Modelling Electronic Commerce Activities Using Control-Driven Coordination

George A. Papadopoulos

Department of Computer Science
University of Cyprus
75 Kallipoleos Str, P.O.B. 537
CY-1678 Nicosia, Cyprus
E-mail: george@cs.ucy.ac.cy

Farhad Arbab

Department of Software Engineering
CWI
Kruislaan 413, 1098 SJ Amsterdam
The Netherlands
E-mail: farhad@cwi.nl

Abstract

Modern Electronic Commerce environments are heavily web-based and involve issues such as distributed execution, multiuser interactive access or interface with and use of middleware platforms. Thus, their components exhibit the properties of communication, cooperation and coordination as in CSCW, groupware or workflow management systems. In this paper we examine the potential of using coordination technology to model Electronic Commerce activities and we show the benefits of such an approach. Furthermore, we argue that control-oriented, event-driven coordination models (which enjoy some inherent properties such as security) are more suitable for Electronic Commerce than data-driven ones which are based on accessing an open shared communication medium in almost unrestricted ways.

Keywords: Coordination Models and Languages; Web-based Applications; Electronic Commerce.

1. Introduction

Modelling of activities within an information system or between different information systems has become a complex task. Performing these activities (often known as *groupware, workflow, electronic commerce* and *enterprize reengineering*) is often done in conjunction with computer-based cooperative environments such as electronic mail, voice and video teleconferencing, electronic classrooms, etc. In addition, the emergence of the World Wide Web as the main medium, not only for passive presentation of information but also for active cooperation between different agents collaborating in a single task, further enhances some properties of those activities such as distribution and openess. Typical examples of such complex-in-nature activities range from finding suitable time-slots and locations for group meetings, to performing administrative procedures (e.g. organising conferences), to carrying out reviews of draft documents, to developing distributed web-based electronic commerce applications (e.g. reserving flight seats and hotel rooms by means of dedicated WWW servers). Modelling these activities has become a task which is often impossible to perform by single individuals, requiring groups of people, sometimes distributed over different organisations, countries, etc.

Recently, we have seen a proliferation of so-called *coordination models* and their associated programming languages ([2,6,15]). Coordination programming provides a new perspective on constructing computer software. Instead of developing a computer program from scratch, coordination models allow the gluing together of existing components. Coordination, as a science in its own right whose role goes beyond computer programming, has also been proposed ([10]). More to the point, it is argued that coordination has a number of advantages over traditional process models, such as explicit representation of organisational goals, constraints and dependencies (as opposed to "compiled" process descriptions), opportunistic selection of required mechanisms given current coordination requirements (as opposed to having fully-defined processes ahead of time), and sensitivity to exception handling as well as the ability to adapt dynamically (as opposed to having processes with rigid, well-defined behaviour).

In this paper we use the generic coordination model IWIM (Ideal Worker Ideal Manager) and a specific *control-oriented event-driven* coordination language (MANIFOLD) based on IWIM ([3,4]) to model Electronic Commerce activities. Electronic Commerce makes heavy use of all aspects related to coordination technologies, namely communication (between, say, sellers and potential customers), cooperation (as in the case of brokering) or coordination (as in the case of distributed auction bidding). Furthermore, web-based Electronic Commerce environments are inherently distributed and require support for security measures. IWIM and its associated language MANIFOLD are based on point-to-point communication and are therefore inherently secured coordination systems, as opposed to the category of Shared Dataspace coordination models which are inherently weaker in security aspects (see the following section).

The rest of this paper is organised as follows: in section 2 we briefly compare the two main approaches to developing coordination models and languages. In section 3 we describe the coordination model IWIM and its associated language MANIFOLD. In section 4 we use MANIFOLD to model Electronic Commerce activities, and, finally, in section 5 we present some conclusions, and related and further work.

2. Data- vs control-driven coordination models and languages

Over the past few years a number of coordination models and languages have been developed ([2,6,15]). However, the first such model, which still remains the most popular one, is Linda ([1]). In Linda, the underlying view of the system to be coordinated (which is usually distributed and open) is that of an asynchronous ensemble formed by *agents* where the latter perform their activities independently from each other and their coordination is achieved via some medium in an asynchronous manner. Linda introduces the so-called notion of *uncoupled communication* whereby the agents in question either insert to or retrieve from the shared medium the data to be exchanged between them. This shared dataspace is referred to as the *Tuple Space* and information exchange between agents via the Tuple Space is performed by posting and retrieving *tuples*. Tuples are addressed *associatively* by suitable patterns used to match one or more tuples. In general, the tuples produced do not carry any information regarding the identity of their producers or intended consumers, so communication is *anonymous*.

Although Linda is indeed a successful coordination model, it has some potentially serious deficiencies (at least for some applications such as Electronic Commerce) which penetrate to all other related models that are based on it. These deficiencies are:

- It is data-driven. The state of an agent is defined in terms of what kind of data it posts to or retrieves from the Tuple Space. This is not very natural when we are interested more in how the *flow of information* between the involved agents is set-up and how an agent *reacts* to receiving some information, rather than *what kind of data* it sends or receives.
- The shared dataspace through which all agents communicate may be intuitive when ordinary parallel programming is concerned (offering easy to understand and use metaphors such as the one of shared memory), but we believe that it is hardly intuitive or realistic in other cases, such as for modelling organisational activities. People in working environments do not take the work to be done by others to common rooms where from other people pass by and pick the work up! It is true that sometimes there is *selective broadcasting* (e.g. in providing a group of people doing the same job with some work and letting them sort out the workload among themselves) but the unrestricted broadcasting that the Tuple Space and its variants suggest and enforce is hardly appropriate and leads to unneccesary efficiency overheads.
- Furthermore, and perhaps more importantly, the use of such a widely public medium as the Tuple Space and its variants, suffers inherently from a major *security* problem which gives rise to problems in at least three dimensions related to the fate of the data posted there: (i) they can be seen and examined by anyone; (ii) they can be removed by the wrong agent (intentionally or unintentionally); and even worse, (iii) they can be forged without anyone noticing it. The repercussions of these deficiencies in modelling information systems are rather obvious and need not be discussed any further. It suffices to say, as an example directly related to the context of this paper, that we would not want to broadcast to the tuplespace our credit card number hoping that it will be picked up by the intended recipient.

Some of the above problems have already been of concern to researchers in the area of shared-dataspace-based coordination models and solutions have been sought ([7,11,15]). Nevertheless, implementing these solutions requires quite some extra effort and effectively leads to the design of new coordination models on top of the "vanilla" type ones; these new models are often counter-intuitive and relatively complex when compared with the inherent philosophy of their underlying basic model.

3. The IWIM model and the language MANIFOLD

MANIFOLD ([4]) is a coordination language which, as opposed to the Linda family of coordination models described in the previous section, is control- (rather than data-) driven, and is a realisation of a new type of coordination models, namely the Ideal Worker Ideal Manager (IWIM) one ([3]). In MANIFOLD there are two different types of processes: *managers* (or *coordinators*) and *workers*. A manager is responsible for setting up and taking care of the communication needs of the group of worker processes it controls (non-exclusively). A worker on the other hand is completely unaware of who (if anyone) needs the results it computes or from where it itself receives the data to process. MANIFOLD possess the following characteristics:

- *Processes*. A process is a *black box* with well defined *ports* of connection through which it exchanges *units* of information with the rest of the world. A process can be either a manager (coordinator) process or a worker. A manager process is responsible for setting up and managing the computation performed by a group of workers. Note that worker processes can themselves be managers of subgroups of other processes and that more than one manager can coordinate a worker's activities as a member of different subgroups. The bottom line in this hierarchy is *atomic* processes which may in fact be written in any programming language.
- *Ports*. These are named openings in the boundary walls of a process through which units of information are exchanged using standard I/O type primitives analogous to read and write. Without loss of generality, we assume that each port is used for the exchange of information in only one direction: either into (*input* port) or out of (*output* port) a process. We use the notation `p.i` to refer to the port `i` of a process instance `p`.
- *Streams*. These are the means by which interconnections between the ports of processes are realised. A stream connects a (port of a) producer (process) to a (port of a) consumer (process). We write `p.o -> q.i` to denote a stream connecting the port `o` of a producer process `p` to the port `i` of a consumer process `q`.
- *Events*. Independent of streams, there is also an event mechanism for information exchange. Events are broadcast by their sources in the environment, yielding *event occurrences*. In principle, any process in the environment can pick up a broadcast event; in practice though, usually only a subset of the potential receivers is interested in an event occurrence. We say that these processes are *tuned in*

to the sources of the events they receive. We write `e.p` to refer to the event `e` raised by a source `p`.

Activity in a MANIFOLD configuration is *event driven*. A coordinator process waits to observe an occurrence of some specific event (usually raised by a worker process it coordinates) which triggers it to enter a certain *state* and perform some actions. These actions typically consist of setting up or breaking off connections of ports and channels. It then remains in that state until it observes the occurrence of some other event which causes the *preemption* of the current state in favour of a new one corresponding to that event. Once an event has been raised, its source generally continues with its activities, while the event occurrence propagates through the environment independently and is observed (if at all) by the other processes according to each observer's own sense of priorities. Figure 1 below shows diagramatically the infrastructure of a MANIFOLD process.

The process `p` has two input ports (`in1`, `in2`) and an output one (`out`). Two input streams (`s1`, `s2`) are connected to `in1` and another one (`s3`) to `in2` delivering input data to `p`. Furthermore, `p` itself produces data which via the `out` port are replicated to all outgoing streams (`s4`, `s5`). Finally, `p` observes the occurrence of the events `e1` and `e2` while it can itself raise the events `e3` and `e4`. Note that `p` need not know anything else about the environment within which it functions (i.e. who is sending it data, to whom it itself sends data, etc.).

The following is a MANIFOLD program that computes the Fibonacci series.

```
manifold PrintUnits() import.
manifold variable(port in) import.
manifold sum(event)
  port in x.
  port in y.
  import.
event overflow.

auto process v0 is variable(0).
auto process v1 is variable(1).
auto process print is PrintUnits.
auto process sigma is sum(overflow).

manifold Main()
{
 begin:(v0->sigma.x, v1->sigma.y,v1->v0,
        sigma->v1,sigma->print).
 overflow.sigma:halt.
}
```

The above code defines `sigma` as an instance of some predefined process `sum` with two input ports (`x`,`y`) and a default output one. The main part of the program sets up the network where the initial values (`0`,`1`) are fed into the network by means of two "variables" (`v0`,`v1`). The continuous generation of the series is realised by feeding the output of `sigma` back to itself via `v0` and `v1`. Note that in MANIFOLD there are no variables (or constants for that matter) as such. A MANIFOLD variable is a rather simple process that forwards whatever input it receives via its input port to all streams connected to its output port. A variable "assignment" is realised by feeding the contents of an output port into its input. Note also that computation will end when the event `overflow` is raised by `sigma`. `Main` will then get preempted from its `begin` state and make a transition to the `overflow` state and subsequently terminate by executing `halt`. Preemption of `Main` from its `begin` state causes the breaking of the stream connections; the processes involved in the network will then detect the breaking of their incoming streams and will also terminate.

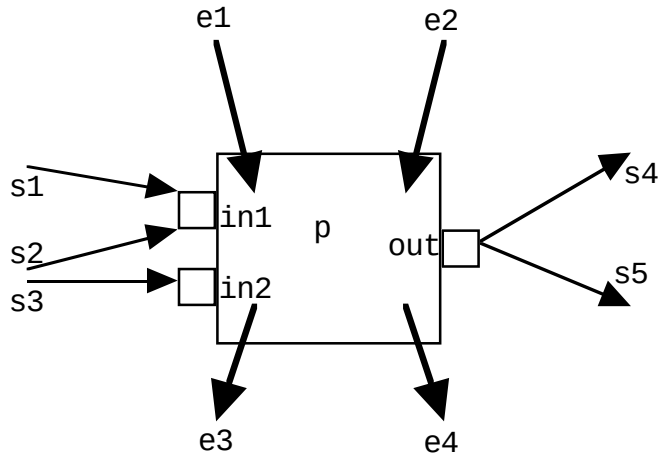

Fig. 1

4. Modelling Electronic Commerce activities in MANIFOLD

In this section we show how a control-based event-driven coordination model like MANIFOLD can be used to model transactions for Electronic Commerce. We start with some simple cases (namely advertising and advising) and we end up with a more complicated scenario (an actual sell-buy transactions). In the process we take the opportunity to introduce some additional features of MANIFOLD.

The following MANIFOLD coordinator models an advertisement scenario; more to the point the coordinator `Advertiser` raises an event signifying its intention to offer a product and additionally it provides the product's description for any potential buyer to examine.

```
event sell_house.

manifold Advertiser()
{
 begin: (raise(sell_house),
         <<House_Description>> -> output).
}
```

Note that `Advertiser` places a tuple with the description of the house to its output port; any potential buyer can connect to `Advertiser.output` and retrieve the description. This will become apparent below when we describe a fully fledged transaction.

The next example involves the coordinator `Advisor` which can be seen as playing the role of an intermediary between a seller and a potential buyer. The idea here is that `Advisor` will first examine the specifications of an offered product (typically such an offer will be coming from an

Advertiser) and will contact the potential buyer only if the specifications satisfy the buyer's needs.

```
manifold Advisor (process buyer)
{
 auto process advise is CheckSpecs(buyer.specs).

 begin: terminated(self).
 sell_house.*advertiser:
    { begin: (getunit(advertiser) -> advise,
              terminated(self)).
      recommend.advise: getunit(advertiser) ->
    }.
 next: post (begin).
}
```

Advisor takes as a parameter a potential buyer; initially it is suspended waiting for the raising of the signal (such a suspension is achieved by means of the construct terminated(self)). Upon observing the raising of the event sell_house (from some process of type Advertiser), it retrieves the specification of the offered sale and passes it on to the atomic process advise (an instant of CheckSpecs - such an atomic process is implemented in some other than the MANIFOLD language). The advise process verifies the specification produced by advertiser against the requirements of buyer. Advisor then suspends waiting for advise to decide as to whether buyer may be interested in the offered sale; if this is the case, it connects advertiser to buyer for the rest. If advise decides that the offer is not appropriate, it raises next instead, which starts the wait for another offer.

We now present a more complicated transaction. In particular, we consider the case of a general scenario whereby sellers and potential buyers are exchanging control and data information as follows:

- A seller can raise the event offer_service whereby it informs the market of some product that it is able to offer (for simplicity, we assume here that the seller in question can offer just one product whose nature is self-evident by the event that is being raised — this, certainly, need not be the case, and the seller may be offering more than one product). In addition to raising this event, the seller places the tuple <<Prod_Desc>> with detailed description of the offered ptoduct to its default output port.
- A potential buyer detects the raising of the event, and if interested, uses the id of the event's sender to connect to the seller's output port in order to retrieve the detailed description of the offered product (here we use the atomic process propose, an instant of CheckDescr, which decides as to whether there is interest in continuing the transaction activities). Then, if it decides to buy, it raises the event i_am_interested (again for simplicity we assume that the event's meaning is self-evident in the sense that no other seller exists and there can be no confusion as to the intention of the potential buyer — we point out once more that this need not be the case and our model can handle arbitrarily complex transaction patterns).
- Upon detecting the presence of the event i_am_interested, the seller uses the event's source id to connect to the default input port of the potential buyer and place there his detailed offer (including, perhaps, discounts, special prices, etc.).
- The potential buyer decides as to whether he wishes to complete the transaction or abort it (here we use the atomic process CheckSpecs process introduced above while describing Advisor) and sends the appropriate accept or reject message to the seller, in the former case possibly along with some further information (e.g. his credit card number).
- If a reject message is sent, the transaction process is aborted. If, instead, an accept message is sent (possibly along with some verification information), the buyer sends the product to the user. Finally, the user sends the buyer the required amount of money.

The above scenario is presented graphically below. It is interesting to point out that figure 2 comes very close to being the visual coordination program that would be written in the visual interface of MANIFOLD, namely Visifold ([5]). This suggests the use of visual programming in modelling Electronic Commerce scenarios.

We should stress the point that, by virtue of the IWIM model, the transactions are secured. In particular, the agents involved in the transaction (namely the seller and the potential buyer) broadcast only their intention of selling something and their intention of possibly buying something respectively. The rest of the information involved in the transaction, i.e. the description of the product, the particular offer that the seller may make to the potential buyer and the acceptance of the offer by the buyer along with possibly sensitive information such as a credit card number, are exchanged between them by means of point-to-point port connections, which are by default secure, private and reliable.

The actual MANIFOLD code for a seller and a potential byer is shown below.

```
event offer_service, i_am_interested.

manifold Seller()
{
 event got_answer, got_money.
 begin: (raise(offer_service),
         <<Prod_Descr>> -> output,
         terminated(self)).
 i_am_interested.*buyer:
    { begin: <<Proposal>> -> buyer;
             if (input==<<Accept>>
                then (<<product>> -> buyer,
                      buyer -> payment).
    }.
}
```

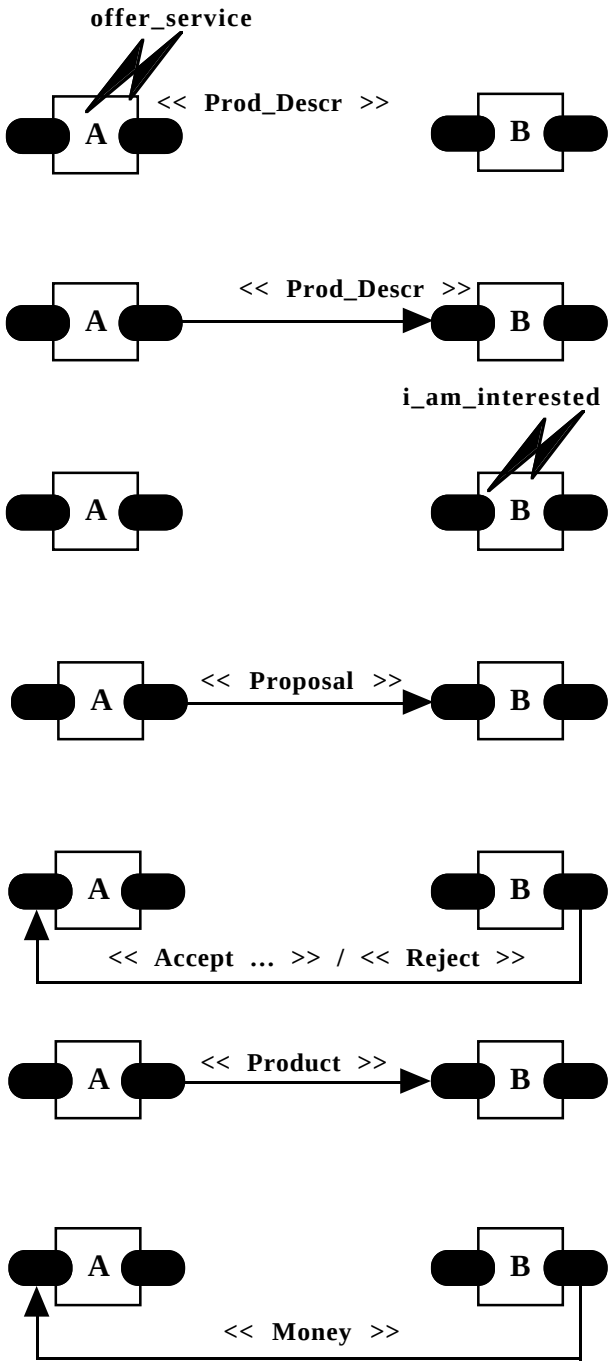

Fig. 2

```
manifold Buyer (port in itemspeecs)
{
 port out specs.
 stream KK *-> specs.
 auto process myspecs is variable(itemspecs).

 /* check product's description */
 auto process propose is CheckDescr().

 /* check product's specs */
auto process advise is CheckSpecs(myspecs).

 begin: (variable(itemspecs) -> specs,
         terminated(self)).
 offer_service.*seller:
    { begin: (getunit(seller) -> propose,
              terminated(self)).
      continue.propose: (raise(i_am_interested),
                         getunit(input) -> advise,
                          terminated(self)).
      recommend.advise: (<<Accept …>> -> seller,
                      getunit(input) -> receiver);
                        <<Money>> -> seller..
      got_product: <<Money>> -> seller.
    }.
 next: post(begin).
}
```

We should probably stress here the fact that the actual information and particular heroes of our scenario are *parametric*. In other words, the code specifies and implements, in a well-defined way, the coordination protocol of the transaction, paying attention to important issues such as security and anonymous communication (by virtue of the IWIM model) but also paying little attention to what is being offered, who is offering it or is interested in buying it, and in the case of the purchase actualy taking place, how the buyer pays. Thus, the protocol is *reusable* and can be applied to many similar cases, combined with other protocols to form more general and complicated ones, etc.

5. Conclusions, related and further work

In this paper we have examined the use of a control-oriented, event-driven coordination mechanism (namely the IWIM model and its associated language MANIFOLD) in modelling Electronic Commerce activities. We believe an Electronic Commerce framework based on MANIFOLD enjoys a number of desirable properties such as natural distribution, hiding of lower level details, exploitation of high-performance computational resources and secure communication without compromising the *flexibility* and *openess* that any such environment should support. Our approach allows for the formation of *generic* coordination patterns for Electronic Commerce transactions which can be used for many cases irrespective of the types of potential sellers and buyers, offered services and products, etc. Coordination languages like MANIFOLD support complete decoupling in both time and space; i.e., agents send information without worrying as to who (if anyone at all) receives this information, while other agents receive information without worrying who has sent it or whether the sender is still alive. Thus, it is possible to introduce new players to a coordination protocol for some Electronic Commerce transaction, enhance or replace existing offered services, etc. Furthermore, the use of coordination technology along the lines described in this paper, is orthogonal to many other issues relevant to the case of Electronic Commerce. More to the point, the work here can be combined with work on *intelligent agents* (typically used to offer customer support in finding and selecting the most appropriate service) to derive coordination protocols where each MANIFOLD process behaves as such an agent, dedicated to perform some

particular task. Furthermore, atomic processes (i.e. processes not written in MANIFOLD due to their involvement with aspects not directly related to the coordination protocols) can be as elaborate as necessary without further complicating the communication protocols. For instance, `CheckDescr` or `CheckSpecs` (and other similar processes) could actually be interfacing to a sophisticated *knowledge base* or use constraint satisfaction techniques, in order to reach any decisions. On another front, MANIFOLD processes can be seen as mobile agents, migrating from one place to another in order to be as efficient as possible and also exploit the underlying hardware infrastructure. We are currently designing a more elaborate environment for Electronic Commerce based on the principles described in this paper.

Furthermore, MANIFOLD coordinators can be used in a somewhat different manner, whereby in addition to the security at the implementation level, we also enjoy security at the logical level. This can be achieved by having special MANIFOLD coordinators which are used as interfaces between the actual agents (themselves being possibly other MANIFOLD coordinators). An agent, say a seller or a buyer, cannot arbitrarily communicate with some other agent but instead it will have to ask for permission a special coordinator; the latter may allow the communication to continue or it may itself do it on behalf of the agent that requested it. Thus, these special coordinators which interpose themselves between an agent and the rest of the world, regulate the behaviour of the agents and provide logical security. These coordinators can be seen as *law enforcers* where the law itself is defined and implemented by their MANIFOLD code; the idea of special law enforcing agents has been introduced in [11,12] and we are currently implementing them in MANIFOLD.

Our paper complements (initial) work by others in the use of coordination models for modelling Electronic Commerce activities. More to the point, [8] describes such a model based on the Linda coordination framework. As we have argued in this paper and elsewhere ([13,14,15]), the (vanilla) Linda formalism is based on the use of an open and public shared communication medium (in the case of [8] this is the PageSpace) where access for either placing or retrieving information is almost unrestricted. Thus, the basic model, is inherently insecure and extra devices must be built on top of it. The same can be said about the work presented in [9] where Prolog is used to model agents communicating via MarketSpace, a medium very similar to Linda's tuplespace. On the other hand, our framework is based on secured (by virtue of the underlying IWIM model) point-to-point communications with broadcasting limited only to publisizing the most necessary information. Finally, we mention again the work reported in [12], and we note that the laws described there can be implemented in MANIFOLD and, thus, can be used to complement the work described in this paper in deriving a framework based on MANIFOLD that enjoys security at both the logical and implementation levels.

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, and also by project ERIS financed by the University of Cyprus.

References

[1] S. Ahuja, N. Carriero and D. Gelernter, "Linda and Friends", *IEEE Computer* **19(8)**, Aug. 1986, pp. 26-34.

[2] J-M. Andreoli, C. Hankin and D. Le Métayer, *Coordination Programming: Mechanisms, Models and Semantics*, World Scientific, 1996.

[3] F. Arbab, "The IWIM Model for Coordination of Concurent Activities", *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 34-56.

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

[5] P. Bouvry and F. Arbab, "Visifold: A Visual Environment for a Coordination Language", *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 403-406.

[6] N. Carriero and D. Gelernter, "Coordination Languages and their Significance", *Communications of the ACM* **35 (2)**, 1992, pp. 97-107.

[7] N. Carriero, D. Gelernter and S. Hupfer, "Collaborative Applications Experience with the Bauhaus Coordination Language", *30th Hawaii International Conference on Systems Sciences (HICSS-30)*, Mauni, Hawaii, 7-10 Jan., 1997, IEEE Press, pp. 310-319.

[8] P. Ciancarini and D. Rossi, "Coordinating Distributed Applets with Shade/Jada", *13th ACM Symposium on Applied Computing (SAC'98)*, Atlanta, Georgia, U.S.A., 27 Feb. - 1 March, 1998, ACM Press, pp. 130-138.

[9] J. Eriksson, N. Finne and S. Janson, "Surfing the Market and Making Sense on the Web: Interfacing the Web to an Open Agent-Based Market Infrastructure", *Workshop on Programming the Web - a Search for APIs", 5th International WWW Conference*, Paris, France, May, 1996.

[10] T. W. Malone and K. Crowston, "The Interdisciplinary Study of Coordination", *ACM Computing Surveys* **26**, 1994, pp. 87-119.

[11] N. H. Minsky and J. Leichter, "Law-Governed Linda as a Coordination Model", *Object-Based Models and Languages for Concurrent Systems*, Bologna, Italy, 5 July, 1994, LNCS 924, Springer Verlag, pp. 125-145.

[12] N. H. Minsky and V. Ungureanu, "A Mechanism for Establishing Policies for Electronic Commerce", *18th International Conference on Distributed Computing Systems (ICDCS'98)*, Amsterdam, The Netherlands, 26-29 May, 1998, IEEE Press (to appear).

[13] G. A. Papadopoulos and F. Arbab, "Control-Based Coordination of Human and Other Activities in Cooperative Information Systems", *Second International Conference on Coordination Models and Languages*, 1-3 Sept., 1997, Berlin, Germany, LNCS, Springer Verlag, pp. 422-425.

[14] G. A. Papadopoulos and F. Arbab, "Modelling Activities in Information Systems Using the Coordination Language MANIFOLD", *13 ACM Symposium on Applied Computing (SAC'98)*, Atlanta, Georgia, U.S.A., 27 Feb. - 1 March, 1998, ACM Press, pp. 185-193.

[15] G. A. Papadopoulos and F. Arbab, "Coordination Models and Languages", *Advances in Computers* **46**, Academic Press, August, 1998 (to appear).