Control-Driven Coordination Programming in Shared Dataspace

George A. Papadopoulos

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

E-mail: george@turing.cs.ucy.ac.cy

Farhad Arbab

Department of Interactive Systems
CWI
Kruislaan 413, 1098 SJ Amsterdam
The Netherlands

E-mail: farhad@cwi.nl

Abstract. This paper argues for an alternative way of designing coordination models for parallel and distributed environments based on a complete symmetry between and decoupling of producers and consumers, as well as a clear distinction between the computational and the coordination/communication work performed by each process. The novel idea is to allow both producer and consumer processes to communicate with each other in a fashion that does not dictate any one of them to have specific knowledge about the rest of the processes involved in a coordinated activity. Furthermore, the model is inherently control-driven where communicating processes observe state changes and react to the presence of events and where the main communication mechanism is limited broadcasting (as opposed to either point-to-point or unrestricted broadcasting communication). Although a direct realisation of this model in terms of a concrete coordination language does already exist, we argue that the underlying principles can be applied to other similar models. We demonstrate our point by comparing our model with an established and widely used coordination framework, namely the Linda-type Shared Dataspace model, and we show how the functionality of the former can be embedded into the latter, thus yielding an alternative Linda-based coordination framework.

1 Introduction

The concept of coordinating a number of activities, possibly created independently from each other, such that they can run concurrently in a parallel and/or distributed fashion has recently received wide attention. A number of coordination models have been developed for many application areas such as high-performance computing and distributed systems ([7]). Most of these models address important issues such as modularity, reuse of existing software components, language interoperability, portability, etc. However, we believe that they also share some weak points: (i) Lack of complete separation between the computational and coordination/communication components of the processes involved; (ii) Lack of complete symmetry between the processes in the sense that traditionally, producers may have to know more

information about consumers than vice versa; (iii) Need for some process (producer or consumer) to know important information about the rest of the processes involved in a coordination activity such as their id, types and fashion (lazy, eager or otherwise) of communicating data, etc., which compromises the decoupling between them.

In this paper we argue for an alternative way of designing coordination models for parallel and distributed environments based on a complete symmetry between and decoupling of producers and consumers, as well as a clear distinction between the computational and the coordination/communication work performed by each process. The novel idea is to allow both producer and consumer processes to communicate with each other in a fashion that does not obligate any one of them to have specific knowledge about the rest of the processes involved in a coordinated activity. Furthermore, the above aims are achieved by employing a control-driven coordination mechanism (as opposed to the data-driven used by many other similar models) where the agent processes that are involved in some coordination activity observe state changes of both themselves and other relevant agents, raise events and react to observing events raised by other agents and communicate between each other by having output ports of themselves connected to input ports of other agents, thus employing a certain form of limited broadcasting. This new model is referred to as Ideal Worker Ideal Manager (IWIM,[3]). Although a direct realisation of IWIM in terms of a concrete coordination language does already exist ([5, 10]), we argue that the underlying principles can be applied to other similar models. We demonstrate our point by comparing our model with a state-of-the-art coordination framework, namely the Linda-type Shared Dataspace model, and we show how the functionality of the former can be embedded into the latter thus yielding an alternative Linda-based coordination framework.

An additional advantage of the proposed framework, stemming from the complete separation between computation and coordination/communication is that the two sets of activities can be isolated in respective sets of modules ([4]). The former set plays the role of what is already known as “computing farm” but the latter offers a new form of reusable entity, a “coordination farm”. Thus, different computational modules of similar operational behaviour can be plugged together with the same set of coordination modules enhancing the reusability of both sets and allowing the design of novel and interesting forms of “coordination programming”.

The rest of this paper is organised as follows. The next section is a brief introduction to IWIM; here we highlight those features of the model which we feel are unique to this particular philosophy of coordination. We then compare IWIM with the family of coordination models based on the metaphor of Shared Dataspace and in particular the most prominent of its members, namely Linda. We show how the IWIM features can be embedded in Linda, thus deriving an alternative Linda-based coordination framework which we term IWIM-Linda. The paper ends with some conclusions where we argue that such a comparison should be extended to include a variety of other families of coordination models.

2 The IWIM Model

Most of the message passing models of communication can be classified under the generic title of TSR (Targeted-Send/Receive) in the sense that there is some asymmetry in the sending and receiving of messages between processes; it is usually the case that the sender is generally aware of the receiver(s) of its message(s) whereas a receiver does not care about the origin of a received message. The following example, describing an abstract send-receive scenario, illustrates the idea:

```
process Prod:                          process Cons:

compute M1                             receive M1
send M1 to Cons                        let PR be M1's sender
compute M2                             receive M2
send M2 to Cons                        compute M using M1 and M2
do other things                        send M to PR
receive M
do other things with M
```

There are two points worth noting in the above scenario:

- The purely computational part of the processes `Prod` and `Cons` is mixed and interspersed with the communication part in each process. Thus, the final source code is a specification of both *what* each process *computes* and how the process *communicates* with its environment.
- Every `send` operation must specify a target for its message, whereas a `receive` operation can accept a message from any anonymous source. So, in the above example, `Prod` must know the identity of `Cons` although the latter one can receive messages from anyone.

Intermixing communication with computation makes the cooperation model of an application implicit in the communication primitives that are scattered throughout the (computational) source code. Also, the coupling between the cooperating processes is tighter than is really necessary, with the names of particular receiver processes hardwired into the rest of the code. Although parameterisation can be used to avoid explicit hardwiring of process names, this effectively camouflages the dependency on the environment under more computation. Thus, in order to change the cooperation infrastructure between a set of processes one must actually modify the source code of these processes.

Alternatively, the IWIM (Ideal Worker Ideal Manager) communication model ([3]) aims at completely separating the computational part of a process from its communication part, thus encouraging a weak coupling between worker processes in the coordination environment. IWIM is itself a generic title (like TSR) in the sense that it actually defines a family of communication models, rather than a specific one,

each of which may have different significant characteristics such as supporting synchronous or asynchronous communication, etc.

How are processes adhering to an IWIM model structured and how is their inter-communication and coordination perceived in such a model? One way to address this issue is to start from the fact that in IWIM 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. This suggests that a suitable (albeit by no means unique) combination of entities a coordination language based on IWIM should possess is the following:

- *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`.
- *Channels*. These are the means by which interconnections between the ports of processes are realised. A channel connects a (port of a) producer (process) to a (port of a) consumer (process). We write `p.o -> q.i` to denote a channel 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`.

The IWIM model supports *anonymous communication*: in general, a process does not, and need not, know the identity of the processes with which it exchanges information. This concept reduces the dependence of a process on its environment

and makes processes more reusable. Using IWIM, our example can now take the following form:

```
process Prod:                          process Cons:

compute M1                             receive M1 from in port I1
write M1 to out port O1                receive M2 from in port I2
compute M2                             compute M using M1 and M2
write M2 to out port O2                write M to out port O1
do other things
receive M from in port I1
do other things with M

process Coord:
do other things
create the channel Prod.O1 -> Cons.I1
create the channel Prod.O2 -> Cons.I2
create the channel Cons.O1 -> Prod.I1
carry on doing other things
```

Note that in the IWIM version of the example all the communication between `Prod` and `Cons` is established by a new coordinator process `Coord` which defines the required connections between the ports of the processes by means of channels. Note also that not only `Prod` and `Cons` need not know anything about each other, but also `Coord` need not know about the actual functionality of the processes it coordinates.

The IWIM model has already been implemented by means of a concrete coordination language, namely MANIFOLD ([3, 4, 5, 10]); however, as we have already said its principles apply to other coordination languages and the rest of this paper will illustrate this point.

3 Comparison with Shared Dataspace and IWIM-Linda

The notion of a conceptually shared dataspace via which concurrently executing agents exchange messages is a fundamental programming metaphor and a number of models make use of it in one way or another. Probably the most well known realisation of this notion is the Tuple Space as used by Linda ([2]). Linda's four tuple operations (`out`, `in`, `rd`, `eval`) constitute a minimal set via which coordination of a number of concurrently executing activities can be achieved. Linda is a true coordination model in that the executing processes can be written in any programming language and indeed there are a number of such successful marriages ([2]) such as C-Linda, Fortran-Linda and Prolog-Linda ([6]).

It is worth pointing out that Linda is not a concrete language but rather a set of "add on" primitives. This has many advantages (such as the fact that these primitives can fit into almost any computational model); however, the functionality offered, although simple to use and intuitive, is also minimal. One still has to program realistic coordination patterns on top of the "vanilla" ones offered by the model. We explore (and exploit) this aspect of comparison in the rest of this section essentially by deriving an IWIM-Linda model which we feel leads to an alternative Linda-based coordination framework.

Probably the most important notion of IWIM that must be expressed in Linda (or any other coordination model for that matter) is the sending and receiving of messages with complete lack of knowledge on the part of a receiver (respectively sender) as to who is the sender(s) (respectively receiver(s)) of a message. In other words we want to model the fundamental operation

```
prod.out -> cons.in
```

Of course, every such transaction can only be done via the Tuple Space. The following is a vanilla realisation of a communication channel.

```
channel(prod,out,cons,in)
{
 while (1)
 {
  in(prod,out,Index,Data);
  out(cons,in,Index,Data);
 }
}
```

The above process is effectively a Linda-like coordinator which makes the data tuples sent to the Tuple Space by some producer available to some consumer process. Each such data tuple, equivalent to an IWIM's unit of data as it flows through a stream, is of the form `<pid,chid,index,data>` where `pid` is the producer or consumer process id, `chid` is the id of the output or input channel, `index` is used to serialise access to the stream of data units and `data` is the actual data sent.

Note that, as it is proper in IWIM, the producer has no knowledge of who will consume its messages and vice versa. Also, the `channel` process can at any time redirect the flow of data without the awareness of the producer and/or consumer. Furthermore, using different values for the tuple field `cons`, a `channel` coordinator can duplicate data tuples from one producer process to a number of consumer ones. For instance the IWIM/ construct `prod.out -> (-> cons1.in, -> cons2.in)` can be realised in IWIM-Linda as follows:

```
channel(prod,out,cons1,in1,cons2,in2)
{
 while (1)
```

```
 {
  in(prod,out,Index,Data);
  out(cons1,in1,Index,Data);
  out(cons2,in2,Index,Data);
 }
}
```

This vanilla apparatus, however, does not really express the actual functionality of a proper IWIM stream connection. There is no provision for the producer and/or the consumer to break its connection with a channel in a graceful way. If `prod` or `cons` dies, `channel` will carry on "forwarding" the data resulting in either an indefinite suspension (if `prod` is dead) and/or data loss (if `cons` is dead). Moreover, there is no provision for *merging* channels; although the output from `prod` can be duplicated, it is not possible to redirect the output of a number of channels into a single input "port". Simply making `channel` (or some other similar process) to transform tuples produced by other producers into ones having the same value for the fields `cons` and `in` does not help because it is not possible to retain the partial ordering of received data within some stream (although the receiving of data between the streams can and will certainly be nondeterministic). This shortcoming is caused by the fact that the n-tuple `<prod,out,cons,in,…>` actually defines a single stream only if someone considers this structure as a whole. If we do not want the producers and the consumers to know about each other, then in order for them to still be able to distinguish between different streams, there is a need for an extra stream id field available to both sets of processes.

The above scheme of implementing channel operations in Linda has a resulting behaviour which is still different in some significant ways from the IWIM channels; these differences stem from the very nature of the Tuple Space. One difference is related to the issue of secured communication. An IWIM channel is secure in the sense that it represents an unbounded buffer where data units that flow through it are guaranteed to be delivered from source to sink. No loss of data can occur either because of some overflow in the channel or for any other reason. The same is not true when the Tuple Space is used as a communication medium since it is by nature a public forum. Safe delivery of data has to rely on the assumption that only the intended processes either send data (`out`) down some "channel" or retrieve data (`in`) from it. Otherwise we would have forging of some channel's output or loss of data units respectively. In order to achieve this security, which is of major importance in realising IWIM, one would have to create additional guard or filter processes which would check whether a process is actually allowed to perform some `in` or `out` operation and if not either postpone or abort it completely ([9]). Alternatively, the notion of multisets as in Bauhaus Linda ([8]) can be used to facilitate such secure private communication.

Another major difference is related to efficiency. An IWIM channel is effectively a point to point communication medium and transfer of data is physically confined to only those processors handling the involved processes. However, this knowledge of locality is lost when realising channels via the Tuple Space since this is only logically shared but physically distributed. This problem can be alleviated to some extent by providing the intended functionality at some higher level of abstraction and thus help a Linda compiler derive more optimised code ([1]). But to what extent this will be practical for large dynamically evolving systems remains to be seen.

The purpose of the discussion so far was to highlight some important issues that must be raised and dealt with in using the Tuple Space as a means to achieve coordination and communication between processes in an IWIM fashion. We now provide a more concrete realisation of IWIM-Linda. An IWIM-Linda computation consists of the following groups of entities:

- Ordinary *computation* processes using the Tuple Space by means of the usual Linda primitives. Each such process should have no knowledge about the rest of the processes involved in a computation. We recognise two different types of such processes: (i) "*IWIM-Linda compliant ones*" which adhere to the principles of IWIM as discussed in the previous section and behave as shown in the relevant figure. These processes enjoy the full functionality of the model since they are able to communicate via multiple ports, broadcast and receive a plethora of events, etc. (ii) Non-compliant (in the sense just described) processes. These processes can still function within our framework where communication is done via default input and output ports and events are raised by the underlying system software (compiler, operating system, etc.).
- For each such process, whether it is IWIM-Linda compliant or not, we introduce a *monitor* process which intercepts all communication between the process and the Tuple Space. The monitor process is effectively responsible for handling all aspects related to the main process' interface with the environment, namely delivering input/output data from/to respective ports, handle raised events, etc.
- Finally, there are *coordination* processes which set up the stream connections between computational processes by communicating with the respective monitoring processes.

The monitor and coordination processes can in fact be implemented in any language (and we admit that a natural choice can be the host language the computation processes are written in); for instance, the vanilla channel code above was written in C. Nevertheless, we have found out that it is more natural to use a symbolic language formalism, namely the concurrent logic language one ([11]). The formulation of the necessary functionality is easily and succintly expressed; in addition the concepts of guarded don't care selection and concurrency, inherent in a concurrent logic program, provide an almost ideal mechanism to express the non-deterministic features of our model. Finally, the derived monitor/coordination concurrent logic program can be used as a "skeleton" ([12]) for adding IWIM-based

coordination to other (non Linda-type) models. Due to space limitations we refrain from describing here a detailed and complete implementation of IWIM-Linda and we go directly to the implementation of a concrete example, namely the Fibonacci series program shown in the previous section, and we describe its implementation in an IWIM-Linda fashion, explaining in the process the features of our methodology.

```
sigma()
{
 while (1)
 {
  in("unit",x,?int1); in("unit",y,?int2);
  if (safe to add numbers)
     out("unit",out,int1+int2)
  else { out("event",overflow); out("unit",out,error); break; }
 }
}
```

`sigma` is written in the host language (C in this case) and it is IWIM-Linda compliant in the sense that it recognises the concept of receiving data from multiple (input in this case) ports, raises events (`overflow`) and repeats the procedure by enclosing everything in an infinite loop. Otherwise, it is an ordinary computation process unware of who sends it data, who (if anyone) is receiving its output, how many producers and/or consumers are connected by means of streams to its input and output ports, etc. We stress again the fact that this need not be the case; a non-compliant `sigma` which simply `in`s two data tuples and `out`s the result without recognising ports, supporting repetition or raising of events can also function in an IWIM fashion at the expense of creating a more elaborate monitor process than the one shown below: the monitor would have to filter out all references to ports from tuples, trap the raising of events from the underlying system software (compiler, operating system, etc.) and reactivate `sigma` each time a new pair of input data arrives at the default nominal input port. The monitor process for a compliant `sigma` follows promptly (only the most essential aspects of the functionality are shown for the sake of brevity and clarity).

The first group of rules serves requests received for stream connections to `sigma`'s ports that are sent by coordinators. For brevity we show the rule for the input port `x`; the same procedure is followed for the rest of `sigma`'s ports whether these are input or output.

```
sigma([in("estbl_str",Self,x,Str_Id)|Rest],X_Strs,Y_Strs,Out_Strs,Eve
nts,…)
   <- X_Strs=[(Str_Id,0)|_], out("ack",Self,Str_Id),
      sigma(Rest,X_Strs,Y_Strs,Out_Strs,Events,…).
```

`sigma`'s first argument is a stream providing a communication link to both the Tuple Space and the respective computation `sigma` process. When a *control tuple* requesting a stream connection and sent by some coordinator process is received, the monitor keeps a note of the stream_id for the respective port and initialises a counter to 0. More to the point, for each port `sigma` keeps in a respective number of arguments a list (set) of the form `[(str_id1,index1), (str_id2,index2), …]` where each member of the list (set) is a pair comprising the id of a stream connected to that port and an associated index to be used in retaining the partial ordering of units of data sent down the stream. There are additional arguments for other related information (for instance, which events can be observed and/or raised by the process, etc.). Note that `Self` refers always to the id of the dual process entity (monitor-computation). Note also that upon receiving a request for a stream connection, `sigma` `outs` an acknowledgment message to whichever coordination process has requested it.

The second group of rules intercepts the attempted communication to and from the Tuple Space and presents it to the processes involved in an IWIM fashion. Here we show the rules executed for a typical `in` operation sending data from the Tuple Space to the `x` port and a typical `out` operation sending data from the `out` port to the Tuple Space.

```
sigma([in("unit",Self,Port_Id,Stream_Id,Data,Index)|Rest],X_Strs,…)
   <- Port_Id==x, (Stream_Id,Index+1)@X_Strs |
      sigma!out("unit",x,Data),
      Index'=Index+1, New_X_Strs=X_Strs.(Stream_Id,Index'),
      sigma(Rest,New_X_Strs,…).

sigma([Self?out("unit",Port_Id,Data)|Rest],…,Out_Strs,…)
   <- Port_Id==out |
      send_out(Data,Self,Port_Id,Out_Strs,New_Out_Strs),
      sigma(Rest,…,New_Out_Strs,…).

send_out(Data,Self,Port,[(Str_id,Index)|Rest],New_Port)
   <- out("unit",Self,Port,Str_Id,Data,Index),
      New_Port=[(Str_id,Index+1)|New_Port_Rest],
      send_out(Data,Self,Port,Rest,New_Port_Rest).
send_out(_,_,_,[],New_Port) <- New_Port=[].
```

The first rule intercepts a unit sent by some coordinator process to the `x` port of `sigma`. The respective monitor process first establishes that the stream connection does indeed exist; note here that the expression `(Stream_Id,Index+1)@X_Strs` which uses the set operator '`@`' succeeds if the pair `(Stream_Id, Index+1)` is an element of the list `X_Strs`. Upon the successful evaluation of the guard, the rule commits to the body where an "ordinary" version of the tuple is sent to

the sigma computation process. Here note the operator '!' which forwards the tuple to sigma. The rule updates the index entry kept for the stream and recurses.

The next two rules are used to intercept outgoing units from the computation process to the Tuple Space via some output port. Upon receiving such a unit the monitor process replicates it and sends it down all streams connected to the port in question updating again, appropriately, the relevant indices. Note here the use of the symmetric (to '!') operator '?' which is used by a monitor process to accept tuples from its respective computation process.

The final group of rules handles the raising and receiving of events. In general, events are truly broadcast to the whole Tuple Space to be picked up by any process interested in them. Note that events are read by processes rather than being ined; this is due to the fact that more than one process may be interested in the raising of some event. The rules below are a sample of the actual set of events that can be raised during the lifespan of the computation either by some process or by the system itself.

```
sigma([Self?out("event",type)|Rest],…,EventsRaised,…)
   <- type@EventsRaised | out("event",type,Self), sigma(Rest,…).
sigma([rd("event",type,Process_Id)|Rest],…,EventsReceived,…)
   <- type@EventsReceived | Self!in("event",type), sigma(Rest,…).
```

The first rule belongs to the category of those intercepting events raised by the computation process. The respective monitor process first checks whether the computation process can indeed raise the event by establishing that there is a relevant entry in the list of events to be raised. It then forwards the event to the Tuple Space, adding the id of the process and, in a typical IWIM fashion, carries on without worrying about the outcome of the raising of the event. The second rule belongs to the category of those events raised by some other process. The monitor process first establishes that the raised event is observable by its dual computation process before forwarding it to the latter. Note that observing an event typically leads to the breaking up of some stream connections and/or termination of processes; eg. observing the occurrence of the event halt leads to the execution of the following rule:

```
sigma([rd("event",halt,Process_Id)|Rest],X_Strs,Y_Strs,Out_Strs,…)
   <- out("event",ack_halt,Self), sigma(Rest,_,_,_,…).
```

Upon observing the raising of halt, the monitor process sends back an acknowledgment message and clears all stream connections effectively suspending the execution of the corresponding computation process. No data units will flow into or out from that process until new stream connections have been established (the actual procedure followed is in fact more complicated involving the termination and restarting of the process but it is not shown here for brevity).

The next set of rules implement an IWIM "variable"; these special types of processes are in practice implemented at a lower level for the sake of efficiency. Note that there is no need to keep the value of the variable as a parameter to the coordinator

process since assignment can be realised by means of feeding the contents of the `out` port back to the `in` port. Also, we choose to support no indices since attaching multiple streams to the `in` port would be logically obscure. Due to the simplicity of the process, there is no need for an associated computation process.

```
variable([in("estbl_str",Self,in,Str_Id)|Rest],In_Strs,Out_Strs,Event
s)
   <- In_Strs=[(Str_Id,0)|_], out("ack",Self,Str_Id),
      variable(Rest,In_Strs,Out_Strs,Events).
variable([in("estbl_str",Self,out,Str_Id)|Rest],In_Strs,Out_Strs,Even
ts)
   <- Out_Strs=[(Str_Id,0)|_], out("ack",Self,Str_Id),
      variable(Rest,In_Strs,Out_Strs,Events).
variable([in("unit",Self,in,Stream_Id,Data,_)|Rest],…)
   <- (Stream_Id,_)@X_Strs |
      send_out(Data,Self,out,Out_Strs,New_Out_Strs),
      variable(Rest,In_Strs,New_Out_Strs,Events).
variable([in("event",halt,Process_Id)|Rest],In_Strs,Out_Strs,Events)
   <- out("event",ack_halt,Self), variable(Rest,_,_,_).
```

This main section of the paper ends with the description of the coordinator process responsible for setting up the whole apparatus. Again for brevity, we present the relevant rules in a sketchy fashion. We urge the reader at this point to notice the benefits of using a concurrent logic notation to express, naturally, the concurrency involved in the activities performed by the coordinator process.

```
main(TS) <- out("estbl_str",variable0,out,StrId1),
            out("estbl_str",sigma,x,StrId1),       /* v0->sigma.x */
        /* etc. for v1->sigma.y, v1->v0, sigma->v1, sigma->print */
            main1(TS,variable0,variable1,sigma,print,
                  StrId1,StrId2,StrId3,StrId4,StrId5).

main1([in("ack",variable0,StrId1),in("ack",variable0,StrId3),…],…)
   <- forward(TS,variable0,out,sigma,x,StrId1),
  /* etc. for the streams created by the rest of the connections */
      monitor_events(TS,…).

forward([in("unit",Producer,Out_Port,Str_Id,Data,Index)|Rest],
        Producer,Out_Port,Consumer,In_Port,Str_Id)
   <- out("unit",Consumer,In_Port,Str_Id,Data,Index),
      forward(Rest,Producer,Out_Port,Consumer,In_Port,Str_Id).

monitor_events([rd("event",overflow,sigma)|_],…) <-
out("event",halt,main).
```

`main` sends out the control tuples to establish the stream connections between the processes involved in the coordination activities. It then calls `main1` which waits for the relevant acknowledgment messages to be sent back to it by the processes in question, signifying that the requested stream connections have been established. It then spawns a number of `forward` processes which are responsible for redirecting the data units from output ports to input ones via the proper stream connections. In addition, a `monitor_events` process scans the Tuple Space for any raised events. For instance, once an `overflow` event is raised by `sigma` `monitor_events` sends out a `halt` control tuple to signal termination of the stream connections and thus the whole spectrum of computation and coordination activities.

We recall that the complete separation of computation and coordination/communication activities enhances the reusability of both groups. Our Fibonacci series example above, being a rather specialised one, is not the ideal candidate for illustrating this point. Even so, one can notice that `sigma` behaves as some sort of possibly specialised merger receiving from two input streams and producing a single output stream after performing some computational activities. Thus, a more general `sigma` could be the following:

```
sigma(err_sig)
{
 while (1)
 {
  in("unit",x,data1); in("unit",y,data2);
  compute results;
  if (all_ok)
     out("unit",out,results)
  else { out("event",err_sig); out("unit",out,error); break; }
 }
}
```

The IWIM coordination framework as it was realised before remains unchanged (we assume only that `monitor_events` has a rule that will handle `err_sig`). In fact, even the name `sigma` can be factored out by using some suitable predicate name building operator which is offered by any standard concurrent logic programming environment, thus supporting the notion of parameterised coordinators which in MANIFOLD are called *manners* ([4]).

Conclusions - Related and Further Work

We have introduced an alternative approach to the modelling of coordination activities based on the IWIM model. Although the model is already realised by means of the concrete language MANIFOLD, we have argued that its principles are more general and lead to the creation of a new formalism and an associated family of

IWIM-like coordination models and languages. We have demonstrated our point by incorporating IWIM's principles in the Linda framework, deriving the IWIM-Linda coordination paradigm.

Our work is also somewhat similar in nature to Law-Governed Linda ([9]) where, again, Linda is enhanced with extra functionality in order to support some desirable features such as secured communication, the lack of which had been noted earlier by other researchers, and to enforce other constraints. In Law-Governed Linda, laws regulate the interactions of individual processes with the shared Tuple Space (and therefore with each other), analogous to the manner in which social laws do, e.g. secure financial transactions in the market place. In effect, laws in Law-Governed Linda establish various forms of secure message passing as well as multiple Tuple Spaces. Every process has a controller that acts as the mediator between it and the Tuple Space to ensure its compliance with the laws of the system. The laws are expressed in a restricted version of Prolog much in the same way that our IWIM-Linda coordinators are written using the concurrent logic programming notation. There is a good deal of similarity in the conceptual level of complexity of controllers and laws in Law-Governed Linda as compared to their analogous coordinators in IWIM-Linda. The notion of events in Law-Governed Linda in particular, and their role in coordination is also similar to the event mechanism supported by IWIM-Linda. However, note here that events in the IWIM-based models are more general since an event can also be erased from the (private) event memory of some coordinator process; due to lack of space we have not dwelt into this aspect of event handling in this paper. Finally, the capability-based message passing mechanism of Law-Governed Linda resembles communication through ports in IWIM-Linda.

Some of the issues and associated problems just mentioned are addressed also by Bauhaus Linda, a generalisation of the vanilla model where the notions of tuple and tuple space are unified into the single notion of a *multiset* ([8]). This generalisation both simplifies Linda and, simultaneously, makes it more powerful and expressive. Bauhaus Linda makes no distinction between passive and active objects. The concept of multisets allows multiple tuple spaces, as well as protected, safe, private communication. It is possible to have a hierarchy of multiple tuple spaces in Bauhaus Linda; this means that, as in IWIM-Linda, we can form meta-level coordinators. However, as in Linda, Bauhaus Linda does not enforce a separation of computation and coordination/communication concerns. Pure coordination and computation modules can be constructed in Bauhaus Linda, due to the availability of multiple tuple spaces and private communication through multisets, although there are no linguistic features to enforce or even encourage such a style of programming. In contrast, in IWIM-Linda this should be the only way to orchestrate the cooperation and communication between the coordinated components.

We are currently pursuing an extensive study of other coordination models and languages with the aim of deriving IWIM-like versions of them that, within the philosophy advocated by each such model, also support IWIM's basic principles. At

the implementation level we have embarked on realising IWIM-Linda by building a complete set of "skeleton functions" ([12]) as described in the previous section using some concurrent logic language and interfacing this apparatus to a Linda environment.

Acknowledgements

Part of this work was done while the first author was visiting CWI as part of the ERCIM-HCM Fellowship Programme financed by the Commission of the European Community under contract no. ERBCHBGCT930350.

References

[1] S. Ahmed, N. Carriero and D. Gelernter, "A Program Building Tool for Parallel Applications", *DIMACS Workshop on Specifications of Parallel Algorithms*, Princeton University, May, 1994.

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

[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] A. Brogi and P. Ciancarini, "The Concurrent Language Shared-Prolog", AC*M Transactions on Programming Languages and Systems* **13(1)**, 1991, pp. 99-123.

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

[8] N. Carriero, D. Gelernter and L. Zuck, "Bauhaus Linda", *Object-Based Models and Languages for Concurrent Systems*, Bologna, Italy, 5 July, 1994, LNCS 924, Springer Verlag, pp. 66-76.

[9] 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.

[10] F. Seredynski, P. Bouvry and F. Arbab, "Parallel and Distributed Evolutionary Computation with MANIFOLD", *Fourth International Conference on Parallel Computing Technologies (PaCT-97)*, Yaroslavl, Russia, 8-12 Sept., 1997, LNCS, Springer Verlag (these proceedings).

[11] E. Y. Shapiro, "The Family of Concurrent Logic Programming Languages", *Computing Surveys* **21 (3)**, 1989, pp. 412-510.

[12] D. B. Skillicorn, "Towards a Higher Level of Abstraction in Parallel Programming", *Programming Models for Massively Parallel Computers (MPPM'95)*, Berlin, Germany, 9-12 Oct., 1995, IEEE Press, pp. 78-85.