Coordinating electronic commerce activities in MANIFOLD

George A. Papadopoulos [a] and Farhad Arbab [b]

[a] *Department of Computer Science, University of Cyprus, 75 Kallipoleos Str, P.O. Box 20537, CY-1678 Nicosia, Cyprus*
E-mail: george@cs.ucy.ac.cy
[b] *Department of Software Engineering, CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*
E-mail: farhad@cwi.nl

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 *enterprise 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 openness. 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 individ-

uals, requiring groups of people, sometimes distributed over different organisations, countries, etc.

Recently, we have seen a proliferation of the so-called *coordination models* and their associated programming languages [2,6,16]. 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 [11]. 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- versus control-driven coordination models and languages

Over the past few years a number of coordination models and languages have been developed [2,6,16]. 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 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 some of its variants suggest and enforce is hardly appropriate and leads to unnecessary 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 tuple space 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, 12,16]. 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 possesses 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

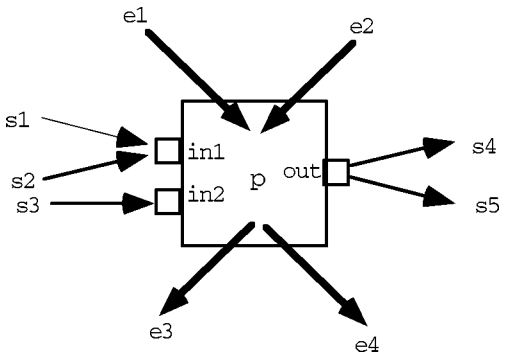

Figure 1.

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 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.

4. Electronic commerce frameworks 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 concentrate on three aspects: modelling e-commerce transactions, realizing security mechanisms, and illustrating the integration of different components. In the process we take the opportunity to introduce some additional features of MANIFOLD.

4.1. Modelling e-commerce transactions

We start with considering 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 product 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).

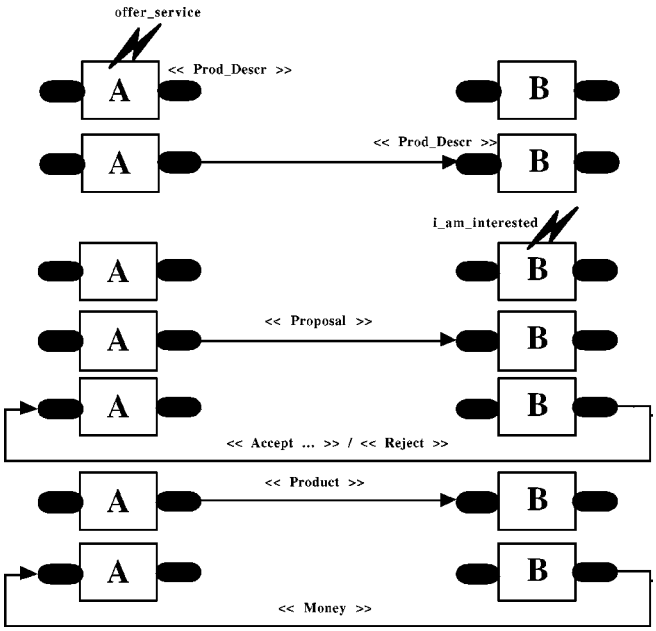

Figure 2.

- 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 above. 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.

Furthermore, 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 buyer 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).
                           }.
}

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 actually 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.

4.2. Realizing security mechanisms

In the previous case we have seen how MANIFOLD can provide the necessary security at the *implementation* level; in other words, we can be sure that the basic communication among interacting agents is secured and the transmitted data cannot be lost, intercepted or forged. However, we have not tackled the issue of security at a *logical* level; i.e., whether the involved agents are of the right type, have a valid identity, and behave in the intended way. So, although the transaction process in figure 2 will be secured as far as communication needs are concerned, the framework allows agents A and B to do anything they wish without restricting or checking their behaviour in any way. This happens because we have mixed together the communication and the behaviour protocols, with every agent being free to define completely its own behaviour irrespective of how it may affect other agents.

However, 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 permission from 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 [12,13] and in this section we illustrate how the framework described in [13] can be implemented naturally in our model. The overall setup is illustrated in figure 3 where for each e-commerce agent (implemented in MANIFOLD and other typical programming languages such as Java) there exists a controller or law enforcer implemented in MANIFOLD; the latter is used to intercept all messages sent from the agent to the rest of the world (i.e., other agents with which it cooperates) as well as messages sent to this agent from other agents. No agent is allowed to enter and get involved in some transaction without the presence of a MANIFOLD controller.

In particular, we show a number of regulator coordinators that enforce the following N-ticket law as defined in [13]. A client wishing to buy goods, first sends to a

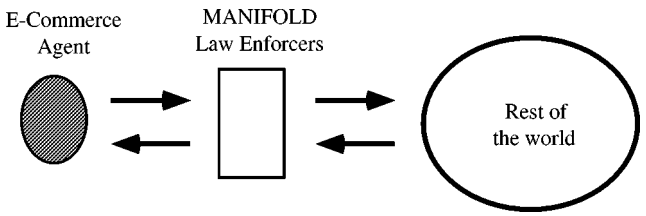

Figure 3.

ticket seller a request consisting of the value N of the ticket and an electronic certificate. If the certificate is valid, the ticket seller sends back to the client the N-ticket. The client then can use the ticket up to N times to buy goods. In addition to the elementary security that is needed at the implementation (or communication) level, we also want logical security in the sense that:

(i) only a ticket seller should be allowed to provide tickets,

(ii) tickets should not be duplicated by clients, and

(iii) clients should not use an N-ticket for more than N times. The MANIFOLD code for this scenario follows promptly.

```
event send_ticket.

auto process agora is Mall().
auto process seller is TicketSeller().
auto process client is Client().

manifold client_law(process client)
{
  event get_ticket, buy, req_ok.
  auto process ec is ElectronicCertificate().
  auto process value is variable.
  auto process certificate is variable.

  begin: terminated(self).
  get_ticket.client: client -> value;
                     certificate=ec;
                     <<&self,value,certificate>> -> seller;
                     terminated(self).
  send_ticket.seller:
    { auto process id_seller is variable.
      auto process id_client is variable.
      begin: getunit(input) ->
             <<id_seller,id_client,value>>;
             if id_seller==seller && id_client=client
             then value -> client;
                 { begin: terminated(self).
                          buy.client: if value>=1
                 then auto process request is variable;
                   auto process valid is Validate();
                   client -> (->valid, -> request);
                   req_ok.valid: request -> agora;
                   value=value-1
                 }.
    }.
}

manifold ticket_law (process seller)
{
  event ec_ok;
```

```
  auto process value is variable.
  auto process ec is variable.
  auto process id_client is variable.

  begin: getunit(input) -> <<id_client,value,ec>>;
         <<value,ec>> -> seller;
         terminated(self).
  ec_ok.seller: (raise(send_ticket),
                 <<&self,id_client,value>> -> id_client).
}
```

For every `client` wishing to get an N-ticket before starting buying goods and every `seller` providing tickets, there exist respectively a `client_law` and a `ticket_law` regulator, which enforce the law and intercept any communication between a `client` and a `seller`. When a `client` process wants to get a ticket, the corresponding `client_law` process sends the tuple `<<client_id,N-value, electr_certificate>>` to the `ticket_law` process. The latter informs the `seller` process it monitors that a ticket is being requested and `seller` checks the certificate. If the certificate is valid, `seller` informs `ticket_law` and the latter uses the `client_id` part of the tuple it received to connect to `client_law` and send the requested ticket. Upon receiving the ticket and checking by means of comparing ids that the ticket has been sent by the right seller and is addressed to the intended client, `client_law` informs `client` that the ticket has arrived. It then monitors any attempted transaction between `client` and `agora` (a group of service providers, themselves monitored by some other regulator coordinator not shown here), making sure that the client does not exceed the N-value limit.

Note that the above coordinators enforce security at both the logical and the implementation level. A `client` process cannot copy a received ticket to some other similar process since the ticket is being held by its regulator `client_law` process; nor can `client` exceed the ticket's value since when $N = 0$ any additional transaction between `client` and `agora` will not be allowed. Furthermore, a `ticket_law` process makes sure that tickets can only be issued by a valid `seller` process and, once a ticket is issued, it is forwarded to the `client` process that requested it; this is achieved again by making use of the ids of itself as well as the `client` process that requested the ticket. Finally, as we have already discussed previously, elementary security at the level of exchanging information is guaranteed by virtue of the IWIM model and MANIFOLD's implementation.

4.3. Integrating different components

In this section we show the applicability of control-based event-driven coordination models such as MANIFOLD for the development of generic interaction frameworks, often referred to as *shopping models* [10], where the interaction and communication part (in other words the program logic) is separated from low level details such as the security or payment mechanisms employed, etc. The top-level environment

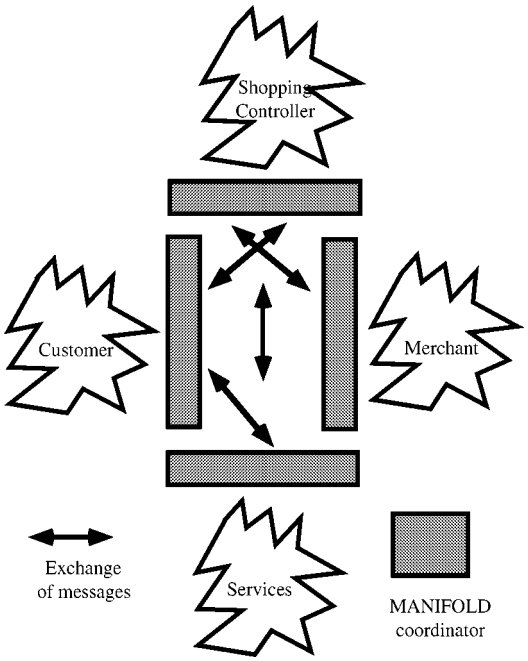

Figure 4.

follows the logic of [10] and consists of four main components: a *merchant handler*, a *customer handler*, a *shopping controller* and a *services controller*. The first two are used to intercept and handle the requests, messages and data interchanged between a customer and a merchant, the third one coordinates the interaction between the previous two entities while the last one controls the invocation of specific services such as payment methods. As illustrated in figure 4, customers and merchants interchange messages indirectly via a shopping controller, which monitors their interaction, and, if the need arises, plays the role of an objective referee. Furthermore, clients interact with service controllers to whom they delegate the, often of lower level nature, tasks of invoking specific services. All interactions are event-driven and the coordinators for these four basic entities (shown in shaded rectangular boxes) are implemented in MANIFOLD. The actual model we have in mind is quite elaborate and complicated but for the purposes of this paper we show the most important parts of the MANIFOLD code for the case of initiating a payment from a customer to a merchant assuming that the order has just been completed.

```
event order_complete, commence_payment, receive_order,
payment_complete.

manifold CustomerHandler()
{
  event proceed.
  process checkpayment(proceed) is CheckPayment atomic.
```

```
  begin: terminated(self).
  pay.*shopping_controller:
      (activate checkpayment, terminated(self)).
  proceed.checkpayment:  raise(commence_payment).
  ...
}

manifold MerchantHandler()
{
  event order_ok.
  auto process checkorder(order_ok) is CheckOrder atomic.

  begin: terminated(self).
  receive_order.*shopping_controller:
               (shopping_controller -> check_order, terminated(self)).
  order_ok.checkorder:  raise(order_complete).
  ...
}

manifold ShoppingController()
{
  begin:  terminated(self).
  ...
  order_complete.*merchant: raise(pay).
  payment_complete.*services_controller: ...
  ...
}

manifold ServicesController()
{
  event payment_done.
  process paymentservice(payment_done) is PaymentService atomic.

  begin: terminated(self).
  commence_payment.*customer_handler:
    (activate paymentservice, terminated(self)).
  payment_done.paymentservice: raise(payment_complete).
  ...
}
```

After being alerted (by means of observing the event `receive_order`) by the `Services` controller to the arrival of a new order, the `Merchant` handler checks if the order is complete. The lower level details of how this is done are immaterial to the basic program logic and are delegated to some atomic process (typically a C program accessing a database of information and making any necessary checks). Upon receiving by this atomic process a confirmation that the order is `ok`, the `Merchant` handler raises the event `order_complete`, indicating that the phase of a customer ordering some goods has been completed. The `Services` controller then commences the payment phase by raising the event `pay`. In response to observing this event, a `Customer` handler first makes sure that he agrees with the details of the payment procedure (this

lower level activity is performed by the atomic process `CheckPayment` possibly by contacting the `Merchant` handler) before raising the event `commence_payment`; the latter will be observed by the `Services` handler which will proceed to perform the payment. Again here the lower level details of the actual payment transactions are being delegated to some atomic process (`PaymentService`) which is parametric to the top level generic shopping model and can be substituted at will (e.g., initially DigiCash is used and later on a switching is made to First Virtual).

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 *openness* 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 an elaborate environment for electronic commerce based on the principles described in this paper.

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 [14–16], 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 tuple space. 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 publicizing the most necessary information.

References

[1] S. Ahuja, N. Carriero and D. Gelernter, Linda and Friends, IEEE Computer 19(8) (1986) 26–34.

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

[3] F. Arbab, The IWIM model for coordination of concurrent activities, in: *1st Internat. Conf. on Coordination Models, Languages and Applications (Coordination '96)*, Cesena, Italy (15–17 April 1996), Lecture Notes in Computer Sciences, Vol. 1061 (Springer, Berlin) 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) 23–70.

[5] P. Bouvry and F. Arbab, Visifold: A visual environment for a coordination language, in: *1st Internat. Conf. on Coordination Models, Languages and Applications (Coordination '96)*, Cesena, Italy (15–17 April 1996), Lecture Notes in Computer Sciences, Vol. 1061 (Springer, Berlin) pp. 403–406.

[6] N. Carriero and D. Gelernter, Coordination languages and their significance, Commun. ACM 35(2) (1992) 97–107.

[7] N. Carriero, D. Gelernter and S. Hupfer, Collaborative applications experience with the Bauhaus coordination language, in: *30th Hawaii Internat. Conf. on Systems Sciences (HICSS-30)*, Mauni, Hawaii (7–10 January 1997) (IEEE Press, New York) pp. 310–319.

[8] P. Ciancarini and D. Rossi, Coordinating distributed applets with Shade/Jada, in: *13th ACM Symp. on Applied Computing (SAC '98)*, Atlanta, GA (27 February–1 March 1998) (ACM Press, New York) 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, in: *Workshop on Programming the Web – a Search for APIs, 5th Internat. WWW Conf.*, Paris, France (May 1996).

[10] S.P. Ketchpel, H. Garcia-Molina and A. Paepcke, Shopping models: A flexible architecture for information commerce, in: *ACM DL '97*, Philadelphia, PA (24–26 July 1997) (ACM Press, New York).

[11] T.W. Malone and K. Crowston, The interdisciplinary study of coordination, ACM Comput. Surveys 26 (1994) 87–119.

[12] N.H. Minsky and J. Leichter, Law-governed Linda as a coordination model, in: *Object-Based Models and Languages for Concurrent Systems*, Bologna, Italy (5 July 1994), Lecture Notes in Computer Sciences, Vol. 924 (Springer, Berlin) pp. 125–145.

[13] N.H. Minsky and V. Ungureanu, A mechanism for establishing policies for electronic commerce, in: *18th Internat. Conf. on Distributed Computing Systems (ICDCS '98)*, Amsterdam, The Netherlands (26–29 May 1998) (IEEE Press, New York) pp. 322–331.

[14] G.A. Papadopoulos and F. Arbab, Control-based coordination of human and other activities in cooperative information systems, in: *2nd Internat. Conf. on Coordination Models and Languages*,

Berlin, Germany (1–3 September 1997), Lecture Notes in Computer Sciences, Vol. 1282 (Springer, Berlin) pp. 422–425.

[15] G.A. Papadopoulos and F. Arbab, Modelling activities in information systems using the coordination language MANIFOLD, in: *13th ACM Symp. on Applied Computing (SAC '98)*, Atlanta, GA (27 February–1 March 1998) (ACM Press, New York) pp. 185–193.

[16] G.A. Papadopoulos and F. Arbab, Coordination models and languages, in: *Advances in Computers 46*, ed. M.V. Zelkowitz (Academic Press, New York, 1998) pp. 329–400.