Models and Technologies for the Coordination of Internet Agents: A Survey

George A. Papadopoulos

Department of Computer Science
University of Cyprus
75 Kallipoleos Street
P. O. Box 20537, CY-1678
Nicosia
CYPRUS
E-mail: george@cs.ucy.ac.cy

Abstract. Agent technology has evolved rapidly over the past few years along a number of dimensions giving rise to numerous “flavours” of agents such as intelligent agents, mobile agents, etc. One of the most attractive and natural fields for the development of agent technology is the Internet with its vast quantity of available information and offered services. In fact, the term “Internet agent” is effectively an umbrella for most of the other types of agents, since Internet agents should enjoy intelligence, mobility, adaptability, etc. All these different types of agents must be able to somehow interact with each other for the purpose of exchanging information, collaborating or managing heterogeneous environments. This survey presents some of the most common models and technologies that offer coordination mechanisms for Internet agents. It argues for the need of using coordination, then it presents some basic infrastructure technologies before examining in more detail particular coordination models for Internet agents, themselves classified into some general categories.

1 Introduction

Agent technology has evolved rapidly over the past few years along a number of dimensions giving rise to numerous “flavours” of agents such as *intelligent agents*, *mobile agents*, etc. One of the most attractive and natural fields for the development of agent technology is the Internet with its vast quantity of available information and offered services. In fact, the term “*Internet agent*” is effectively an umbrella for most of the other types of agents, since Internet agents should enjoy intelligence, mobility, adaptability, etc. All these different types of agents must be able to somehow interact with each other for the purpose of exchanging information, collaborating or managing heterogeneous environments. This survey presents some of the most common models and technologies that offer coordination mechanisms for Internet agents. It argues for the need of using coordination, then it presents some basic infrastructure technologies before examining in more detail particular coordination models for Internet agents, themselves classified into some general categories.

The rest of this chapter is organised as follows. In the rest of this introductory section we give some preliminary information regarding the relationship between Internet agents on the one hand and the notion of coordination on the other. In the process, we identify three main basic areas where the notion of coordination is involved, namely: basic (coordination and communication) infrastructure, coordination platforms (i.e. models and languages that offer coordination functionality as a first class citizen) and "logical" coordination at the level of agent behaviour. The next three sections present in more detail some representative approaches in introducing coordination behaviour into these three levels. The chapter ends with some conclusions and references.

1.1 Internet Agents

The issue of what is precisely a (software) agent is a rather hot topic of discussion and has attracted much controversy. Traditionally, the notion of agents has its roots in areas such as Distributed Artificial Intelligence, Distributed Computing and Human Computer Interaction, as entities that enjoy such properties as proactivity, reactivity, autonomous behaviour or adaptability. Thus, it is often taken as a default that an agent is, in general, intelligent and a non-intelligent agent is, in a way, a contradiction in terms. However, many people have questioned this approach (see for instance the discussion in \{Petrie} where it is claimed that an agent can be autonomous without being also intelligent and, in fact, intelligence is not always necessarily a useful property); another related discussion can be found in \{Flores-Mendez}. The rapid growth of the Internet has further complicated this issue, where now in addition to being intelligent an Internet agent is also expected to be mobile.

Thus, in this chapter we will refrain from adhering to a particular definition or assume that an Internet agent has some specific properties. In fact, for the purposes of this work we do not have to do that since all definitions of what an agent (Internet or otherwise) is, agree that an agent should be able to: (i) *communicate* with other agents, and (ii) *cooperate* with other agents. In particular, an agent should be able to engage in, possibly, complex communications with other agents in order to exchange information or ask their help in pursuing a goal. The latter leads naturally to the notion of a number of agents cooperating with each other towards the accomplishment of some common objective. The need to communicate and cooperate leads to the need for coordinating the activities pursued by agents in order to both simplify the process of building multi-agent systems but also provide the ability to reuse descriptions of coordination mechanisms and patterns.

1.2 Internet, WWW and Coordination

Recently, there has been an increase in the development of applications that need to cooperate, coordinate and share their information with other applications, either with or without user intervention. This is particularly true for Web-based applications that operate in an open system environment and where data and resources are distributed. This leads to the need for developing techniques that allow negotiation and cooperation. In particular, it has been suggested that basic services for collaboration that include the coordination of activities and the exchange of information should be provided by the Web infrastructure and related enabling technologies. More to the point, some approaches refer to the development of infrastructures for sharing artifacts, the use of shared languages for exchanging information, and the creation of shared working spaces for providing collaboration (\{Kotsis}). This leads to the notion of having coordination architectures for the purpose of building collaborative applications.

1.3 Coordination and Internet Agents

Coordination has been defined as the process of managing dependencies between activities. In a seminal paper, Crowston and Malone (\{Malone}) characterise coordination as an emerging research area with an interdisciplinary focus, playing a key issue in many diverse disciplines such as economics and operational research, organisation theory and biology. In the field of Computer Science coordination is often defined as the process of separating computation from communication concerns and a number of coordination models and languages have been developed (\{Papadopoulos}).

From the discussion so far it has become clear that the need for coordinating activities is inherent in both the case of building (multi-) agent systems and in the case of developing Web-based environments, independently from each other. Therefore in the case of Internet agents which combine the notion of multi-agent collaboration with that of using a Web-based environment, the development of suitable coordination technologies is of paramount importance.

In the rest of this chapter we present some representative approaches in developing such coordination frameworks. However, before we embark on this era, we would like to put the rest of the work into some perspective and we argue that one way to classify these approaches is to group them into three categories as follows:

- *Basic coordination infrastructure*. The most primitive form of coordination is that of communication. In this first, "lower", level of coordination formalisms we present the elementary enabling technologies for building coordination frameworks. We can identify two such groups of enabling technologies: various families of Agent Communication Languages (ACLs) such as KQML and its variants, and support computing technologies that act as compositional platforms for multi-agent systems executing in a distributed environment.
- *Coordination frameworks*. In this second, "middleware", level we examine some representative frameworks that offer mechanisms for modeling coordination activities and expressing them as first class citizens. At this level we have the traditional approach to developing coordination models and languages, with emphasis on issues particularly pertaining to Internet agents.
- *Logical coordination*. In this third, "upper", or user-level we present some approaches which deal with the coordination functionality of the agents themselves such as contracting, planning or negotiation. Such coordination agents include cooperation domain agents, interface agents, and collaborative agents.

Thus, we approach the issue of coordination in Internet agents from a more general perspective examining not only the mainstream coordination notions (the second level) but also the issues of information exchange (the first level) and managing interdependencies between agents (the third level). In the next three sections we elaborate further on these three dimensions of coordination.

2 Basic Coordination Infrastructure

2.1 Agent Communication Languages

Aside from how one perceives the notion of an agent, one has to accept the fact that an agent should be able to communicate with other agents and cooperate with them. Typically, agent-based applications comprise many agents (possibly of different type and functionality). In order to enhance such a multi-agent framework with communication and cooperation capabilities we need an *Agent Communication Language* (ACL) which will be used for the purpose of exchanging information, intentions or goals. An ACL is also used to allow agents to ask for support from other agents in order to achieve collectively some goal, monitor agent execution, report the status of some computation, organize task allocation, etc. In other words, an ACL offers the ability to formulate basic coordination patterns.

There are basically two main categories of ACLs (\{Nwana(a)}): Traditional third generation multiple-purpose languages that are used, among other things, for agent communication. Such languages are C and Java, AI languages such as Lisp, Prolog and Smalltalk, and OOP languages. We will not elaborate further on this category. The second category involves the development of language formalisms specifically designed for the purpose of offering inter-agent communication and cooperation. Below, we review some of these approaches.

KQML. The Knowledge Query and Manipulation Language (KQML) is probably one of the most widely accepted and used ACLs (\{Finin}). It has been developed as part of the DARPA Knowledge Sharing Effort project and is considered an evolving standard. KQML is based on the notion of modeling illocutionary acts, such as requesting or commanding an agent to perform certain things. These requests are called *performatives* and can be classified into nine categories, some of which are directly related to the notion of coordination. More to the point, there are performatives that offer basic communication capabilities; for instance, the *networking* performative offers the primitives `register`, `unregister`, `forward`, `broadcast`, `pipe`, and `break` with self-explanatory functionality. However, there is also the *facilitation* performatives with the primitives `broker-one`, `broker-all`, `recommend-one`, `recommend-all`, `recruit-one`, and `recruit-all` which offer more sophisticated coordination patterns. The facilitation performatives are used by a special class of agents, called *facilitators* or *mediators*, which are used effectively as coordinator agents for the rest, and whose purpose is to manage various communication actions such as maintaining a registry of service names, forwarding messages to named services, routing messages based on content, providing “matchmaking” between information providers and clients, providing mediation and translation services, etc.

KQML can be viewed as comprising three layers of abstraction:

- The bottom layer, referred to as *content layer*, specifies the actual content of the message. This can be represented in any programming language, as long as it is ASCII-representable.
- The middle layer, referred to as *message layer*, consists of the primitives that comprise the nine classes of performatives, and forms the core of the language. This layer specifies the protocol for delivering the message, whose contents are specified by the previous layer.
- The upper layer, referred to as *communication layer*, is used to encode communication parameters such as the identities of senders and receivers.

The actual format of a KQML message is shown below:

```
(register
         :sender             agentA
         :receiver           agentB
         :reply-with         message
         :language           common_language
         :ontology           common_ontology
         :content            "something_to_do"
)
```

The first keyword identifies the particular performative that is being used (in this case it is the `register` one), followed by the a number of parameteres. These include the parameter `ontology` which identifies the ontology (i.e. the specification scheme for describing concepts and their relationships in a domain of discourse) to interpret the information in the content field of this message.

As a particular example, the agent `customer` could ask for all the different types of 4x4 cars from an agent `seller`, as follows:

```
(ask-all
         :sender             customer
         :receiver           seller
         :content            "cars(4by4(Type,Price))"
         :reply-with         4by4-request
         :language           Prolog
         :ontology           CARS
)
```

The agent `seller` receiving this message would interpret the request using the ontology `CARS` and bearing in mind that the request was formulated in the language Prolog. In due time, it will reply using the identification `4by4-request` so that the agent `customer` would know for which of its (potentially many) requests this message constitutes a reply. Such a reply message could be the following:

```
(tell
         :sender           seller
         :receiver         customer
         :content          "[4by4(honda,15000), …]"
         :in-reply-to      4by4-request
         :language         Prolog
         :ontology         CARS
)
```

A coordinator agent, monitoring continuously the changes in the 4x4 cars market and informing the agent customer, could be defined as follows:

```
(monitor
         :sender           customer's_agent
         :receiver         seller
         :to               customer
         :content          "cars(4by4(Type,Price))"
         :reply-with       4by4-request-for-customer
         :language         Prolog
         :ontology         CARS
)
```

In the above example, the intermediary agent `customer's_agent` has used the performative `monitor` (which is an abbreviation for a combination of some other primitives) to keep itself informed of all future changes to the 4x4 cars and inform the agent `customer`.

Although KQML has been criticized for a number of shortcomings, such as the lack of precise formal semantics, it is interesting to highlight the three layer abstraction that separates the communication aspect of the language from the way the actual information is being computed and presented. This separation of concerns is one of the most important features of coordination languages and thus KQML can be seen as a truly, if elementary, coordination language.

Agent Oriented Programming. AOP is a more elementary, compared to KQML, framework for agent programming, and can be considered a specialization of OOP where agents are viewed as objects with *mental* states (such as beliefs, desires and intentions) and a notion of time. AOP is effectively a family of evolving formalisms. A program written using the first member of this family, Agent-0 (\{Shoham}), executes at each time step the loop comprising two steps: during the fist step incoming messages are gathered and the mental state of the agent is updated appropriately, and during the second step *commitments* (i.e. guarantees that the agent will carry out an action at a certain time) are executed using *capabilities* (actions the agent is able to perform). Basic communication in Agent-0 is achieved by means of the primitives `(INFORM t a fact)`, `(REQUEST t a action)`, `(UNREQUEST t a action)` and `(REFRAIN action)` where `t` specifies the time the message is to take place, `a` is the agent that receives the message, and `action` is any action statement. An `INFORM` action sends the fact to agent `a`, a

REQUEST notifies agent a that the requester would like the action to be realized. An UNREQUEST is the inverse of a REQUEST. A REFRAIN message asks that an action not be committed to by the receiving agent.

PLACA (\{Thomas}) extends Agent-0 with *intentions* (a commitment to achieve a state of the world) and ability to plan composite actions. New syntactic structures added to PLACA include the following: (INTEND x) intending to make sentence x true by adding it to the list of intentions, (ADOPT x) adopting the intention / plan x to the intention / plan list, (DROP x) dropping the intention / plan x, and the set (CAN-DO x), (CAN-ACHIEVE x), (PLAN-DO x), (PLAN-ACHIEVE x), (PLAN-NOT-DO x) which are truth statements used in mental conditions. For the purposes of this chapter, we can view the primitives of Agent-0 as offering basic coordination at the level of communication whereas those of PLACA as offering basic logical coordination.

Agent-K (\{Davies) is an attempt to standardize the message passing functionality of Agent-0 by combining the syntax of Agent-0 with the format of KQML. In the process, the communication primitives of Agent-0 have been replaced by a single generic message action kqml(time,message) where message is of the form [performative,keyword(action)].

A simple communication pattern in Agent-K between two agents agent1 and agent2 would be modeled as follows:

```
commit([clock(Now),b([Now,alive(agent2)])],
       kqml(Now,['ask-all',sender(agent1),
            receiver(agent2),content(…),language(…)])

commit([clock(Now),b([Now,alive(agent1)]),
       kqml(Now,[reply,sender(agent2),
            receiver(agent1),content(…),language(…)])
```

where b(…) represents a belief and a predicate clock provides the current time.

Market Internet Format. We include in our survey the MIF (\{Eriksson}) formalism, as an example of how KQML can influence the design of more specialized ACLs which are designed and oriented towards specific applications. In the case of MIF this application is e-commerce which has attracted a wide interest and has shown to be a natural application domain for both agent technology and coordination models. In MIF agents share a common language which is a formalized subset of commerce communication. MIF agents leave within the MarketSpace, a medium reflecting a market place where consumer goods and services can be offered and bought. Interaction between agents is modeled in MIF, a Lisp-like frame language, which has both a textual and graphical representation. Typical MIF expressions of interest for e-commerce transactions are the following one:

```
(def car "trade-object"
     color (instance "red"))
… )
```

```
(instance "contract"
   date(interval 1/1/2000 1/1/2001)
   buyer(instance "person"
         (name "John Brown")
         (address …) …)
   goods(instance "car"
         color(instance "red") …)
```

MIF expressions are exchanged using messages written in the Market Interaction Language (MIL) which have the following format:

```
(offer
    :from           url
    :to             url
    :in-reply-to    message_id
    :language       "MIF 1.0"
    :content        <MIF expression>
)
```

The basic communication primitives that can be used to formulate complex coordination patterns can be grouped into two categories: non committing messages (`ask` (for an expression-of-interest), `tell` (an eoi) and `negotiate` (an eoi)) and committing ones that are understood to make legally binding agreements (`offer` (an offer), `accept` (an offer), and `decline` (an offer)). The involvement of specific agents in such communication scripts can be parameterized, thus rendering frequently used scenarios (such as auction protocols) reusable.

April. The Agent PRocess Interaction Language (\{McCabe}) is a process oriented symbolic language influenced by the actor paradigm. April is oriented primarily towards offering a basic language for modeling agent interactions rather than a high-level set of agent related features such as planners, knowledge representation systems, etc. In that respect April is used mainly for managing the processes representing agents and their actions as the former interact with each other in a distributed system. The following example shows how a broker agent can be modeled in April:

```
agent_record ::= (handle?agent,symbol[]?relnames);
subscription_record ::= (handle?agent,symbol?relname);
broker(){
   agent_record[]?has_rels := [];
   subscription_record[]?sub_for := [];
   repeat {
      (advertise,symbol[]?rels) -> { … }
    | (subscribe,symbol?rel) -> { … }
    | (remove_subscription,symbol?rel) -> { … }
    | …
   until quit ::= sender == creator()
}
```

The code starts with the declaration and initialization of the data structures that will hold the data on those agents that have advertised or subscribed to the broker. The process then enters a loop to handle the requests for its services, such as advertising something, registering or removing a subscription, etc. (the particular actions taken for each request are not shown above for reasons of brevity). This process executes until its creator has sent it a `quit` message. Recently (\{Skarmeas}) April has been used as a component-based platform for implementing KQML in a distributed environment.

2.2 Compositional Platfoms

In this subsection we briefly describe some approaches that lend support in the development of Internet agents-based systems. We refer to both general purpose technologies that are used in distributed computing but also to those that have been designed primarily to assist in the development of agent-based systems.

Java-Based Agent Toolkits. The rapid developments in object-oriented programming and wide area networkings has led to the integration of these technologies and the formation of distributed object-based computing platforms. These platforms allow the development of systems as a synthesis of pre-existing components. Furthermore, they provide a natural medium for constructing agent layers. This has led to the development of agent toolkits that are typically Java-based. One such toolkit is JavaSpaces (\{JavaSpace}) and its associated infrastructure Jini (\{Jini}). JavaSpaces allows dynamic sharing, communication and coordination of Java Objects. It is a loosely coupled cooperative marketplace model, based on the metaphor of Linda like models (see next section), whereby producers store objects in a shared working space and consumers lookup and retrieve objects from this shared medium. The shared medium is effectively a networked repository of Java Objects where the latter exist in the form of *entries* (serialized objects with both data and behaviours) and their lookup is done via *templates* allowing type and value matching. JavaSpaces is 100% pure Java-based and provides a simple solution to lightweight distributed applications. Furthermore, it decouples requestors from providers, thus relieving responsibilities and complexity and reducing the difficulty of building distributed applications and maintaining them. The model supports five simple but very powerful primitives:

- `write`, puts a copy of an entry into the space.
- `read` & `readIfExists`, (blocking and non-blocking versions) return a matching entry from the space.
- `take` & `takeIfExists`, (blocking and non-blocking versions) remove the matching entry from the space.
- `notify`, sends an event when the matching entry is written to the space.
- `snapshot`, returns another entry object that contains the snapshot of the original one.

As a basis technology, JavaSpaces uses Jini, a protocol allowing “plug and play” functionality for new entities connected to the network. Such an entity can be a device or a software service, which when connected to the network announces its presence. Clients can then use lookup facilities to locate and invoke the services offered by such entities. Thus, a Jini environment is made up of three main parts: the services offered, the clients that will invoke the services, and a service locator that implements the lookup capability.

There are a number of other similar toolkits which for reasons of brevity we will not describe in this chapter. These include Concordia by Mitsubishi (\{Concordia}), IBM's Aglets (\{Aglets}), Odyssey by General Magic (\{Odyssey}), and Voyager by ObjectSpace (\{Voyager}). The interested reader can consult the relevant references for further details.

Law Governed Interaction. One of the most important problems that must be faced in open distributed systems, such as Internet (multi-) agent ones, is that of security. A framework based on specifying and enforcing "laws" which must be obeyed by all entities involved in an application and which provide, among others, safe communication is described in \{Minsky}. The model has a wide applicability, but is particularly attractive to the case of Internet agent-based systems. In Law Governed interaction protocols, there exist a set of *controllers*, one controller per entity involved in an application, which intercept all communication between this entity and the rest in the apparatus. Each controller executes a copy of the law which defines in precise ways how the communication between the entities must be realized. The controllers monitor all message exchanges and allow the completion of only those which do not violate the law. As an example, the following is part of a law which establishes a secure bidding policy between agents involved in an e-commerce transaction:

```
R1. sent(C1,out([requester(C2),service(S)]),ts :-
    C1==C2, do(forward).
R2. sent(C1,in([requester(C2),service(S)]),ts :-
    C1==C2, do(forward).
R3. sent(C1,in([offerFor(C,S),fee(f),provider(P),
    provider(P),contact(Addr)]),ts) :-
    C1==C2, do(forward).
```

In the above example we assume that the agents communicate via a common medium, referred to as `ts`; furthermore, a message can be sent to `ts` by means of an `out` primitive and retrieved from there by means of an `in` primitive. Assuming further that a controller monitoring an agent `C1` executes the above law, then the first rule of the law says that `C1` can request a service provided it is for himself, the second one allows the withdrawal of the request, and the third rule allows `C1` to retrieve an offered service if it has been posted to `ts` (by some other agent) for `C1`'s sake.

IMPS. The Internet-based Multi Agent Problem Solving (IMPS, \{Crow}) is a compositional platform for developing Internet multi-agent systems. It features the use of *Problem Solving Models* (PSMs) which may be used in ontology construction and knowledge acquisition. A knowledge library is available to all agents containing information about PSMs in terms of their competencies and domain knowledge requirements, and about types and locations of domain knowledge sources and how to extract different kinds of information from them. This library can be distributed over the Internet and is designed to be modular and extensible by means of 'plug-and-play' new classes written in Java. More to the point, knowledge sharing in IMPS is realized by means of using emerging standards such as KIF, KQML and Java, thus ensuring interoperability.

IMPS is built on top of JAT (Java Agent Template), an agent-level architecture featuring the use of two specialist server agents; these are the Knowledge Extraction Agent (KexA) and the Ontology Construction Agent (OCA) which are used for providing on-demand knowledge to Inference Agents (IAs). IAs specialize in performing particular process or inference steps; thus, IMPS enhances *cooperation* between agents collaborating towards the achievement of a common goal, reduces *redundancy* and increases *reusability* of agent cooperation patterns. The use of KQML allows the dynamic configuration of agents and further enhances the coordination capabilities of this framework. IMPS can be viewed as an enabling technology for modeling logical coordination (see section 4 below).

ADK. The AgentBean Development Kit (ADK, \{Gschwind}) is a component-based framework for developing mobile multi-agent systems with some emphasis on network and system management issues. In ADK an agent is understood to be composed of components belonging to one of the following three sets:

- *Navigational* components. They are responsible for managing the itinerary of an agent which may be static of dynamically modifiable at run time.
- *Performer* components. They are responsible for carrying out the management tasks that should be executed at the host of the currently visited place. Tasks performed by agents comprise one or more components.
- *Reporter* components. They are responsible for delivering agents' results to designated destinations. Delivery can be a simple point-to-point exchange of messages of more complicated — for instance, collecting a number of messages (possibly from different sender agents) and forwarding all of them to some recipient agent.

For the purposes of this chapter, the above separation of concerns between the three categories of components enhances the coordination capability of the system. For instance, the reporter components can be seen as encapsulating the coordination activities of some ensemble of agents, thus separating them from other concerns, but also rendering useful coordination interaction patterns reusable. The interaction between components is done in an *event/action*-based fashion: events generated by one component may trigger actions to be performed by another.

JGram. JGram (\{Sukthankar}) is another multi-agent development platform using the component-based approach. Agents' services are specified in the JGram Description Language and automatically converted into Java source templates. These services may then be invoked synchronously or asynchronously in a manner transparent to the services' implementation. Compositionality of tasks is realized by means of the notion of *pipelining*: an agent may dynamically delegate tasks to other agents and chain together their results. Thus, the complexity of handling a task is distributed across a number of agents, each being aware of only part of it. This notion of pipelining can be seen as an extension of the Unix pipes where the agents involved in a pipeline can be distributed across the network. Furthermore, it is possible to form hierarchies of pipelines with transparent propagation of results.

Communication between the agents involved in pipelines is realized by means of JGram *slates*, consisting of a header specifying addressing information and delivery instructions and a body containing a set of typed entities. Slates are passed over from one agent to another for the purpose of accessing and if necessary modifying the entries in it. The system adheres to elementary coordination principles by taking the responsibility itself for performing parameter checking, thread management, authentication, agent name service and error handling. Thus, the user can concentrate only on creating and using agent services. In particular, the JGram Description Language provides high-level concepts for agent behaviour in the form of (offered) *services* and *requests* (for services). A user expresses an interaction scenario among a number of involved agents using these two notions and the underlying system handles the low level communication details.

RETSINA. The REusable Task Structure-based Intelligent Network Agents (RETSINA, \{Giampapa}) system plays particular emphasis to the process of formulating planning actions between a number of agents involved in the pursuing of a common goal. The RETSINA system architecture is composed of four autonomous units as follows:

- *Communicator*. It is responsible for exchanging requests between agents in KQML format.
- *Planner*. It transforms goals into plans that solve these goals.
- *Scheduler*. It schedules for execution the tasks representing the plans.
- *Monitor*. It monitors the execution of the plans.

The above separation effectively provides elementary coordination capabilities to the system. Furthermore, the system employs a *planning* and *refinement* algorithm which decomposes a complex plan into a *Hierarchical Task Network* (HTN) of more elementary plans. Every partial plan in the HTN is handled by the planner (and the rest of the units as listed above) of some agent, irrespective of how other agents deal with the rest of the plans. Thus, at any moment in time each agent is aware and interested only in his own local partial plan with the positive consequence that agents may dynamically join and leave the system. This mechanism provides further potential for developing parametric and reusable coordination patterns between cooperating agents.

Domain Specific Languages. The advantages of using Domain Specific Languages (DSLs) for modeling collaboration scenarios between Internet agents is discussed in \{Fuchs}. DSLs provide a common communication language between all types of agents (human or other) involved in Internet applications. In DSLs a clear separation is enforced between syntax and semantics, so that each agent in a collaboration is capable of applying a behavioural semantics appropriate to its role (e.g., buyer, seller, etc.). Thus, DSLs support the development of multi-agent applications from heterogeneous agents, an issue of importance to coordination frameworks.

Language semantics in DSLs are separated into two levels: the abstract semantics refer to the objects in the domain itself whereas the operational semantics refers to how the messages received by objects will be processed by the machine. What differentiates DSLs from ordinary languages in that respect, is that the "machine" is a specialized computational entity according to what precisely is the domain in question. So, this entity could be a machine specialized in playing bridge or a whole corporation with workflow infrastructure, databases and Internet communication mechanisms, according to the application framework.

\{Fuchs} uses SGML/XML as a metagrammar for defining DSLs. The following piece of code defines the grammar of part of the scenario that models a game of bridge:

```
<!ELEMENT bridge (player+,deal,bidding,dummy,play)>
<!ELEMENT player #EMPTY>
<!ATTIST player  position (north|south|east|west) #REQ
                 name cdata #REQ>
<!ELEMENT deal (card+)>
<!ELEMENT card #EMPTY>
<!ATTIST card suit (spades|hearts|diamonds|clubs) #REQ
              face cdata #REQ>
… … …
```

Part of an actual scenario (for dealing only), based on the above grammar is shown below:

```
<bridge>
<player position='north', name='george'> …
<deal><card suit='spades', face='king'>, … </deal>
… … …
</bridge>
```

In a distributed realization of this game, each agent will receive and interpret a bridge string as the one shown above. This string may be parsed as a data structure by a computational agent or be presented to a human agent in some visual and interactive form. The instance of a bridge game, as specified by the code enclosed in the `<bridge>…</bridge>` tags, will be played according to the rules for bidding, passing and using trump cards (not shown above) as they have been defined in the grammar above.

3 Coordination Frameworks

In this section we describe a number of approaches where coordination between Internet agents is supported as a "first class citizen". Contrary to the previous section where the emphasis was on systems whose principles have the *potential* of developing coordination frameworks, here we concentrate on genuine coordination models and languages that have specific features for Internet agents. Many of these models have evolved from earlier, more conventional versions, that deal with non (Internet) agent-based distributed computing (\{Papadopoulos}). Most of the models in this section adopt the notion of a Linda-like shared dataspace; however, some follow a more control-driven approach (\{Papadopoulos}) and a few are based on other notions such as using graphical notations.

3.1 Shared Dataspace Models

TuCSoN. TuCSoN (\{Cremonini}) addresses in particular two important problems that must be faced in open Internet agents-based systems: those of security and authentication. The model is an extension of the Linda framework and it uses the same set of primitives for dealing with tuples. As in other variations of the vanilla model, TuCSoN introduces multiple tuple spaces, referred to as *tuple centers*. Thus, an ordinary Linda tuple operation `op(tuple)` is now parametric to the particular tuple center `tc` which is being accessed and takes the form `tc?op(tuple)`. Furthermore, the tuple centers are associated with their own policies for being accessed by agents; an agent attempting to access some tuple center will undergo an authentication screening according to the particular policies of the tuple center that it tries to access. Thus, the tuple centers become *programmable media* that define locally the way agents will interact with them. TuCSoN views the Internet world as a hierarchical collection of different tuple centers. For instance, one tuple center can be the main gateway of some Web site (e.g. the gateway for some organization), comprising a number of subordinate tuple centers (e.g. local departments within the organization). This allows the optimization of enforcing security policies in the sense that the programmable "logic" of a tuple center for some tree hierarchy regarding how it is being accessed by external agents, may refrain from authenticating an agent that tries to access a sub-domain of the tree if that agent has already received security clearance from the top domain.

The consequences of the tuple space acting as a programmable medium means that there exist now two different levels of perception: that of the agents accessing it and that of the medium itself handling the queries. Thus, every logical operation at the level of the agent must be mapped onto one or more corresponding system actions at the level of the medium and vice versa. This introduces the notion of *reactions* and the primitive `reaction(Operation, Body)` where every logical `Operation` is mapped onto one or more system operations (`Body`). As an example, consider the problem of coordinating the well-known dining philosophers to access their forks:

```
reaction(out(forks(F1,F2)),
(in_r(forks(F1,F2)), out_r(fork(F1)), out_r(fork(F2))))

reaction(in(forks(F1,F2)),(pre,out_r(required(F1,F2))))
reaction(in(forks(F1,F2)),(post,out_r(required(F1,F2)))
)
reaction(out_r(required(F1,F2),
(in_r(fork(F1),in_r(fork(F2)),out_r(forks(F1,F2))))

reaction(out_r(fork(F)),(rd_r(required(F1,F)),
in_r(fork(F1)),in_r(fork(F)),out_r(forks(F1,F))))
```

```
reaction(out_r(fork(F)),(rd_r(required(F,F2)),
in_r(fork(F)),in_r(fork(F2)),out_r(forks(F,F2))))
```

In the above modeling of the problem in TuCSoN, two points of reference are being involved. The agent philosopher perceives the fork resources as pairs and asks for them in that fashion, namely `forks(F1,F2)`. The programmable medium however must map the agent's perception of pairs-of-forks to single forks and furthermore, ensure that these are accessed atomically and in a way that is fair to all philosophers. This is achieved by means of a number of reaction rules. The first one changes a release of the left and the right fork as a pair to two single releases. The next two rules refer to the case of an agent requesting a pair of forks in which case a request is posted to the medium (first rule of this group) and is retracted when the forks have been allocated to the agent (second rule of this group.). These requests are handled by the last three rules; the first rule of this group allocates immediately the two forks if they are both available whereas the last two rules handle the case when only one of the two forks can be immediately given to the requesting agent.

KLAIM. The Kernel Language for Agent Interaction and Mobility (KLAIM, \{Nicola}) is another Linda variant for coordinating Internet agents with similar characteristics to TuCSoN. A KLAIM program is a *net*, comprising a set of *nodes*. Each node has a name and is associated with a process component and a tuple space component. The name of a node is effectively an Internet site and allows access to the network. Processes access tuple sites via symbolic *locality* references; in other words, they need not know explicit network references. Thus the net can be seen as a distributed infrastructure for coordinating processes in accessing and sharing resources.

In particular, each node in a net is of the form `e {P | T}` where `e` is an allocation environment that maps symbolic locality references to actual tuple spaces, `P` is a set of running processes, and `T` is a tuple space. Consider the following piece of KLAIM code:

```
def Client = out(Q)@l ; nil
def Q = in('foo',!X)@self ; out('foo',X+1)@self ; nil
def Server = in(!P)@self ; eval(P)@self ; nil
```

In the above example, the first line of code defines a process `Client` that `out`s a process `Q` to the environment `l`. The actual definition of `Q` is given in the second line and involves the increment of the value of some variable `X`. The third line of code defines a process `Server` which retrieves a process `P` from its own tuple space and executes it. The idea in this example is that `Client` sends an increment process `Q` to some other process `Server` which will then execute `Q` and send the new increment value back to `Client`. Assuming that `Client` and `Server` run on the nodes `s1` and `s2` respectively, this will work provided that `l` is mapped to `s2` and the `self` references of `Client` and `Server` map themselves to `s1` and `s2` respectively. Furthermore, we assume that before the execution of these processes commences, an initial tuple `<'foo',1>` exists in the tuple spaces involved in the scenario. These are achieved by the following piece of code:

```
node s1 :: e1 {Client | out('foo',1)} ||
node s2 :: e2 {Client | out('foo',1)}
```

When `Client` sends `Q` for execution to `Server`'s tuple space, it will appear there as the process:

```
Q' = in('foo',!X)@s1 ; out('foo',X+1)@s1 ; nil
```

Thus, `Server` will execute `Q'` in its own tuple space `s2` but post the result to the tuple space `s1`, i.e. will effectively send the result back to `Client`.

KLAIM has a capability-based type system to express and enforce access control policies and thus provide security. These types provide information regarding the intention of some process with respect to producing or using tuples, creating new nodes or activating processes. Permissions have a hierarchical structure in the sense that if a process is allowed to perform an operation of a certain generality or "strength", then by default it is also allowed to perform all other operations that are less general or weaker. For instance, if it is allowed to read a real value then it is also allowed to read an integer value, and if it is allowed to remove a tuple (`in`) then it is also allowed to simply read it (`rd`). As an example, the following KLAIM code specifies access control rights for the processes `Client` and `Server`:

```
def Server = out(<l:void,Top>)@self ; nil
def Client = read(!u:<[self -> e], ac>@l-S;
             eval(P)@u ; nil
```

According to the above definitions, `Server` adds a tuple containing the locality `l` to its own tuple space; no access restrictions are specified on `l`. The process `Client` first accesses the tuple space `l-S` to read an address `u` before sending process `P` to execute at `u`. However, this will only be possible to achieve if `P` is of type `ac`, because the second rule states that only processes of this type are allowed to be sent from the site of `Client` to the site `u`.

LIME. Linda in a Mobile Environment (LIME, \{Picco}) is yet another extension of the Linda model for coordinating Internet agents. However, contrary to the previous two models that develop fully-fledged languages, LIME offers only a minimalist set of constructs. The philosophical difference between models like the previous two and those such as LIME is that in the former case the user has direct and explicit control on how to deal with coordination matters specific to Internet agents (such as mobility or security) while in the latter case the user only implicitly expresses the intended actions to be performed and it is the system that is primarily responsible for dealing with such issues.

The fundamental abstraction provided by LIME is that of a *transiently shared tuple space*. In particular, each agent is associated with its own personal tuple space, referred to as *interface tuple space* (ITS). An agent may have one or more ITSs identified by a separate name. The union of the ITSs with the same name that belong to all the agents that are co-located at some host form the transiently shared tuple space for that host with respect to the currently residing there agents. When a new agent moves to some location, the LIME system recomputes the transiently shared tuple space for that location taking into consideration the ITSs of that agent. This process is called *engagement*. When an agent leaves the host, LIME again recomputes the transiently shared tuple space for that host by removing those tuples that belong only to the ITSs of the departing agent. This reverse process is called *disengagement*.

Thus, if two agents A and B reside on the same host and A performs the operation out(t) to its own ITS (we assume for simplicity here that only one ITS per agent is involved in our scenario), then because the two agents are co-located and the two ITSs form a common transiently shared tuple space, B can perform the operation in(t) and retrieve the tuple t (which can be seen by B through his own ITS). Care must be taken when, after performing the operation out(t), A migrates to some other host. In this case, the transiently shared tuple space between the ITSs of A and B does not exist any more and the tuple t would go along with A and would not be any more accessible to B for retrieval. In order to allow access to t even after the departure of A, the latter must out it to the ITS of B, rather than to its own ITS by means of executing the primitive out[B](t). In this case, t becomes part of B's ITS and will remain there even after the departure of the agent (A) which created it, although the two agents are not co-located on the same host any more.

Berlinda. Berlinda (\{Tolksdorf}) is a meta-coordination Linda-based platform developed in Java. The system offers a highly abstract model of coordination that can be used as the basis for developing more concrete coordination frameworks for Internet agents. As in all Linda models, there exists a common communication medium in the form of *multisets*, which comprises a collection of *elements*. Elements carry *signatures* with meta information and provide a *matching function* for their access by agents according to the semantics of the particular coordination framework that is being employed. All these entities are implemented as Java classes that form a hierarchical structure and provide appropriate operations that realize the functionality of the coordination framework.

The Berlinda platform has been used for developing coordination frameworks for Linda and KQML. As an example, the following piece of code creates a set of agents in a Linda coordination framework that traverse a file system and remove unnecessary files, i.e. those files that can be generated from some other file:

```
public class SweepAgent extends LindaAgent {
public static void main (String argv[]) {
   // create tuple space
   TupleSpace ts = new TupleSpace();
   // create agents
   for (int i=1; i<=walker; i++) ts.eval(new Walker());
   for (int i=1; i<=sweeper; i++)ts.eval(new
Sweeper());
   // allocate work to agents
   ts.out(new Tuple("Sweeped", new Integer(0)));
   ts.out(new Tuple("Walker",start_directory_path));
   … … …
   }
   …
}
```

The `Walker` agents traverse a directory, spawning themselves to traverse in parallel any subdirectories. The results of their search are passed on to the `Sweeper` agents that remove the selected files.

PageSpace. The PageSpace platform (\{Ciancarini}) is effectively a meta-architecture or reference architecture for developing Internet agents-based applications. Applications are composed of a set of distributed agents and are conceived as comprising three layers: a *client* layer, a *server* layer and an *application* layer, that coordinate modules belonging to client agents, server agents or intranet applications respectively. These applications may be distributed transparently across a network and used in serving several users who independently access a shared Linda-like environment via their WWW browsers. Independently progressing applications may interact with each other in order to cooperate towards the achievement of a common goal. Furthermore, the configuration of users, applications and hosts may change dynamically without disrupting the offered services.

Depending on their functionality, PageSpace distinguishes several kinds of agents, such as *user interface* agents, *application* agents (that manage the running of some application), *gateway* agents (that provide access to the outside world), *kernel* agents (that perform management and control tasks), etc. PageSpace is effectively the product of combining related research at the University of Bologna and the Technical University of Berlin and it thus uses a number of more specialized software architectures and associated coordination models and languages that have been developed by the two groups such as MUDWeb, Shade/Java, or MJada (\{Rossi}). As an example of Internet agent coordination in PageSpace, we show extracts of a Shade/Java program that coordinates the process of bidding in an e-commerce scenario involving three groups of agents: an auctioneer agent that sells items to participant agents while some observer agents passively watch the process. Shade/Jada is a combination of Java with the Linda-based coordination language Shade. A Shade program is a collection of classes and each object in a Shade application is a class instance. Objects communicate by means of Linda-like communication primitives. Thus, Shade/Java is a syntactic extension of Java with the coordination features of Shade (\{Rossi}). Regarding the example in question, the code is part of the auctioneer agent functionality:

```
class auctioneer extends ShadeObject {
  in ("begin");
  out ("bid","bid1"), ("next_item","next1"),
      ("cartoon","car1"), ("display","dis1"),
      ("BasePrice",5000), ("next_init","init_2"),
      ("item#",1);
  #
  in ("BasePrice",?i:base_bid);
  out (("display",?s:display), ("item#",?i:num);
  send [bid, ("begin_auction","auctioneer",base_bid,0),
             ("item#",num)],
  send [display, ("begin_auction","auctioneer",base_bid,
                  0)];
```

```
   out ("auction_active"), ("first_timer"),
       ("current_bid",base_bid), ("TimeStamp",0);
   #
 … … …
```

The above piece of code shows two of a number of methods that the auctioneer agent comprises (the code for each method is separated by '#'). The auctioneer starts the auction when it receives the tuple `begin` in which case the first method above is activated. This method broadcasts a number of initialization tuples that activate the agents to be engaged in the scenario. Furthermore, when the first item goes on sale, it is displayed by the agent `dis1` of class `display`. The second method commences the coordination of the bidding process. An auction starts from a base price which is sent to all participating agents (the tag `item#` on some item denotes that that item is to be sold). Further methods receive bids, validate them and modify appropriately offered prices for sellable items.

MARS-X. MARS-X (\{Cabri(b)}) is a programmable coordination architecture for Internet agents based on a combination of XML and a Linda-like communication mechanism. The XML component enhances interoperability by separating the treatment of data from its representation. The Linda-like communication mechanism offers the required coordination mechanisms for modeling cooperation between agents. MARS-X is a four layer architecture: at the lowest layer lays the actual information being manipulated and in the next level the XML dataspace; the third layer comprises the Linda-like interface (based on Sun's JavaSpaces) and at the upper (application) level lie the executing agents. There exists a local per node in the network XML dataspace, and when a mobile agent arrives at a node it is provided with a reference to this dataspace. Groups of nodes can create shared federated dataspaces. Access to a dataspace is realized by means of the operations `read`, `take` and `write` which correspond directly to Linda's `rd`, `in` and `out`. There are also the aggregate variants `readAll` and `takeAll` which retrieve all matching tuples. The following piece of codes illustrates the modeling of agents in MARS-X:

```
<?XML version="1"?>
<course>
   …
   <lesson>
     <lessonname>Introduction</lessonname>
     <lessonnumber>1</lessonnumber>
     <abstract>…</abstract>
     …
</course>

class_lesson extends AbstractEntry {
   static private URL DTDfile = new URL(http://…);
   public String lessonname;
   public Integer lessonnumber;
   public String abstract;
   … }
```

```
lesson tmplesson = newlesson();    // template lesson
tmplesson.abstract="networks"      // partially def field
for (i=0; i<number_of_federation_sites; i++)
   {go(site[i];        // current site in the federation
    if(lesson=S.read(tmplesson,…) // if a lesson with
      go(home); // the right keyword is found go home
   }
```

In the above example, the first part of the code describes in XML the structure of a lesson, as part of some course. The second part defines as extended Java classes the MARS-X tuple corresponding to a lesson. Finally, the last part makes use of the Linda like primitives to define the behaviour of an agent which roams a federated site (i.e. a collection of local XML dataspaces) in order to find and retrieve the lesson with the keyword "`networks`".

3.2 Other Coordination Models for Internet Agents

We present below some other coordination models, particularly suited to Internet agents, which however are not based (at least directly) on the Linda model. Here we find a variety of flavors: those that use a point-to-point communication mechanism and can be characterized as being control-oriented (\{Papadopoulos}), or others which are based on extension of existing programming paradigms such as logic programming or visual programming.

STL++. The Simple Thread Language ++ (STL, \{Schumacher}), an evolution from earlier coordination languages, is a control-driven coordination formalism for Internet agents which is based on the Encapsulation Coordination Model (ECM). Unlike the members of the previous category of coordination models in this section which are variants of Linda (and therefore they are relying on a notionally shared dataspace), ECM and its associated language ECL++ are relying on point-to-point communication. In particular, there exist five building blocks:

- *Processes*, as a representation of active entities.
- *Blops*, as an abstraction and modularization mechanism for a group of processes and ports.
- *Ports*, as the interface between processes/blops and the outside world.
- *Events*, as a mechanism for synchronizing the execution of processes and blops.
- *Connections*, as a means of connecting ports.

A coordination ensemble in STL++ is a collection of agents, themselves grouped in blops, with well defined port-based input-output interfaces which communicate via their respective ports by means of port-to-port connections, and synchronize their activities by means of events. The language is object-oriented and has been realized as an extension of C++. As an example, the following is an extract from an STL++ program coordinating the activities in a restaurant:

```
void Waiter :: start() {
  Agent :: start();
  int income;
  tablr_port = new BB_Port<int>(this,nV("MealBB"),INF);
  int i, j;
  for (i=0; i!=nbrOfClients; i++) {
   j=(i+1)% nbrOfClients;
   createAgent(Client,&i,&j);      // Create the clients
  }
  // Restaurant closes - take the money
  income=table_port->get("money");
  while (income) {
   total_income+=income;
   income=table_port->get("money");
  }
  stopMe();
}
```

The above code refers to a `Waiter` agent which manages the diner area in a restaurant. It is responsible for organizing a place for each newly arriving customer. It initializes the scenario by creating an initial number of customers. Finally, it collects the money for providing dinner and closes the restaurant. `Waiter` creates a port `table_port` with the name `MealBB` which will be used to collect money. Then it creates a number of clients, and finally it receives the money through `table_port` and sums up the income. The setting up of a connection between an appropriate port of the agents of type `Client` and `table_port` so that the money can be received, is not shown above and is part of the code for `Client`.

Mobile Streams. Mobile Streams (\{Ranganathan}) is a middleware platform for the development of distributed multi-agent systems. What is of particular importance to the issue of coordination, is that Mobile Streams is especially suitable for applications that require dynamic (re-) configuration. Furthermore, the system is event-driven in the sense that its components (namely mobile agents) synchronize their cooperation by means of sending and receiving events. The combination of these two features (event-driven dynamic reconfigurability) is typical of a particular class of control-driven coordination formalisms (\{Papadopoulos}) and Mobile Streams can be seen as being a mobile version of them.

A Mobile Stream (MStream) is a globally unique name for a communication end-point in a distributed system that can be moved from machine to machine, during the course of a computation and preserving the order of messages. A MStream is part of a hierarchical tree structured logical organization whose root is a *Session*, namely a distributed application. A *Session* comprises a set of *Sites*, at each one of which a number of agents execute and communicate via one or more MStreams. Each agent comprises a number of *Event Handlers* which handle events. This apparatus separates the logical design of a distributed application from the physical placement of components. A distributed application is constructed by first specifying the communication end-points as MStreams and attaching agents to them. The latter create event handlers, one (and only one) for a different event associated with an agent. When an event occurs, the appropriate handler in each agent is concurrently and independently invoked with appropriate arguments. When a MStream moves from one site to another, it (logically) moves the code of all the agents attached to it to the new site along with their state. The code can have initialization and finalization parts that execute once the agent first arrives at a site or when it is about to be killed. As an example, consider the following piece of code:

```
stream_create foo
stream_create bar
stream_move foo 1
stream_move bar 2

# external input
stream_open bar
stream_append foo "Hello World"

register_agent foo () {
  stream_open bar
  on_stream_append {
     stream_append bar $argv
  }
}

register_agent bar () {
  on_stream_append {
     puts $argv
     stream_relocate 1
  }
  on_stream_relocation {
     set my_loc [stream_location]
     puts "I am at $my_loc"
  }
}
```

The above script initially defines two streams, `foo` and `bar`, and locates them at different sites (`1` and `2`). We further assume that a string is sent to `foo` from an external source. The MStream `foo` receives the string message and sends it to the MStream `bar`, which outputs the message via its handler. Finally, `bar` moves to the site of `foo` and prints an appropriate message to announce its new location.

GroupLog. The agent coordination language GroupLog (\{Barbosa}) is based on an extended Horn Clauses formalism. Elementary agents in GroupLog are modeled as (possibly perpetual) processes which receive messages and react to them by invoking appropriate methods. A clause can have AND-conjunctions with sequential or parallel operational semantics. Sets of clauses defining the overall behaviour of agents are grouped into modules. In that respect, GroupLog is very similar to object-oriented concurrent logic programming languages such as POOL. What is particularly relevant to the notion of coordination however is the notion of *group*, which effectively structures the communication space of agents and allows the modeling of various cooperation patterns between them. Agents can dynamically join and leave groups and can be members of multiple groups at the same time. Group communication can be either broadcast or point-to-point. The following code defines such a group:

```
group meet_schedule {
  context().
  interface(begin).
  meet_schedule(Id) : begin :-
        members(meet_schedule(Id),[H,I]),
        rd(meet_schedule(Id),meet(MeetId)),
        H << begin(I,MeetId) || I << begin(H,MeetId)
                    | meet_schedule(Id).
  meet_schedule(Id) : new | meet_schedule(Id).
}
```

The group `meet_schedule` defines a broadcast communication mechanism between a number of agents `[H,I]` belonging to the same group `MeetId`. Using the communication operation `<<`, and the predefined primitives `members` (returns the agents belonging to the same group `Id`) and `rd` (returns the subset from a set of agents belonging to the same group), `meet_schedule` sets up communication paths between all members of this group so that they can exchange messages in a broadcast fashion. An agent joining the group `meet_schedule` will automatically become part of this communication apparatus. Furthermore, the communication strategy of this group may change dynamically without the agents belonging to it realizing any difference.

Little-JIL. We end this section with a brief description of Little-JIL (\{Jensen}), a visual language for agent coordination. Little-JIL has been designed to address in particular the problem of knowledge discovery in databases, an issue of particular interest to Web environments, especially with regard to aspects of traffic analysis, fraud detection, etc. Activities of processes in Little-JIL are represented as *steps*, decomposed into *substeps*. Substeps belonging to a step can be invoked either proactively or reactively. Steps may have *guards* to be executed upon entering or exiting a step, as well as handlers to deal with exceptions. They can also include *resource* specification. One special resource associated with each step is an *agent* which is responsible for initiating and carrying out the work of the step.

Coordination of agents is achieved by means of an *agent management system* (AMS). An AMS is based on the metaphor of to-do lists for activities to be performed by agents, human or automated. Assignment of tasks to be executed by some agent(s) are placed on the to-do lists of those agents. Agents monitor to-do lists (they may be associated with more than one list if they are involved in performing several disjoint processes), in order to receive tasks to perform. Any changes in the to-do list cause notification to be sent to the interested agents which then execute the corresponding tasks. Task execution causes changes to the system state and these changes are recorded by the AMS. Thus, the AMS provides language-independent facilities that allow coordination to take place in a way that separates the concerns about why and when coordination should occur (handled by AMS) from how it will be achieved (handled by the agents).

4 Logical Coordination

The previous two sections have dealt with a rather "technical" aspect of coordination, as it applies to the field of Internet agents. In particular, we first examined some enabling technologies that provide the necessary infrastructure for building coordination frameworks. We then presented some approaches in developing models and languages for Internet agents where coordination is treated as a first class citizen. However, the concept of coordination exists also at a higher, more logical, level where we are interested in organizing the cooperation behaviour between the members of a multi-agent ensemble. In this case, *middle agents* are used with the sole purpose of acting as coordinators managing the activities of other agents. Such coordination can be done centrally or in a distributed fashion. Depending on precisely what sort of coordination these agents realize, they can belong to a number of different categories, some of which are the following (\{Flores-Mendez}):

- *Facilitators* or *Mediators*, which satisfy requests on behalf of other agents, usually by offering certain services to these agents.
- *Brokers*, which also satisfy requests received by other agents but often by using services provided by third parties rather than themselves.
- *Matchmakers* (*Yellow Pages*), which offer look-up services.
- *Blackboards*, which are repository agents that receive and hold requests for other agents to process.
- *Local area coordinators*, which are agents responsible for assisting the other agents in some well defined area to initiate and conduct inter-agent communication and interaction.
- *Cooperation domain servers*, which provide agents in some domain with facilities to subscribe, exchange messages and access shared information.

Logical coordination techniques have been classified by \{Nwana(b)} into four main categories:

- *Organisational Structuring*, which provides a framework for activity and interaction through the definition of roles, communication paths and authority relationships; here classic *master-slave* or *blackboard* coordination techniques are being employed.
- *Contracting*, which involves the use of manager agents who break a problem into subproblems and assign each one of them to some contract agent to deal with them. This apparatus is often referred to as *contract-net protocol.*
- *Multi-Agent Planning*, where the agents build a plan that defines all current and future interactions among them in such a way that avoids inconsistent or conflicting actions. There are two ways to execute the plan: in *centralized* planning, a coordinating agent is responsible for setting up and executing the plan, whereas in *distributed* planning each agent is aware of the plans of the other agents and acts appropriately.

- *Negotiation*, involves a particular form of coordination where a number of agents interact with each other in order to reach a mutually accepted agreement on some matter. Negotiation techniques can be *game theory*-based, *plan*-based, or *human inspired.*

In the rest of this section we will present some approaches in realizing logical coordination at the level of modeling the behaviour of agents. The models in this section try to address such questions as how agents communicate and coordinate themselves in achieving a common goal, how are problems stemming from dynamical evolutions of agents or incomplete knowledge handled during the coordinated behaviour, or how patterns of interaction and interoperation that characterize coordinated behaviour are modeled. A major consequence of addressing these issues is the ability to *reuse* descriptions of generally useful coordination mechanisms.

COOL. The COOrdination Language (\{Barbuceanu}) is part of an effort to develop a more general Agent Building Shell that will provide reusable languages and services for agent construction that will relieve developers from the burden of developing from scratch essential interoperation, communication and cooperation services. The COOL architecture comprises three layers, with a basic KQML-like communication mechanism at the lower level, an agent and conversation management at the middle level for defining and executing agents and coordination structures, and an upper level that supports in context acquisition and debugging of coordination knowledge. We will not elaborate on the lowest level which is covered adequately by the material in section 2. Regarding the way the agent and conversation management models the behaviour of agents, we note that every agent is associated with a name and an interpreter which then selects and manages its conversations. The interpreter applies continuation rules to determine which conversation to work on next. The interpreter may also invoke more specialized agents for knowledge acquisition and/or debugging services. Such a scenario is shown below:

```
(def-agent 'customer
  :continuation-control 'agent-control-ka
  :continuation-rules '(cont-1 cont-2 cont-3 cont-4))
(def-agent 'logistics
  :continuation-control 'agent-control-ka
  :continuation-rules '(cont-1 cont-2 cont-3 cont-4))
(def-agent 'plant
  :continuation-control 'agent-control-ka
  :continuation-rules '(cont-1 cont-2 cont-3 cont-4))
```

In the code above three agents are defined, part of a supply chain application. The execution and control of these agents is managed by a conversation manager like the one below:

```
(def-conversation-manager 'm1
  :agent-control 'execute-agent
  :agents '(customer logistics plant …))
```

This manager decides which agent to run next, manages message passing, etc. Agents interact with each other by means of carrying out *conversations*; such a conversation for the agent `customer` is defined below:

```
(def-conversation-class 'customer-conversation
  :name 'customer-conversation
  :content-language 'list
  :speech-act-language 'kqml
  :initial-state 'start
  :final-states '(rejected failed satisfied)
  :control 'interactive-choice-control-ka
  :rules '( (start cc1) (proposed cc-13 cc-2)
           (working cc-5 cc-4 cc-3)
            (counterp cc-9 cc-8 cc-7 cc-6)
             (asked cc-10) (accepted cc-12 cc-11)))
```

The above code associates with some agent the conversation rules that govern its interaction with other agents. What particular activities are performed during the execution of such a rule is illustrated by the following code:

```
(def-conversation-rule 'crn-1
  :current-state 'start
  :received '(propose  :sender customer
                       :content(customer-order :item ?l))
  :next-state 'order-received
  :transmit '(tell     :sender ?agent
                       :receiver customer
                       :content '(working on it)
                       :conversation ?convn)
  :do '(put-conv-var ?conv '?order
                       (cadr(member :content ?message)))
  :incomplete nil)
```

This code defines the behaviour of the agent `logistics` in our scenario of supply chain management. When `logistics` is at the `start` state, it receives a proposal for an order and informs the sender (`customer`) that it is working on it before going to the next state `order-received`. The language also allows the formulation of two other coordination dimensions, cooperative information distribution and cooperative conflict management, but for reasons of brevity we do not discuss them here.

Agent Groups. The model described in \{Baumann(a)} introduces the notion of Agent Groups, comprising agents working together on a common task. Agent groups may be arbitrarily structured and highly dynamic. Communication and synchronization between the agents of a group is event-driven; the model assumes the existence of a mechanism for sending and receiving events. The group model used involves the following types of agents:

- A *Group Initiator*, which creates the agent group, an activity that involves assigning agents to groups, defining group coordinators, administrators and receivers of results.
- *Group Members*, which is the collection of agents forming a group, according to a common task pursued.
- A *Group Coordinator*, which models dependencies within an agent group but also between the group and the outside world. Dependencies are implemented as condition-action pairs, where the condition is defined by means of event types received and the action can be internal to the group or external, in the latter case involving entities existing outside the group.
- A *Group Administrator*, which manages the group as a whole and decides on issues such as the life span of the group or orphan detection.
- A *Results Receiver*, which is an agent collecting the results of the agents forming a group.

The above logical organization of a multi-agent system has been applied by the authors to their system Mole (\{Baumann(b)}) while allows migration of agents and supports a distributed event service.

Dynamic Agents. Dynamic Agents (\{Chen}) is a similar model to the one presented above, based on the dynamic modification of the behaviour of agents. In an ordinary (mobile) agent its behaviour is fixed at the time of agent creation, and in order for this behaviour to change this agent must effectively be replaced by another agent with the newly required behaviour. A dynamic agent, however, is not designed to have a fixed set of predefined functions, but instead to carry application specific actions, which can be loaded and modified on the fly. In that respect, dynamic agents can adjust their capabilities to accommodate changes in the environment and requirements, and play different roles across multiple applications.

Dynamic agents are created by an *agent factory* running on each local site. Each such agent is identified by a symbolic name and an Internet address (including a socket number) which are unique within the boundaries of some *agent space*, itself defined in terms of the agents residing within. The agent space is managed by an *agent coordinator* which maintains the agent name registry of this space. The coordinator is the first agent to be created within an agent space. When it is created, it publishes its socket address to a designated location and in that respect it makes it known to all other dynamic agents. A dynamic agent that is being created, first registers its (unique) name and address with the coordinator. In that respect, any other agent who wishes to send a message to the agent in question and does not know its address, consults the coordinator. The coordinator keeps also an address list for any services offered in its space (e.g. program utilities) and any agent wishing to use some service can again consult the coordinator. The address list of agents is kept up to date, so any agent termination, for instance, results in the name and address of the agent being removed from the list. The coordinator also broadcasts an appropriate message to all the agents in the agent space so that they become aware of the termination of that agent. Hierarchical groups of agent spaces with associated coordinators can also be formed.

Finally, the proposed system offers other types of coordination services in terms of more specialized dynamic agents. In particular, there exist *resource brokers* that provide global resource management services, *request brokers* that provide look up services for service requests, and *event brokers* that manage event-based synchronization between agents.

Role Models. In \{Kendall}, it is argued that a way to express coordination and collaboration relationships between agents is by means of roles. A role model identifies and describes an archetypal or recurring structure of interacting entities in terms of roles. The latter define a position and a set of responsibilities within role models. External interfaces are used to make a role's services and activities accessible. In addition to responsibilities and external interfaces, an agent role comprises a number of other parameters such as collaborators (roles it interacts with), and *coordination and negotiation* information related to communication protocols, conflict resolutions, permissible actions, etc. The author presents a UML-based formal notation (which can be also presented in terms of Patterns) for describing agent roles and shows how executable code can be generated using Aspect Oriented Programming techniques. She further argues that the model can be used at the systems analysis and design phase of developing a multi-agent system.

TRUCE. The TRUCE (\{Jamison}) coordination language can be seen as a concrete realization of some of the above mentioned notions such as roles, groups and dynamic agents. TRUCE is a *scripting* language where scripts define a protocol specification for coordination. Such a script is given to all agents that are members of some collaboration group and they interpret it in a concurrent fashion. An agent receiving a script does not execute it in its entirety, but chooses to execute only that part which is relevant to its activities, as the latter are defined from the *role* that has been assigned to it. Every instruction in a script has two components: an *action* to be executed and a set of *collaborators* that participate in carrying out this action. Every such collaborator has a specific role, e.g. initiate the action, receive the result of the action, etc. Depending on the state of the system, an agent may execute different parts of the script and play different roles, thus exhibiting behaviour similar to that of dynamic agents (see above). The following fragment of code shows the modeling of some coordination scenario:

```
protocol selling-protocol {
  when { $selling=true {
     sellers.if {myturn=true} {
        set $"current-seller"=_me;
     }
     retract {selling-protocol};
     facilitator.set {$selling} false;
     auction {facilitator, $"current-seller"
        {buyers, sellers} };
     recover {selling-protocol};
     facilitator.proceed {$"current-seller"};
```

```
    }
}
```

This code is part of a more elaborate scenario on auctions. Roughly speaking, the protocol is triggered by a global property `$selling` which is set by some `seller` agent. Only sellers test the value of their local parameter `myturn`. They then set `current-seller` to `_me` which causes the temporal suspension of the protocol until the selling has taken place (not shown above) in which case the protocol is re-activated. The roles `facilitator`, `buyers` and `sellers` are bound to agent names according to other parts of the rest of the script.

E-Commerce Mediators. The survey paper by \{Guttman} discusses a rather important type of Internet agents which act as coordinators, the Electronic Commerce *mediators*. The authors define a set of characteristics that these mediators should possess, namely *need identification* that assists the consumer to define precisely his needs, *product brokering* that helps to determine what must be bought, *merchant brokering* that helps to determine where to buy from, *negotiation* that determines the terms of conducting a business transaction, *purchase* and *delivery* of the bought product, and post *analysis* of quality of service. A number of tools and products are then analyzed against these parameters, namely Personal Logic, Firefly, Bargain Finder, Jango, Kasbah, Auction Bot and Tete-a-Tete. According to the authors' survey, only the last model addresses all the requirements they have defined.

5 Conclusions

In this survey chapter we have presented an overview of the various types of models and technologies that enable the use of coordination principles in the development of Internet (multi-) agent systems. As in another survey of similar nature (\{Papadopoulos}) we have advocated a rather liberal approach in what constitutes a coordination framework. In particular, for the case of Internet agents we have identified three broad categories of associated coordination models and languages. The first category comprises those approaches which can be viewed as providing the basic coordination infrastructure. The models in this category do not deal with coordination mechanisms *per se*, but instead they provide the means necessary to build fully-fledged coordination frameworks. We can identify two subcategories here; the first one deals with the most fundamental issue of coordination, namely that of communication. Therefore this subcategory comprises the Agent Communication Languages, where prominent members are KQML and its derivatives. The second subcategory comprises those approaches which provide useful infrastructure to other important aspects of developing Internet (multi-) agent systems such as security or basic mechanisms for building compositional environments.

The second main category presents some fully-fledged coordination models and languages where coordination principles are treated as first class citizens. Historically, many of these models have evolved from more traditional (non-agent-based) versions that have been developed as proposals to advance the Software Engineering techniques for building Parallel and Distributed Information Systems. Here we can also identify two subcategories. The first deals with those approaches that have been inspired from the Linda model of coordination and the use of a Shared Dataspace. Many researchers agree that the concept of having a common forum of communication and cooperation among a number of processes (or agents) is particularly appealing to the case of Internet-based Information Systems. The second subcategory comprises the rest of the proposals, which use some alternative approach.

The final main category is concerned with those approaches which deal with coordination at a higher, logical or algorithmic, level. In this category we review some models whose main aim is to program coordination techniques into the behaviour of the agents that comprise an application. This leads to the creation of specialized types of Internet agents that deal with one or another aspect of inter-agent coordination (contracting, negotiation, etc.), and languages able to express coordination patterns of agent behaviour at a higher level.

Needless to say, the above three-level organization of the presented approaches in this chapter is hardly the only one that has been suggested. There is a number of other survey papers that the interested reader may want to consult. \{Nwana(b)} present a survey on the basic infrastructure technologies for developing agent-based systems with emphasis on the Agent Communication Languages. \{Cabri(a)} present a similar in scope survey where the taxonomy used is based on the criterion as to whether a model is independent or not in time and/or space. \{Gomaa} presents another survey where the main aspect of coordination which is of concern in this work is that of the different cooperation patterns employed by the various systems examined. The author focuses his analysis on the issue of application frameworks suited to each model with particular emphasis to e-commerce ones. A similar survey, with even more emphasis on e-commerce issues, is reported in \{Kerschberg}. However, all these surveys deal with only one aspect of coordination, as this notion is conceived in this chapter. Effectively, \{Nwana(b)} deals with the first level of our taxonomy, \{Cabri(a)} is focused on the second one, and \{Gomaa} and \{Kerschberg} examine models belonging to the third level.

It should be clear from the various trends that have been presented in this chapter, that the notion of coordination is inherent and important in building Internet multi-agent systems. We believe we will see in the future more models and languages that will advance this framework in all three dimensions as we have used them to classify the different approaches. At the lower level, we will see more advanced techniques for dealing with issues of basic inter-agent communication (i.e. more powerful and expressive KQML or otherwise based ACLs), security, etc. The middle level will continue to populate with know-how from mainstream coordination technologies and further associated coordination languages will be proposed. The upper level will also evolve, driven by the needs for coordination patterns from important Internet-based applications such as e-commerce or Cooperative Information Systems.

Finally, one should note that one of the aims of this chapter is to become a roadmap for the more focused and specialized chapters that follow and which shed even more light in the usefulness and importance of using coordination principles in developing Internet-based multi-agent systems.

Acknowledgments

This work has been partially supported by the INCO-DC KIT (Keep-in-Touch) program 962144 “Developing Software Engineering Environments for Distributed Information Systems” financed by the Commission of the European Union.

References

Aglets, IBM, http://www.trl.ibm.co.jp/aglets/index.html.

Concordia, Mitsubishi Electric, http://www.meitca.com/HSL/Projects/Concordia/Welcome.html.

F. Barbosa and J. C. Cunha, “A Coordination Language for Collective Agent Based Systems: GroupLog”, *ACM SAC 2000: Special Track on Coordination Models, Languages and Applications*, Como, Italy, 19-21 April, 2000, pp. 189-195.

M. Barbuceanu and M. S. Fox, “Capturing and Modeling Coordination Knowledge for Multi Agent Systems”, *International Journal on Cooperative Information Systems*, World Scientific, Vol. 5 (2 & 3), 1996, pp. 275-314.

J. Baumann and N. Radouniklis, “Agent Groups in Mobile Agent Systems”, *IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*, Cottbus , Germany, 30 Sept. - 2 Oct., 1997, Chapman & Hall.

J. Baumann, F. Hohl, K. Rothermel and M. Straßer, “Mole - Concepts of a Mobile Agent System, *World Wide Web*, Vol. 1 (3), 1998, pp. 123-137.

G. Cabri, L. Leonardi, and F. Zambonelli, “Coordination Models for Internet Applications Based on Mobile Agents”, *IEEE Computer Magazine*, IEEE Computer Society Press, 1999.

G. Cabri, L. Leonardi, and F. Zambonelli, “XML Dataspaces for Mobile Agent Coordination”, *ACM SAC 2000: Special Track on Coordination Models, Languages and Applications*, Como, Italy, 19-21 April, 2000, pp. 181-188.

Q. Chen, P. Chundi, U. Dayal and M. Hsu, “Dynamic Agents”, *International Journal on Cooperative Information Systems*, World Scientific, Vol. 8 (2 & 3), 1999, pp. 195-223.

P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi and A. Knoche, “Coordinating Multi Agent Application on the WWW: a Reference Architecture”, *IEEE Transactions on Software Engineering*, IEEE Computer Society Press, 1998, Vol. 1 (2), pp. 87-99.

M. Cremonini, A. Omicini and F. Zambonelli, “Multi Agent Systems on the Internet: Extending the Scope of Coordination Towards Security and Topology”, *Ninth European Workshop on Modeling Autonomous Agents in a Multi Agent World (MAAMAW’90)*, Valencia, Spain, 30 June - 2 July, 1999, LNAI 1647, Springer Verlag, pp. 77-88.

L. R. Crow and N. R. Shadbolt, “IMPS - Internet Agents for Knowledge Engineering” *Eleventh Workshop on Knowledge Acquisition, Modeling and Management (KAW'98)*, SRDG Publications, Calgary, 18-23 April, 1998, http://ksi.cpsc.ucalgary.ca/KAW/KAW98/KAW98Proc.html.

W. H. E. Davies and P. Edwards, *Agent K: An Integration of APO and KQML*, AUCS/TR9406, Department of Computer Science, University of Aberdeen, 1994.

J. Eriksson, N. Finne and S. Janson, “SICS MarketSpace — An Agent based Market Infrastructure”, *First International Workshop on Agent-Mediated Electronic Trading (AMET-98)*, Minneapolis, MN, USA, 10 May, 1998, LNAI 1571, Springer Verlag.

T. Finin, R. Fritzson, D. MaKay and R. McEntire, “KQML as an Agent Communication Language”, *Third International Conference on Information and Knowledge Management (CIKM’94)*, New York, USA, 29 Nov. - 2 Dec., 1994, ACM Press, pp. 456-463.

R. A. Flores-Mendez, “Towards the Standardization of Multi Agent Systems Architectures: An Overview”, *ACM Crossroads - Special Issue on Intelligence Agents*, Vol. 5 (4), ACM Press, Summer, 1999.

M. Fuchs, “Domain Specific Languages for ad hoc Distributed Applications”, *USENIX Conference on Domain Specific Languages*, Santa Barbara, USA, 15-17 Oct., 1997.

J. A. Giampapa, M. Paolucci and K. Sycara, “Agent Interoperation Across Ulti-agent System Boundaries”, *Fourth International Conference on Autonomous Agents (Agents 2000)*, Barcelona, Spain, 3-7 June, 2000, ACM Press (to appear).

H. Gomaa, “Inter-Agent Communication in Cooperative Information Agent based Systems”, *Third International Workshop on Cooperative Information Agents (CIA’99)*, Uppsala, Sweden, 31 July - 2 Aug., 1999, LNAI 1652, Springer Verlag, pp. 137-148.

T. Gschwind, M. Feridun and S. Pleisch, “ADK — Building Mobile Agents for Network and Systems Management from Reusable Components”, *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA’99 & MA’99)*, 3-6 Oct., 1999, Palm Springs, California, USA, IEEE Press, pp. 13-21.

R. Guttman, A. Moukas and P. Maes. “Agent-Mediated Electronic Commerce: A Survey”, *Knowledge Engineering Review*, Cambridge University Press, Vol. 13 (3), 1998, pp. 147-160.

W. C. Jamison and D. Lea, “TRUCE: Agent Coordination Through Concurent Interpretation of Role based Protocols”, *Third International Conference on Coordination Languages and Models (Coordination’99)*, Amsterdam, The Netherlands, 26-28 April, 1999, LNCS 1594, Springer Verlag, pp. 384-398.

JavaSpaces, Sun Microsystems, http://www.javasoft.com/products/javaspaces/index.html.

D. Jensen, Y. Dong, B. S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton, Jr. and A. Wise, “Coordinating Agent Activities in Knowledge Discovery Processes”, *International Joint Conference on Work Activities Coordination and Collaboration (WACC’99)*, San Francisco, CA, USA, 22-25 Feb., 1999, pp. 137-146.

Jini, Sun Microsystems, http://java.sun.com/products/jini.

E. A. Kendall, “Role Modeling for Agent System Analysis, Design and Implementation”, *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA’99 & MA’99)*, 3-6 Oct., 1999, Palm Springs, California, USA, IEEE Computer Society Press, pp. 204-218.

L. Kerschberg and S. Banerjee, “An Agency based Framework for Electronic Business”, *Third International Workshop on Cooperative Information Agents (CIA’99)*, Uppsala, Sweden, 31 July - 2 Aug., 1999, LNAI 1652, Springer Verlag, pp. 265-290.

G. Kotsis and G. Neumann (eds.), *IEEE Eight International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE’99)*, Stanford, USA, 16-18 June, 1999, IEEE Press.

T. W. Malone and K. Crowston, “The Interdisciplinary Study of Coordination”, *ACM Computing Surveys* **26**, 1994, pp. 87-119.

F. G. McCabe and K. Clark, “April — Agent PRocess Interaction Language”, *Intelligent Agents — Theories, Architectures, and Languages*, LNAI 890, Springer Verlag, 1995, pp. 324-340.

N. H. Minsky, Y. M. Minsky and V. Ungureanu, “Making Tuple Spaces Safe for Heterogeneous Distributed Systems”, *ACM SAC 2000: Special Track on Coordination Models, Languages and Applications*, Como, Italy, 19-21 April, 2000, pp. 218-226.

R. De Nicola, G. Ferrari and R. Pugliese, “KLAIM: a Kernel Language for Agent Interaction and Mobility”, *IEEE Transactions on Software Engineering*, IEEE Computer Society Press, Vol. 24 (5), 1998, pp. 315-330.

H. S. Nwana, L. Lee and N. R. Jennings, “Coordination in Multi Agent Systems”, *Software Agents and Soft Computing: Towards Enhancing Machine Intelligence*, LNAI 1198, Springer Verlag, 1997, pp. 42-58.

H. S. Nwana and M. Wooldridge, “Software Agent Technologies”, *Software Agents and Soft Computing:Towards Enhancing Machine Intelligence*, LNAI 1198, Springer Verlag, 1997, pp. 59-78.

Odyssey, General Magic, http://www.genmagic.com/technology/odyssey.html.

G. A. Papadopoulos and F. Arbab, “Coordination Models and Languages”, *Advances in Computers*, Marvin V. Zelkowitz (ed), Academic Press, Vol. 46, August, 1998, pp. 329-400.

C. J. Petrie, “Agent based Engineering, the Web, and Intelligence”, *IEEE Expert*, Vol. 11 (6), Dec. 1996, IEEE Computer Society Press, pp. 24-29.

G. P. Picco, A. L. Murphy and G-C. Roman, “LIME: Linda Meets Mobility”, *Twenty First International Conference on Software Engineering (ICSE’99)*, Los Angeles, California, USA, 16-22 May, 1999, IEEE Computer Society Press.

M. Ranganathan, V. Schaal, V. Galtier and D. Montgomery, “Mobile Streams: A Middleware for Reconfigurable Distributed Scripting”, *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA’99 & MA’99)*, 3-6 Oct., 1999, Palm Springs, California, USA, IEEE Computer Society Press, pp. 162-175.

D. Rossi, *Coordination: an Enabling Technology for the Internet*, Ph.D. Thesis, University of Bologna, 1999.

Y. Shoham, “Agent Programming Languages”, *Artificial Intelligence* ***60 (1)***, 1993, pp. 51-92.

N. Skarmeas and Keith L. Clark, “Component Based Agent Construction”, *Autonomous Agents and Multi Agent Systems*, 1999 (submitted).

R. Sukthankar, A. Brusseau, R. Pelletier and R. Stockton, “JGram: Rapid Development of Multi Agent Pipelines for Real-World Tasks”, *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA’99 & MA’99)*, 3-6 Oct., 1999, Palm Springs, California, USA, IEEE Computer Society Press, pp. 30-40.

M. Schumacher, F. Chantemargue and B. Hirsbrunner, “The STL++ Coordination Language: A Base for Implementing Distributed Multi Agent Systems”, *Third International Conference on Coordination Languages and Models (Coordination’99)*, Amsterdam, The Netherlands, 26-28 April, 1999, LNCS 1594, Springer Verlag, pp. 399-414.

S. R. Thomas, “The PLACA Agent Programming Language”, *Intelligent Agents — Theories, Architectures, and Languages*, LNAI 890, Springer Verlag, 1995, pp. 355-370.

R. Tolksdorf, “Berlinda: An Object-Oriented Platform for Implementing Coordination Languages in Java”, *Second International Conference on Coordination Languages and Models (Coordination’97)*, Berlin, Germany, 1-3 Sept., 1997, LNCS 1282, Springer Verlag, pp. 430-433.

Voyager, ObjectSpace, http://www.objectspace/com/products/voyager1.htm.