarrow$$
$$r.(\mathbf{A} \oplus Ag, DS) \oplus \mathbf{s}.(Ag', DS' \oplus \mathbf{a}) \oplus C$$

$$r.(\mathbf{eval}(\mathbf{A}')@\mathbf{s}.\mathbf{A} \oplus Ag, DS) \oplus \mathbf{s}.(Ag', DS') \oplus C \;\longrightarrow$$
$$r.(\mathbf{A} \oplus Ag, DS) \oplus \mathbf{s}.(Ag' \oplus \mathbf{A}', DS') \oplus C$$

$$r.(\mathbf{new}(\mathbf{s}).\mathbf{A} \oplus Ag, DS) \oplus C \;\longrightarrow$$
$$r.(\mathbf{A}\{\mathbf{u}/\mathbf{s}\} \oplus Ag, DS) \oplus \mathbf{u}.(\emptyset, \emptyset) \oplus C \qquad u \text{ is a fresh space name}$$

$$\frac{r.(\mu.\mathbf{A} \oplus Ag, \mathbf{DS}) \oplus C \;\longrightarrow\; r.(\mathbf{Ag}', \mathbf{DS}') \oplus C}{r.(\mu@\mathbf{r}.\mathbf{A} \oplus Ag, \mathbf{DS}) \oplus C \;\longrightarrow\; r.(\mathbf{Ag}', \mathbf{DS}') \oplus C}$$

$$\frac{r.(\mathbf{A_j}, \mathbf{DS}) \oplus \mathbf{C} \;\longrightarrow\; r.(\mathbf{A}', \mathbf{DS}') \oplus \mathbf{C}'}{r.(\sum_{i \in I} \mathbf{A_i} \oplus Ag, \mathbf{DS}) \oplus \mathbf{C} \;\longrightarrow\; r.(\mathbf{A}' \oplus Ag, \mathbf{DS}') \oplus \mathbf{C}'} \quad \text{if } j \in I$$

$$\frac{r.(\mathbf{A}, \mathbf{DS}) \oplus \mathbf{C} \;\longrightarrow\; r.(\mathbf{A}', \mathbf{DS}') \oplus \mathbf{C}'}{r.(\mathbf{K} \oplus Ag, \mathbf{DS}) \oplus \mathbf{C} \;\longrightarrow\; r.(\mathbf{A}' \oplus Ag, \mathbf{DS}') \oplus \mathbf{C}'} \quad \text{if } K = A$$

Table 2. The multiple flat dataspace coordination model.

standard coordination primitives, but in the fact that the structure of the spaces can be dynamically modified by the $new(s)$ primitive. Indeed, in the absence of this operation it is possible to map the new framework with multiple spaces to the previous calculus with a single dataspace. The idea is to associate to each single datum the name of the space where it actually resides. More formally, we consider data taken from $Data \times Space$ and translate, e.g., $in(a)@s.A$ into $in(\langle a, s\rangle).[\![A]\!]$. The complete definition of the mapping is defined in Table 3.

$$
\begin{array}{l}
[\![\bigoplus_i s_i.(Ag_i, DS_i)]\!] = (\bigoplus_i [\![Ag_i]\!]_{s_i}, \bigoplus_i (DS_i \times \{s_i\})) \\
[\![out(a).A]\!]_s = out(\langle a, s\rangle).[\![A]\!]_s \\
[\![in(a).A]\!]_s = in(\langle a, s\rangle).[\![A]\!]_s \\
[\![rd(a).A]\!]_s = rd(\langle a, s\rangle).[\![A]\!]_s \\
[\![eval(A).A']\!]_s = eval([\![A]\!]_s).[\![A']\!]_s \\
[\![out(a)@r.A]\!]_s = out(\langle a, r\rangle).[\![A]\!]_s \\
[\![in(a)@r.A]\!]_s = in(\langle a, r\rangle).[\![A]\!]_s \\
[\![rd(a)@r.A]\!]_s = rd(\langle a, r\rangle).[\![A]\!]_s \\
[\![eval(A)@r.A']\!]_s = eval([\![A]\!]_r).[\![A']\!]_s \\
[\![\sum_{i \in I} A_i]\!]_s = \sum_{i \in I} [\![A_i]\!]_s \\
[\![K]\!]_s = K_s \quad \text{with } K = A \text{ and } K_s = [\![A]\!]_s
\end{array}
$$

Table 3. Encoding multiple flat dataspaces in a single dataspace.

This approach for providing the Linda model with multiple spaces has been adopted in Klaim [DFP98]. Klaim considers both logical and physical names for the spaces and uses two scoping disciplines in order to bind logical to physical names: one static (used in the *out* primitive) and one dynamic (used in the *eval* primitive). Due to the presence of this dynamic scoping the encoding reported in Table 3 cannot be extended to Klaim.

4.2 Multiple Nested Spaces

An alternative approach consists of adopting a hierarchy of spaces structured according to the typical parent/child relationship. This structure permits to avoid the use of space names as identifiers of the space target of a particular coordination primitive.

Many Linda-like languages exploiting a hierarchical structure of nested dataspaces have been proposed, see, e.g., Bauhaus Linda [CGZ95], Melinda [Hup90], and Polis [CFM00]. Even if it is not proposed as a coordination language (but a calculus for mobility) another interesting example of language with nested spaces is Ambient [CG98]. Mobility is achieved by allowing a dynamic reconfiguration of the structure of the spaces by, e.g., permitting to move a space inside another one.

All the above proposals mainly differ in the kind of primitives which permit the inter-space communication or the dynamic modification of the structure of the spaces. As an example of possible inter-space communication mechanism we consider a proposal of [CFM00] to extend the Linda generative communication mechanism to nested dataspaces. The idea is that two brother spaces communicates by introducing and retrieving messages in the parent space. This can be simply achieved by adding the possibility to indicate the parent space as target of the standard Linda primitives. Moreover, we do not consider here mechanisms for the treatment of spaces as first-order objects, that is, they cannot be removed, copied, or moved; they can only be dynamically created. The syntax of the new calculus is obtained by adding the following prefixes:

$$\mu ::= \ldots \mid\ out(a)\uparrow \mid\ in(a)\uparrow \mid\ rd(a)\uparrow \mid\ eval(A)\uparrow \mid\ new(A)$$

The new set of system configuration $Conf$ is inductively defined as:

$$Conf\ ::=\ \mathcal{M}(Agent) \times \mathcal{M}(Data) \times \mathcal{M}(Conf)$$

We omit the third element of the configurations when it is empty, that is (Ag, DS) is a space without subspaces. As an example, we consider the term $(Ag, DS, \{(Ag', DS'), (Ag'', DS'')\})$ as the representation of a configuration with the root space containing the data in DS, the agents in Ag, and two subspaces represented by (Ag', DS') and (Ag'', DS'').

The operational semantics (denoted also in this case with $\longrightarrow$) is defined in Table 4; also here we do not report the rules for the operations acting on the local dataspace as they are trivial adaptations of the rules in Table 1.

Observe that the coordination primitives considered here may act only inside the local or in the parent space. Nevertheless, communication between spaces which are not in a direct parent/child relation may happen via the least common ancestor, provided that each intermediary space contains processes able to move the messages along the considered path, thus moving the messages up and down in the hierarchy from the source to the target space.

As a last remark, we have that an encoding similar to the one of Table 3 can be defined also here to map the hierarchically nested dataspace model to the single dataspace model. Thus, the observations in the previous subsection can be reported also here.

5 Programming the coordination rules

Here, we consider recent extensions which permit a dynamic modification of the coordination rules thus obtaining a programmable coordination medium.

$$(Ag, DS, (\mathbf{out(a)} \uparrow .\mathbf{A} \oplus Ag', DS', S') \oplus S) \longrightarrow$$
$$(Ag, DS \oplus \mathbf{a}, (\mathbf{A} \oplus Ag', DS', S') \oplus S)$$

$$(Ag, DS \oplus \mathbf{a}, (\mathbf{in(a)} \uparrow .\mathbf{A} \oplus Ag', DS', S') \oplus S) \longrightarrow$$
$$(Ag, DS, (\mathbf{A} \oplus Ag', DS', S') \oplus S)$$

$$(Ag, DS \oplus \mathbf{a}, (\mathbf{rd(a)} \uparrow .\mathbf{A} \oplus Ag', DS', S') \oplus S) \longrightarrow$$
$$(Ag, DS \oplus \mathbf{a}, (\mathbf{A} \oplus Ag', DS', S') \oplus S)$$

$$(Ag, DS, (\mathbf{eval(A)} \uparrow .\mathbf{A}' \oplus Ag', DS', S') \oplus S) \longrightarrow$$
$$(\mathbf{A} \oplus Ag, DS, (\mathbf{A}' \oplus Ag', DS', S') \oplus S)$$

$$(\mathbf{new(A)}.\mathbf{B} \oplus Ag, DS, S) \longrightarrow$$
$$(\mathbf{B} \oplus Ag, DS, (\mathbf{A}, \emptyset, \emptyset) \oplus S)$$

$$\frac{\mathbf{C} \longrightarrow \mathbf{C}'}{(Ag, DS, \mathbf{C} \oplus S) \longrightarrow (Ag, DS, \mathbf{C}' \oplus S)}$$

Table 4. The multiple nested dataspace coordination model.

Examples of coordination languages of this class are Law-Governed Linda [ML95], Tucson [OZ98], and Mars [CLZ98]. In all the cases, the introduction of the programmability of the medium is justified as the attempt to enable some form of control of the access to the shared repository. For example, a repository may be programmed in such a way that the no read or input operation are enabled to unknown agents.

We need to revisit our formalism in order to model the dynamical modification of the coordination rules. First of all, some mechanism is required to identify the rules to change. One possible approach is to define a one-to-one mapping between rules and coordination primitives. More precisely, we define Op as the set of all the possible prefixes μ (e.g., $in(a)$ and $out(a)$) and we use the elements in Op as the identifiers for the rules. Let $\mathcal{R} = Op \times \mathcal{M}(Data) \times \mathcal{M}(Agent) \times \mathcal{M}(Data)$ be the set of possible rules (ranged over by R). We will denote elements of $\mathcal{R}$ by $\mu : DS \triangleright [Ag', DS']$ where μ is the identifier of the rule, DS is the part of dataspace which is consumed, Ag' is the set of new spawned agents, and DS' is the part of dataspace created.

As an example, we have that the original single shared dataspace model reported in Table 1 can be described by the following rules:

$$out(a) : \emptyset \triangleright [\emptyset, a]$$
$$in(a) : a \triangleright [\emptyset, \emptyset]$$
$$rd(a) : a \triangleright [\emptyset, a]$$
$$eval(A) : \emptyset \triangleright [A, \emptyset]$$

As the rules may change during the computation, we need to introduce also the set of rules inside the description of the configuration. Formally, a system configuration is a triple (Ag, DS, R) in $\mathcal{M}(Agent) \times \mathcal{M}(Data) \times \mathcal{P}(\mathcal{R})$ where $\mathcal{P}(\mathcal{R})$ denotes the set of all possible sets of rules. We assume that for each operation μ there exists one and only one rule with this identifier.

This kind of system configuration may evolve according to the considered coordination rules. Formally, we need the following rule:

$$\frac{(\mu : \mathbf{DS} \triangleright [\mathbf{Ag}', \mathbf{DS}']) \in R}{(\mu.\mathbf{A} \oplus Ag_c, \mathbf{DS} \oplus DS_c, R) \;\longrightarrow\; (\mathbf{A} \oplus \mathbf{Ag}' \oplus Ag_c, \mathbf{DS}' \oplus DS_c, R)}$$

In this new scenario it is easy to model a primitive $newRule$ responsible for the modification of the set of the actually active rules. First of all we extend the syntax adding the new prefix:

$$\mu ::= \ldots \mid \; newRule(\mu : DS \triangleright [Ag', DS'])$$

and then add the new rule:

$$\begin{array}{l}(\mathbf{newRule}(\mu : \mathbf{DS} \triangleright [\mathbf{Ag}', \mathbf{DS}']).\mathbf{A} \oplus Ag_c, DS_c, (\mu : \mathbf{DS_o} \triangleright [\mathbf{Ag'_o}, \mathbf{DS'_o}]) \oplus R) \\ \;\longrightarrow\; (\mathbf{A} \oplus Ag_c, DS_c, (\mu : \mathbf{DS} \triangleright [\mathbf{Ag}', \mathbf{DS}']) \oplus R)\end{array}$$

5.1 Event notification in the shared dataspace model

Recently, we have assisted to the definition of coordination languages based on the shared dataspace model which consider also mechanisms inspired by the event reaction metaphor. In particular, JavaSpaces and TSpaces allows for the creation, during the computation, of listeners which are able to observe the occurrence of some particular events inside the data repository. The reaction to the event consists of the activation of a process.

In JavaSpaces the observable events are of a specific kind: "introduction inside the data repository of a new object satisfying a certain pattern". The primitive which permits to create a new listener is called $notify$. This new primitive has been formally modeled and analysed in [BZ00b] (see [BGZ00c] for a complete modeling of JavaSpaces).

Even if the concepts of programmable medium and event-notification metaphor have been defined in different contexts with different aims, it is interesting to see that we can model the $notify$ primitive of [BZ00b] in our framework adopting the same approach described above for the modeling of programmable media. We can simply add the following new prefix:

$$\mu ::= \ldots \mid \; notify(a, A)$$

where a is the kind of object which the created listener is interested in, and A is the agent that will be eventually activated as reaction.

The presence of a listener observing the introduction of data of kind a can be realized by modifying the semantics of $out(a)$ in such a way that its execution will activate also all the associated reactions. Formally, we have:

$$(\mathbf{notify}(\mathbf{a}, \mathbf{A}').\mathbf{A} \oplus Ag_c, DS_c, (out(a) : DS \triangleright [Ag', DS']) \oplus R) \longrightarrow$$
$$(\mathbf{A} \oplus Ag_c, DS_c, (out(a) : DS \triangleright [\mathbf{A}' \oplus Ag', DS']) \oplus R)$$

In [BZ00b] the expressiveness of this $notify$ primitive is investigated in the setting of a Linda-based process calculus. Several interesting results are proved there. First of all it is shown that $notify$ strictly increases the expressiveness of a simple Linda-like coordination language with output, input, and read operations only; indeed, there exists no general encoding of $notify$ in terms of these three standard primitives. On the other hand, in the presence of the inp operation it is possible to simulate the $notify$ by activating a special protocol each time an output operation is executed: first the presence of interested listeners is tested, then an activation message is sent to each listener, and finally an acknowledgement from the listeners is required. Another interesting result is that even if $notify$ strictly increases the expressiveness of the basic shared dataspace model, it is still strictly less expressive than the complete Linda model comprising also the inp operation. This because it is not possible to provide a general encoding of inp in terms of output, input, read, and notify operations only.

In [BZ00a] it is shown that the notify primitive makes the unordered semantics, recalled in Section 2, as expressive as the ordered one. The result is achieved by providing an encoding of the calculus interpreted with the ordered semantics on top of the calculus with the unordered one.

6 Conclusion

Coordination languages offer a bridge between concurrency theory and software engineering, as they single out those concurrency principles and primitives that have been more widely accepted by the software engineering community, for their simple understanding, easiness of use and for their support to software reuse and heterogeneity. Indeed, these are the main features explaining the great success of this class of languages.

Even if we expect the readership for this chapter is mainly composed of software engineers, we have presented a panorama of models for coordinating agents in a style that reminds classic concurrency theory. Indeed, we have used notations and techniques similar to those used for process calculi such as CCS. In fact, the theory of coordination can be seen as a new branch of concurrency theory: while in classic process algebras (such as CCS, CSP and related calculi) communication takes place by message passing, by synchronous handshake (point-to- point as in CCS or broadcast as in CSP), in coordination languages on the contrary the emphasis is on the shared communication medium (reminiscent of shared memory systems and related architectures, e.g., blackboard) that implements the Linda-like generative communication mechanism (hence, communication is asynchronous, anonymous and by pattern-matching).

The foundations for this class of languages is still poor, if compared with classic process algebras. For instance, a basic open problem is the definition of suitable behavioural semantic equivalences for the many Linda-like languages discussed here (some initial work is in [BGZ98]). Nonetheless, the problem raised by this class of languages are very challenging as, contrary to many process calculi, coordination languages are widely used in practice and so a good foundational theory for them can really be of practical help, e.g., in devising tools for reasoning about coordination architectures or multiagent systems.

Acknowledgements. This paper has been partially supported by Italian Ministry of University - MURST 40% - Progetti SALADIN and TOSCA, and by a grant by Microsoft Research Europe.

References

[ACP93] JM. Andreoli, P. Ciancarini, and R. Pareschi. Interaction Abstract Machines. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Trends in Object-Based Concurrent Computing*, pages 257–280. MIT Press, Cambridge, MA, 1993.

[AHS93] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency - Practice and Experience*, 5(1):23–70, February 1993.

[AP91] JM. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[BB92] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BGZ98] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[BGZ00a] N. Busi, R. Gorrieri, and G. Zavattaro. Comparing Three Semantics for Linda-like Languages. *Theoretical Computer Science*, to appear, 2000.

[BGZ00b] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156:90–121, 2000.

[BGZ00c] N. Busi, R. Gorrieri, and G. Zavattaro. Process Calculi for Coordination: from Linda to JavaSpaces. In *Proc. of AMAST 2000*, volume to appear of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2000.

[BL93] JP. Banatre and D. LeMetayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, January 1993.

[BZ00a] N. Busi and G. Zavattaro. Event Notification in Data-driven Coordination: Comparing the Ordered and Unordered Approaches. In *Proc. of SAC 2000*, pages 233–239. ACM Press, 2000.

[BZ00b] N. Busi and G. Zavattaro. On the Expressivenes of Event Notification in Data-Driven Coordination Languages. In *Proc. of ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 41–55. Springer-Verlag, Berlin, 2000.

[CFM00] P. Ciancarini, F. Franzé, and C. Mascolo. Using a Coordination Language to Specify and Analyze Systems Containing Mobile Components. *ACM Transactions on Software Engineering and Methodology*, (to appear), 2000.

[CG92] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[CG94] N. Carriero and D. Gelernter. Case Studies in Asynchronous Data Parallelism. *Int. Journal of Parallel Programming*, 22(2):129–149, 1994.

[CG98] L. Cardelli and A. Gordon. Mobile Ambients. In M. Nivat, editor, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), European Joint Conferences on Theory and Practice of Software (ETAPS'98)*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155, Lisbon, Portugal, 1998. Springer-Verlag, Berlin.

[CGZ95] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 66–76. Springer-Verlag, Berlin, 1995.

[CJY95] P. Ciancarini, K. Jensen, and D. Yankelewich. On the Operational Semantics of a Coordination Language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 77–106. Springer-Verlag, Berlin, 1995.

[CLZ98] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 237–248, Stuttgart, Germany, 1998. Springer-Verlag, Berlin.

[COZ99] P. Ciancarini, A. Omicini, and F. Zambonelli. Coordination Technologies for Internet Agents. *Nordic Journal of Computing*, 6(3):215–240, 1999.

[CTV99] P. Ciancarini, R. Tolksdorf, and F. Vitali. The World Wide Web as a Place for Agents. In M. Woolridge and M. Veloso, editors, *Artificial Intelligence Today. Recent Trends and Developments*, volume 1600 of *Lecture Notes in Artificial Intelligence*, pages 175–194. Springer-Verlag, Berlin, 1999.

[DFP98] R. DeNicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[HT91] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Proc. European Conf. on Object Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 141–162. Springer-Verlag, Berlin, 1991.

[Hup90] S. Hupfer. Melinda: Linda with multiple tuple spaces. Technical Report RR YALEU/DCS/R-766, Dept. of Computer Science, Yale University, New Haven, CT, 1990.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[ML95] N. Minsky and J. Leichter. Law-Governed Linda as a Coordination Model. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 125–146. Springer-Verlag, Berlin, 1995.

[OZ98] A. Omicini and F. Zambonelli. Coordination of mobile information agents in tucson. *Journal of Internet Research*, 8(5), 1998.

[RW98] A. Rowstron and A. Wood. Solving the Linda multiple `rd` problem using the `copy-collect` primitive. *Science of Computer Programming*, 31(2-3):335–358, 1998.

[W+98] J. Waldo et al. Javaspace specification - 1.0. Technical report, Sun Microsystems, March 1998.

[Wal99] J. Waldo. The Jini Architecture for Network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[WMLF98] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[Zav98] G. Zavattaro. Towards a Hierarchy of Negative Test Operators for Generative Communication. In *Proc. of Express'98*, volume 16(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

[Zav00] G. Zavattaro. *Coordination Models and Languages: Semantics and Expressiveness*. PhD thesis, Department of Computer Science, University of Bologna, Italy, february 2000.