MODELLING ACTIVITIES IN INFORMATION SYSTEMS USING THE COORDINATION LANGUAGE MANIFOLD

George A. Papadopoulos

Department of Computer Science
University of Cyprus
75 Kallipoleos Str., P.O.B. 537
CY-1678, Nicosia, Cyprus

george@turing.cs.ucy.ac.cy

Farhad Arbab

Department of Software Engineering
CWI
Kruislaan 413, 1098 SJ Amsterdam
The Netherlands

farhad@cwi.nl

Keywords: Coordination Languages and Models; Modelling Information Systems; Design of Distributed and Open Information Systems; Collaborative Computing Environments.

ABSTRACT

We argue for the need to use *control-based, event-driven* and *state-defined,* coordination models and associated languages in modelling at least some of those activities that constitute the functionality of a modem, open and distributed information system. We illustrate our points by using such a coordination model and language and showing by means of suitable examples how it can be used for the above mentioned purpose. We also compare our approach with the (rather few) approaches that as yet exist in this rather new area of information systems, and we discuss our model's advantages over those other approaches.

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 (eg., organising conferences), to carrying out reviews of draft documents, to developing distributed web-based electronic commerce applications (eg., reserving flight seats and hotel rooms by means of dedicated WWW servers). Modelling these activities has become a task which often is not possible to perform by single persons, but by groups of people, often even distributed over different organisations, countries, etc.

Recently, we have seen a proliferation of so-called *coordination models* and associated programming languages ([2,8]). Coordination programming provides a new perspective on constructing computer programs. Instead of developing a computer program from scratch, the coordination model allows the gluing together of existing components. Whereas in ordinary programming languages a programmer describes individual computing components, in a coordination language the programmer describes interrelationships between collaborating but otherwise independent components. These components may even be written in different programming languages or run on heterogeneous architectures. Thus, the computational parts comprising a computer program are treated by the coordination component as black boxes whose internal constituents and behaviour is of no concern to the coordination framework, which is interested only in the components' external interaction with others.

Coordination, as a science of its own whose role goes beyond computer programming, has also been proposed ([13]). In [12] for instance, 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 ability to adapt dynamically (as opposed to having processes with rigid, well-defined behaviour).

Nevertheless, using the notion of coordination models and languages in modelling the hybrid complex activities of information systems, the so called *coordination language-based approach to groupware construction* ([9]), is a rather new area of research. [Although, as it has been argued elsewhere, coordination in general has already been used (even implicitly sometimes) to model aspects that define partially information systems such as human behaviour or object-oriented systems. But these approaches generally concentrate on addressing specific issues and the produced environments (such as OVAL, PAGES, Lotus Notes or Conversation Builder) do not address the whole spectrum of modelling information systems.] Using such a coordination model and language has some clear advantages:

- Work can be decomposed into smaller steps.
- Steps can be assigned to and performed by various people and tools.
- Execution of steps can be coordinated (eg., in time).
- Coordination patterns that have proved successful for some specific scenario can be reused in other similar situations.
- Inherent support for reuse, encapsulation and openess, distribution, heterogeneous execution (at both the s/w and h/w levels), and dynamic evolution by means of supporting unrestricted joining and leaving of components (human or otherwise).
- The coordination model offers a concrete modelling framework which is coupled with a real language in which we can effectively compose *executable specifications* of our coordination patterns.

However, in those few recent works where some coordination model and language is used to model activities in information systems, we have identified a number of inherent and potentially serious deficiencies, at least as far as some activities of information systems are concerned. We believe that the alternative approach presented in this paper addresses these deficiencies in a satisfactory way. The rest of the paper is organised as follows. Section 2 presents the state-of-the-art in using coordination models and languages for modelling activities in information systems and highlights the deficiencies we just mentioned. Section 3 describes an alternative coordination model and associated language and section 4 illustrates the capabilities of this model for modelling information systems and also how the deficiencies of the other models are overcome. The paper ends with some conclusions and further work.

2 COORDINATION MODELS AND LANGUAGES IN THE MODELLING OF INFORMATION SYSTEMS

Over the past few years a number of coordination models and languages have been developed such as Linear Objects (LO), TAO, Gamma and the Chemical Abstract Machine (see, for instance, the collection [2]). 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 coordination between them 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*.

Linda is indeed a simple, intuitive and appealing coordination model enjoying properties such as uncoupling of agents, concurrency and non-determinism and easiness of learning and use. In fact, it is not a concrete language *per se*, but a small set of four *primitives* (`in`, `out`, `read`, and `eval`) which implement the above mentioned functionality and can be added to any programming language.

It should therefore come as no surprise that the first coordination models and languages which were developed for modelling information systems were based on the Linda formalism by suitably modifying and/or extending its basic mechanism. One such model is Sonia ([6]) which features the notion of *Agora* — a shared space where agents participating in some activity post to or retrieve from tuples featuring types and possibly timeout constraints on their access. Another related model is Laura ([17]) where the shared space (referred to as *service-space*) is used by agents to post to or retrieve from *forms*, each form containing a description of a *service-offer*, a *service-request* with arguments or a *service-result* with results. Finally, in Ariadne ([10]) the shared workspace is used to hold tree-shaped data and access to them is performed by means of *record templates*.

Although Linda is indeed a successful coordination model, when it is evaluated from the point of view of acting as a framework for modelling human and other activities in information systems, it has some potentially serious deficiencies (at least for some cases of modelling information systems) which carry over to all the other related models that are based on it (like the ones briefly mentioned in the previous paragraph). These deficiencies are:

- It is data-driven. The state of some 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 consider the case of coordinating human and other activities; here, 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 and realistic when 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* (eg., 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 more important perhaps, the use of such a widely public medium as the Tuple Space and its variants, suffers inherently from a major *security* problem which comprises 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); (iii) even worse, they can be forged without anyone noticing it. The repercussions of these deficiencies when modelling of information systems is concerned are rather obvious and need not be discussed any further.
- The Tuple Space and its variants are inherently flat structures; hence, it is difficult to create *meta coordinators*.

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 ([9,15]). Nevertheless, to implement 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 the underlying basic model.

In the rest of this paper we attempt to contribute to the discussion as to how the notion of coordination languages can be applied to the modelling of information systems activities by presenting an alternative, to the ones mentioned in this section, approach. This approach is based on the use of coordination languages whose main characteristics are the following: (i) communication between agents is done by means of point-to-point *stream* connections; (ii) the agents comprising a coordination pattern are defined by means of being in one of a number of predefined *states*, where a state is a set of observable stream connections; (iii) evolution of a community of coordinators is *event-driven* (or control based) in the sense that the agents in question observe the presence of events and react accordingly. Section 3 describes such a coordination language and its underlying computational model whereas in section 4 we use this framework to model activities in information systems and in the process we illustrate the benefits it enjoys compared with the other models mentioned in this section. Although the proposed framework is developed with a particular control-driven coordination language in mind, we believe the underlying principles can apply to other similar models.

We should probably at this stage stress the fact that we do not claim to have developed a fully-fledged coordination framework for information systems that adheres to established principles and practices of CSCW, groupware, and related technologies. Such a framework should take into account the findings reported in the vast literature on the above mentioned areas. Instead, we point out that the majority of approaches encountered so far in using coordination languages as vehicles for developing such coordination frameworks are data-driven and we show the benefits of using instead a control-driven approach.

3 THE COORDINATION LANGUAGE MANIFOLD

MANIFOLD ([4,5]) 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]). MANIFOLD is by no means the only control-driven coordination language; another typical member of this family of coordination languages is ConCoord ([11]). For a more extensive description and comparison of these two main families of coordination models we refer the reader to [16].

In MANIFOLD there exist 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 channels, 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.

The following is a MANIFOLD program computing 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. Finally, note that every process has a default input and a default output port. Thus, whenever a construct of the form `prod -> cons` is encounetered (such as `sigma -> print` above), it should be interpreted as `prod.out -> cons.in` where `out` and `in` are the default output and input ports of `prod` and `cons` respectively.

4 A MODELLING FRAMEWORK BASED ON MANIFOLD

In this section we describe a framework for modelling activities in organisations based on MANIFOLD (and its underlying coordination model IWIM). The first part of the section describes the mataphors used in our framework while the second part describes in detail the modelling of a specific scenario (and in the process we take the opportunity to introduce some more features of MANIFOLD). It should be noted that it is our intention that our framework be used by many different people, not necessarily proficient in coordination (or any other form of) programming. Hence, our framework is essentially a three level one: (i) the top part is an easy to use visual interface which defines graphically the interrelationships and behaviour of the involved agents; (ii) the middle part is a verbal (semi-formal) description of the states defining each agent and how it reacts to receiving some event; (iii) the actual implementation of the scenario to be modelled in MANIFOLD.

For reasons of brevity we do not describe level (i); this is essentially the environment presented in [7] adapted (by means of extensions and simplifications) to fit our purposes. What the environment described there does is to translate the visual description to the MANIFOLD code comprising level (iii). Level (ii), in addition to providing a semi-formal verbal description of the coordination activities, can also be used by people not comfortable with programming at either of levels (i) or (iii) to specify some agents' behaviour (possibly by filling predefined forms) and then these specifications may be translated by other people to either level (i) or directly to level (iii). It is an on-going research issue to formalise this level properly so that its mapping to either of the other two levels be done in a, as much as it is possible, straightforward manner. Eg., the person defining an activity at this level should be instructed to include a complete description of all possible states some entity can be in, data-flow relationships and raising and reacting to events (here, the person can be assisted by assuming some predefined event behaviours such as detecting the presence of input data, timeouts, etc.).

4.1 The Metaphors

In our model, all entities participating in an activity (humans, devices, CSCW tools, shared resources such as active or passive documents, etc.) are *agents*. We distinguish two categories of agents: *worker* agents which perform computational work (whatever that may be according to the specific details of some particular scenario) and *manager* agents which are responsible for coordinating the activities of worker agents. Note that manager agents may themselves be seen as worker agents from other, higher up in the hierarchy, manager agents. However, the genuine worker agents (i.e., those performing some actual computational task) such as computer programs, CSCW tools, hardware devices, etc. form the bottom level of the layer and cannot be subdivided any further. We are not concerned with the internal details of these agents, only with their interaction with their environment.

Every agent, whether it is a manager or a worker agent, communicates with its environment by means of at least one *input port* and one *output port*. If the agent in question is a human being then a useful metaphor for these ports is in-trays and out-trays; if the agent is a device or software program then ports refer to traditional input/output streams (for the case of programs) or input/output connection slots (for the case of devices). In general, an agent may have more than one input and/or output port.

Furthermore, every agent observes a number of *events* and reacts to them accordingly. Depending on the situation, useful metaphors for events are telephone calls, commands and instructions broadcast over speakers, etc. However, it should be made clear that the purpose of an event is to make one or more agents aware of some situation that must be handled; not to transfer actual data. This is the purpose of streams introduced promptly.

Agents communicate actual data between themselves by means of connecting respective pairs of their input and output ports via *streams*. Again, depending on the particular scenario that is being modelled, such a stream could be, say, a human courier who transfers work to be done from the out-tray (output port) of some agent to the in-tray (input port) of some other agent. It is possible that more than one in- and one out-trays are involved in a data exchange, eg. three agents may send data to the in-tray of a fourth agent or that data from the out-tray of some agent are sent (in duplicate) to the in-tray of two agents.

Every agent is defined at every moment in time by means of being in some *state*. It is known beforehand which states an agent can be in during the lifespan of its activities. Again, useful metaphors are easy to find but depend on the actual scenario (eg., the states that a porter could be in are observing whether somebody wants to enter/leave a building, opening the door to let the other person in/out, close the door and go back to the state of observing). In order for some agent to change its state, it must observe the raising of a particular event. Reacting to an event (and therefore changing the current state) typically means establishing new stream connections between in/out ports and abandoning old connections.

The above general description of our model, whose functionality will become more clear in the presentation of a specific example that follows, has some clear advantages over the traditional Linda-like approaches we have seen so far:

- Every worker agent is only concerned with getting workload from its input port(s), performing the required work for which it is responsible, and putting the outcome to its out port(s). Its only other communication with its environment is by means of observing (if at all) any events. This is an ideal worker: it can be moved to any other environment and as long as it gets suitable input will produce the required output without any need to know who has sent it the work to be done or, indeed, who (if anyone at all!) will make use of its output.
- Every manager agent is only concerned with making sure that the output produced by some worker agents are sent to some other worker agents that require it. The manager identifies workers by means of their responsibilities and workflow interdependencies, not the actual work (data) they produce. Such a manager agent is also an ideal one: it can be moved to another environment where worker agents have responsibilities with similar interdependencies, and will perform equally well.
- All entities comprising an activity, whether they be humans or otherwise, are treated *homogeneously*. This renders the model very flexible; for instance, new agents can come and go dynamically, tools can be upgraded to updated versions of them or completely substituted with new ones, "active" documents and other types of interactive and dialogue mode applications are supported easily, etc.
- The model is inherently secured at all levels where someone would wish to introduce such security measures. The workers need know nothing about the environment that they are working in. Even the managers, are responsible for organising the workflow of information, but cannot see the actual data that is being transferred. Furthermore, by virtue of the IWIM model, streams are secured connections: data cannot be lost or forged, except in catastrophic circumstances. However, if desired, selective broadcasting of data is possible by means of sending it to the input ports of more than one agent.

4.2 A Specific Scenario

The scenario that we will model consists (mainly) of four agents, as it happens all of them being humans, collaborating in the development of some document. The first agent is the *author* who is responsible for writing up the document as well as performing significant changes to it. The second agent is the *editor* who checks the document's validity, performs any corrections and, if necessary, returns the document back to the author for further substantial changes. The third agent is the *manager* (still a worker process though, as far as our model is concerned) which either approves the document or sends it back to the editor. The fourth agent is a true coordinator agent responsible for managing the workflow between the other three agents (it could represent a department's supervisor). We also assume and sometimes refer to the presence of other agents (especially atomic agents performing purely computational work) whose detailed description is not given here. For each agent and for reasons of brevity we will describe levels (ii) and (iii) of our framework as presented in section 4.1. Level (ii) will be described in terms of the states an agent can be in and what event it must observe to make a transition to another state, as well as what events it itself may raise. If the agent has more than one input or output port, this will also be mentioned explicitly. Level (iii) is the actual MANIFOLD code produced.

Author [level (ii)]

State 0: Receive document in in-tray:
Initially, the agent is sitting waiting for some input (an initial document in this case) to be received in its in-tray (input port).

State 1: Produce or modify document:
Upon receiving a document, the agent performs the required work. This is done by forwarding the document to an atomic process `writer` which actually does the work (say, a word processor) and wait for the changes to be made.

State 1a: Put document in out-tray:
When the work to be done on the document has completed, `writer` sends the document back to the agent and the latter puts it in the out-tray.

State 2: Recur from the begining:
The process `writer` terminates and the author goes back to the initial state.

State 3: Request to be substituted:
To illustrate the dynamic joining and leaving of agents we enhance the basic functionality of this agent so that if it wants to leave, it will raise the event `time_to_go`. The coordinator process then, will substitute this agent with another one.

Author [level (iii)]

```
manifold Author (event i_had_enough)
{
 event prod_amend_doc, doc_ready,
       send_doc, time_to_go, flushed.

 // State S0
 begin:(guard(input,full,prod_amend_doc),
        terminated(void)).

 // State S1
 prod_amend_doc:{
    process writer is Word_Program(doc_ready).
    begin:(activate(writer),
           input->writer,
           terminated(void)).

 // State S1a
    doc_ready:(writer->output,
               post(send_doc)).
    }.

 // State S2
 send_doc:(post(begin)).

 // State S3
 time_to_go:{begin:(raise(i_had_enough),
 guard(output,a_disconnected,flushed),
       terminated(void)).
 flushed:halt.
 }
}
```

We will describe in some detail the code produced for this first agent since it introduces new features of MANIFOLD. Initially `Author` sits back observing whether its in-tray has any work to be done. This is achieved by means of the construct guard(input,full,prod_amend_doc) which sets a guard on the input port and if some data arrives and is ready to be read (by virtue fo the second flag full) the event prod_amend_doc will be *posted* to the vicinity of Author (no other agent will observe it). The construct terminated(void) suspends execution of an agent until it observes some event that causes it to go to a new state. Upon detecting the presence of prod_amend_doc, Author creates an atomic process writer (an instance of a word processor in this case), it redirects the document which now sits in its in-tray to the default input port of writer and suspends waiting for the signal doc_ready which will be raised by writer once the latter has finished doing the required job and terminated its execution. Upon detecting doc_ready, Author forwards the now updated document to its out-tray and recurs back to its initial state waiting for another document to appear in its in-tray. This life-cycle continues until Author has decided that it had had enough work for today (it is not shown here how the agent reaches this decision but only that as a consequence it posts the event time_to_go which causes its transition to a new state). It then *raises* the event i_had_enough (which is observable to the environment of Author) and waits until someone (the coordinator in this case) has disconnected it from the rest of the apparatus and any work in its out-tray has been delivered (this is achieved by setting on its output port another guard with the flag a_disconnected). When Author has indeed been isolated from the other agents, it is free to halt.

Editor [level (ii)]

State 0: Receive document in in-trays:
Initially, the agent is sitting waiting to receive a document from either the author (in which case the document will be received in the in-tray from_author) or from the manager (by means of the in-tray from_manager). It will then post to its vicinity the events check_doc or send_doc_back respectively.

State 1: Check document:
If the document was received from the author, the editor forwards it to, say, a spell checker program which will make any corrections necessary.

State 1a: Send document to the manager:
If the document is ok, it is sent to the manager by putting it in the out-tray to_manager. The spell checker program terminates execution and the editor recurs to the initial state waiting to receive the next document.

State 1b: Send document back to the author:
If it is decided that the document needs major revisions, it is sent back to the author by means of putting it in the out-tray to_author. The spell checker program terminates execution and the editor recurs to its initial state waiting to receive the next document.

State 2: Forward document back to the author:
If the document was received from the manager, then the implicit understanding is that it requires major revisions and it is simply forwarded back to the author (in practice, the editor agent could itself attempt to make any required amendments and send it back to the manager — this additional scenario is not covered here for reasons of brevity). The editor recurs to the initial state waiting to receive the next document.

Editor [level (iii)]

```
manifold Editor (port in from_author, from_man,
                 port out to_author, to_man)
{
 event check_doc, doc_ok, doc_not_ok,
 send_doc_back, send_doc_man.

 // State S0
 begin: (guard(from_author,full,check_doc),
        guard(from_man,full,send_doc_back),
        terminated(void)).

 // State S1
 check_doc: {process checker is
              Speller(doc_ok, doc_not_ok).
    begin: (activate(checker),
            from_author->checker,
            terminated(void)).

 // State S1a
    doc_ok: (checker->to_man,
             guard(checker.output,empty,begin),
             terminated(void)).

 // State S1b
    doc_not_ok: (checker->to_author,
              guard(checker.output,empty,begin),
                 terminated(void)).
    }

 // State S2
 send_doc_back: (from_man->to_author,
                post(begin)).
}
```

Manager [level (ii)]

State 0: Receive document in in-tray:
Initially, the agent is waiting for a document sent by the editor for approval. If the editor has not responded within 5 minutes, then an alarm is raised.

State 1: Verify and forward document:
Upon receiving a document, the manager verifies whether the document is indeed ok (the actual verification procedure is not shown here). If the document needs further improvement it is sent back to the editor by putting it in the out-tray to_editor. Otherwise, the document is sent to the head of the department (whose actual description is not shown here) by putting it in the out-tray to_dept_head.

State 2: Timeout functionality:
If the alarm is raised, then the manager raises an appropriate signal, which when detected by the coordinator process causes the latter to create and link into the existing infrastructure a new editor process.

Further functionality:
If the manager detects that according to its own criteria not enough work is produced (say, receiving sufficient number of documents to approve), it raises the signal more_work. The coordinator process then adds more authors (and/or editors).

Manager [level (iii)]

```
manifold Manager (event more_work,
                  port out to_dept_head, to_editor)
 {
```

```
event verify_doc, overdue, new_editor.
priority overdue < verify_doc.

process timeout is alarm(60,000*5,overdue).

 // State S0
begin:(activate(timeout),
       guard(input,full,verify_doc),
       terminated(void)).

 // State S1
verify_doc:(if (document is ok)
              then input->to_dept_head
              else input->to_editor,
              guard(input,empty,begin)).

 // State S2
overdue: raise(new_editor).
… raise(more_work).
}
```

Note the timeout functionality included in the manager process. Timeouts are very important in the modelling of open systems and organisational activities since there is no guarantee that processes either do exist for the whole period of a system's operations or do not malfunction at some period in time. Thus, in order to avoid deadlocks and be able to detect faults and abnormal termination of processes, timeouts on at least some vital operations must be planned and activated when the need arises. In fact, timeout functionality is so important that we believe all the processes involved in some computation should have by default some timeout strategy. Purely for reasons of brevity, we have chose in this paper to demonstrate how our model handles timeouts by concentrating on the case of editor. Note that the predefined manifold `alarm(port time, event signal)` will post `signal` to the vicinity of `alarm` once `time` milliseconds have passed. Furthermore, note that by means of the `priority` statement, we have given different priorities to the events `overdue` and `verify_doc`. If it happens that at precisely the end of the 5 minutes interval the editor manages to respond, then the manager will go into state 1 rather than 2; otherwise, according to the semantics of the language, the commitment to state 1 or 2 would be done nondeterministically.

<u>Coordinator [level (ii)]</u>

State 0: Set up initial configuration:
The coordinator creates three processes representing an author, an editor and a manager and sets up the intended stream connections between the in- and out-trays (ports) of these processes.

State 1: Substitute author:
If the coordinator detects the presence of the signal `sub_me` from the author, it creates a new author process and redirects the stream connections accordingly. The rest of the processes involved in the configuration would never know the difference.

State 2: Add extra author:
If the coordinator detects the presence of the signal `raise_work` from the manager, it creates a new author process and links it to the editor. The latter will never detect the presence of more than one author (except implicitly by detecting the raising of the number of documents it receives to check). Note that in this state we introduce limited broadcasting; a document to be sent back from the editor to the author, is actually duplicated and sent to both authors. We assume that one of them will then work on the document and the other will ignore it (the code required for this synchronisation is not shown here, purely for reasons of brevity).

State 3: Timeout functionality:
If the coordinator detects the presence of the signal `new_editor` from the manager, it creates a new editor process and links it to the existing infrastructure.

<u>Coordinator [level (iii)]</u>

```
// We also show the overall MANIFOLD code
required to set up the apparatus.
// The "coordinator" process is the special
   manifold 'Main'.

manifold Author (event) import.
manifold Editor (port in, port in,
                 port out, port out).
manifold Manager (event, port out, port out).
manifold Writer (event) atomic.
manifold Speller (event, event) atomic.
manifold HeadDept atomic.
manifold Document input.

manifold Main
{
 event sub_me, raise_work.
 auto process author is Author(sub_me).
 auto process editor
                <doc_from_author, doc_from_man
                 | doc_to_author, doc_to_man>
             is Editor.
 auto process manager
 <input | doc_to_dept_head, doc_back_to_editor>
             is Manager(raise_work).
 auto process head_dept is HeadDept.
 auto process init_doc is Document.

  // State S0
 begin:(init_doc->author,
       author->editor.doc_from_author,
       editor.doc_to_author->author,
       editor.doc_to_man->manager,
       manager.doc_to_dept_head->head_dept,
       manager.doc_back_to_editor
                      ->editor.doc_from_man,
       terminated(void)).

  // State S1
  sub_me.*some_author:{
    auto process new_author is Author(sub_me).
    hold (new_author).
    begin:(some_author->new_author,
    new_author->editor.doc_from_author,
    editor.doc_to_author->new_author,
    editor.doc_to_man->manager,
    manager.doc_to_dept_head->head_dept,
    manager.doc_back_to_editor
                         ->editor.doc_from_man,
    terminated(void)).
    }.

  // State S2
  more_work.manager:{
    auto process another_doc is Document.
    auto process another_author is
                         Author(sub_me).
```

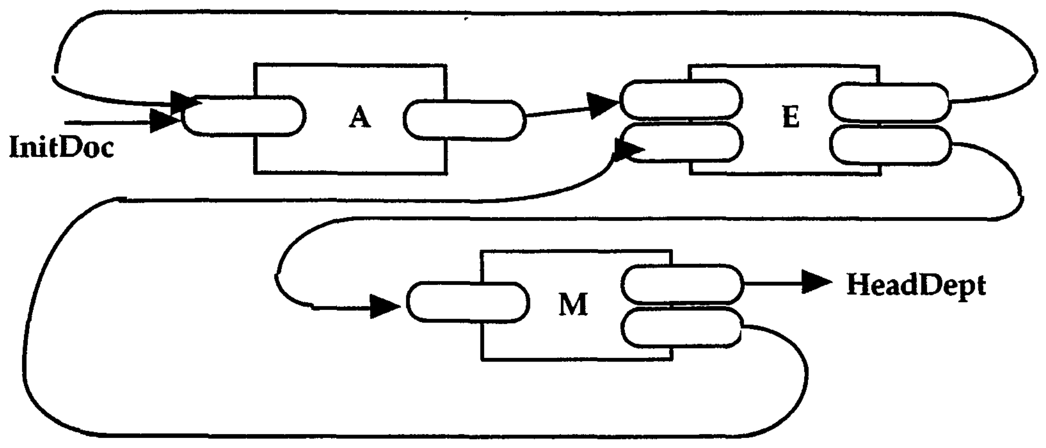

Fig. 1

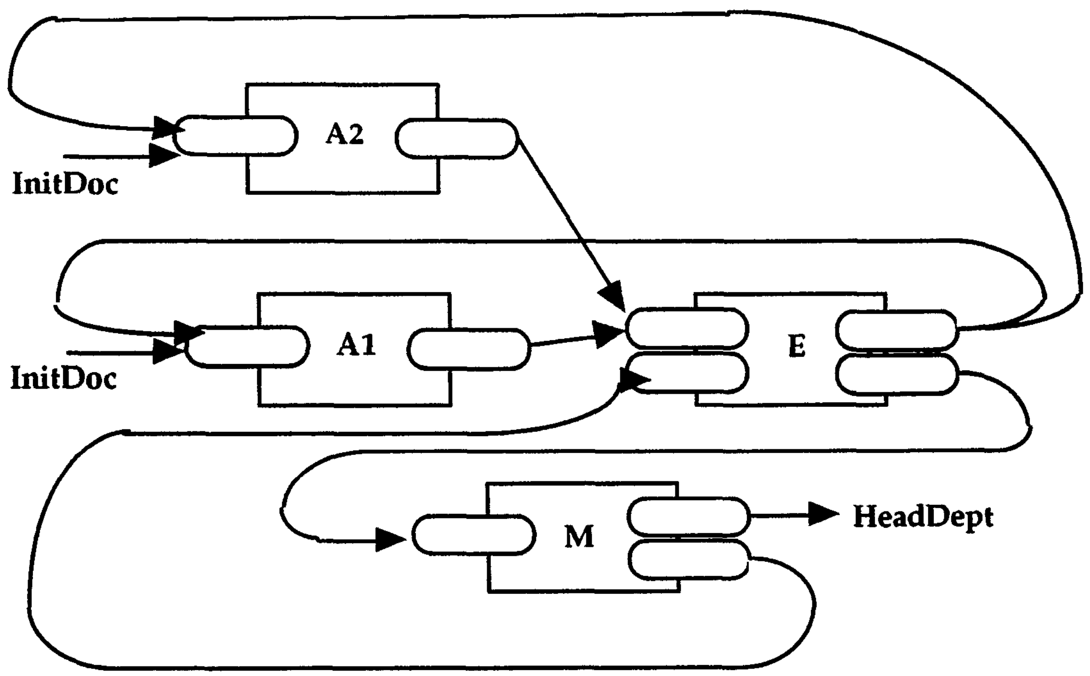

Fig. 2

```
   hold (another_author).
   begin:(another_doc->another_author,
          author->editor.doc_from_author,
          another_author
                   ->editor.doc_from_author,
          editor.doc_to_author ->
               (-> author, -> another_author),
          editor.doc_to_man->manager,
          manager.doc_to_dept_head->head_dept,
          manager.doc_back_to_editor
                   ->editor.doc_from_man,
          terminated(void)).
         }.

// State S3
new_editor.manager:{
   auto process another_editor is editor.
   hold (another_editor).
   begin:(author ->
                 another_editor.doc_from_author,
          manager.doc_back_to_editor ->
                 another_editor.doc_from_man,
          another_editor.doc_to_author->author,
          another_editor.doc_to_man->manager,
          terminated(void)). }.
}
```

Note the difference in the names of ports and events that are passed as actual parameters to various manifolds compared with those names used to represent the formal parameters. We end this section by visualising the framework in Visifold ([7]), MANIFOLD's visual interface. Figure 1 shows the setup at state 0 and figure 2 the setup at state 2.

5 CONCLUSIONS AND FURTHER WORK

We have argued for the need to use control-based, event-driven and state-defined coordination programming to model human and other activities in information systems. We have presented such a model, explained its benefits compared with other approaches that have been used so far, and illustrated its capabilities by means of a specific, even if rather simple, scenario.

In the short space of a conference paper it would be impossible to describe in detail all the characteristics of our model or compare them in detail with other related approaches. For instance, we have said nothing about examining types of values transmitted via streams (sometimes it may be desirable to know of the data's structure, if not content), etc. This and other issues can be adequately addressed by our model.

The reader may be concerned that we make no mention on the vast literature that exists for CSCW, groupware, and associated fields. We should again stress the fact that we do not claim to have presented a fully fledged coordination framework for information systems. We have only pointed out the benefits for using a control-driven coordination paradigm as opposed to a data-driven one. We are currently thoroughly evaluating the effectiveness of this basic framework to model various scenarios such as version management, electronic voting, teleconferencing, open distributed systems, etc. In addition, we will let users in private and public institutions use the model for their own activities, thus providing invaluable feedback in further fine-tuning it (especially with respect to levels (i) and (ii)). Eventually, we aim to developing a fully-fledged control-driven framework for coordinating people and supporting cooperative work by combining the characteristics of a coordination language such as MANIFOLD with the principles and ptactices pertaining in CSCW and related fields.

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

[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, C. L. Blom, F. J. Burger and C. T. H. Everaars, 'Reusable Coordinator Modules for Massively Concurrent Applications', *Europar'96*, Lyon, France, 27-29 Aug. 1996, LNCS 1123, Springer Verlag, pp. 664-677.

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

[6] M. Banville, 'Sonia: an Adaptation of Linda for Coordination of Activities in Organizations', *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 57-74.

[7] 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.

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

[9] 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.

[10] G. Florijn, T. Besamusca and D. Greefhorst, 'Ariadne and HOPLa: Flexible Coordination of Collaborative Processes', *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 197-214.

[11] A. A. Holzbacher, 'A Software Environment for Concurrent Coordinated Programming', *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 249-266.

[12] M. Klein, 'Challenges and Directions for Coordination Science', *Second International Conference on the Design of Cooperative Systems*, Juan-les-Pins, France, 12-14 June, 1996, pp. 705-722.

[13] T. W. Malone and K. Crowston, 'The Interdisciplinary Study of Coordination', *ACM Computing Surveys* **26**, 1994, pp. 87-119.

[14] M. Marchini and M. Melgarejo, 'Agora: Groupware Metaphors in OO Concurrent Programming', *Object-Based Models and Languages for Concurrent Systems*, Bologna, Italy, 5 July, 1994, LNCS 924, Springer Verlag.

[15] 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.

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

[17] R. Tolksdorf, 'Coordinating Services in Open Distributed Systems With LAURA', *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 386-402.