International Journal of High Speed Computing, Vol. 9, No. 2 (1997) 127–160
© World Scientific Publishing Company

COORDINATION OF DISTRIBUTED AND PARALLEL ACTIVITIES IN THE IWIM MODEL

GEORGE A. PAPADOPOULOS
Department of Computer Science, University of Cyprus,
75 Kallipoleos Str, P. O. Box 537,
CY-1678 Nicosia, Cyprus
e-mail: george@turing.cs.ucy.ac.cy

FARHAD ARBAB
Department of Software Engineering,
Centre for Mathematics and Computer Science (CWI),
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
e-mail: farhad@cwi.nl

Received November 1996

ABSTRACT
We present an alternative way of designing new as well as using existing coordination models for parallel and distributed environments. This approach is based on a complete symmetry between and decoupling of producers and consumers, as well as a clear distinction between the computation and the coordination/ communication work performed by each process. The novel ideas are: (i) 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, and (ii) to introduce control or state driven changes (as opposed to the data-driven changes usually employed) to the current state of a computation. Although a direct realisation of this model in terms of a concrete coordination language does exist, we argue that the underlying principles can be applied to other similar models. We demonstrate our point by showing how the functionality of the proposed model can be realised in a general coordination framework, namely the Shared Dataspace one, using as driving force the Linda-based formalism. Our demonstration achieves the following objectives: (i) yields an alternative (control- rather than data-driven) Linda-based coordination framework, and (ii) does it in such a way that the proposed apparatus can be used for other Shared-Dataspace-like coordination formalisms with little modification.

Keywords: Coordination languages and models, distributed and parallel computing models, software engineering for distributed systems.

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 [11]. Most of these models address important issues such as modularity, reuse of existing (sequential) software components, language interoperability, portability, etc. However, we believe that they also share some weak points: (i) lack of complete separation between the computation 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 complete decoupling between them; finally, (iv) almost always the coordination model is based on some "vanilla type" functionality which, although simple and intuitive for use, does not address other important issues such as security, optimised message handling, etc.

More to the point, one can distinguish three major uses of coordination models, namely the separation of the following components:

- Control issues from the computation concerns in the design and development of programs, thus allowing the correctness concerns (i.e., the computation) and the efficiency issues (i.e., the coordination) to be dealt with separately.
- Uniform operations on primarily passive data that are common in a large number of parallel/distributed applications, away from the application code, into a small set of generic primitives with their own independent well-defined semantics; this is applicable more to data-oriented applications.
- Communication protocols necessary for the cooperation of the active entities that comprise a parallel/distributed application into independent coordination modules; this is applicable more to control-oriented applications.

The natural description of the activities and the history of the computation carried out in a *data-oriented* application tends to center around the data; the application is essentially concerned with what happens to the data. However, there is a large class of applications, called *process-oriented* or *control-oriented* where it is unnatural to view their activities and their history as centered around the data. Indeed, often, the very notion of the *data*, as such, simply does not exist in these applications.

Comparatively, less work has been reported on models and languages for coordination with specific focus on control-oriented applications. Most of such relevant work takes the message passing paradigm as its base and modifies it by introducing such additional notions as synchronisers, contracts, constraints, and events. A major drawback of these approaches to control-oriented coordination models and languages is that they cannot overcome the deficiencies that are inherent in their underlying message passing paradigm.

On the other hand, there already exist a number of highly popular and successful coordination models and languages which are data-oriented rather than control-oriented. We would like to be able to use these models and languages for control-oriented applications too, and do so consistently and in as less ad hoc a way as possible.

In this paper we argue for an alternative way of designing new coordination models for parallel and distributed environments based on a complete symmetry between and decoupling of producers and consumers. We place particular emphasis on using existing coordination models to accomplish this, and make a clear distinction between the computation and the coordination/communication work performed by each process. The novel ideas are: (i) 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, (ii) to introduce *control or state driven* changes (as opposed to the *data driven* changes ordinarily employed by exisiting coordination models) to the current state of a computation, (iii) to clearly separate issues related to coordination and/or communication from the purely computational ones, and (iv) to address within the coordination/communication component, other important issues not directly handled by most coordination models, such as security of sending/receiving data, optimisation of message handling, etc. These ideas form the basis of a new model, referred to as Ideal Worker Ideal Manager (IWIM, [4]). Although a direct realisation of IWIM in terms of a concrete coordination language does exist [5], we argue that the underlying principles can be applied to other similar models, thus yielding control-oriented variants of them. 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 (control- rather than data-driven) Linda-based coordination framework. The apparatus we developed to achieve this can be used by other Shared Dataspace based coordination models with minimal modification.

This last point is achieved to a major extent by noting an additional advantage of the proposed framework; namely, that the complete separation between computation and coordination/communication results in the creation of two sets of activities which can be isolated in respective sets of modules [6]. 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 computation 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 the control-oriented coordination model IWIM with the data-oriented family of coordination models based on the metaphor of Shared Dataspace and in particular the most prominent of its members, namely Linda. We also make a first attempt to express IWIM's functionality in Linda, the purpose being to highlight some of the issues that must be addressed. In Section 4 we present a concrete realisation of our proposed framework, thus deriving an alternative control-oriented Linda-based coordination framework which we term IWIM-Linda. The paper ends with some conclusions and references to related and further work where we argue that a variety of other families of coordination models could benefit from similar IWIM extensions of them.

2. The IWIM coordination 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 computation 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 (computation) 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 [4] aims at completely separating the computation 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`. A channel can support either synchronous or asynchronous communication; we often refer to channels realising asynchronous communication as *streams*.
- *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.

In general, there are five different ways to model a communication channel C in IWIM, depending on what happens when either of the two ends of the channel (referred to as its *source* and *sink*) breaks connection with the respective (producer or consumer) process and what happens to any units pending in transit within C.

- Both ends of C have type S (synchronous connections). In this case there can never be any pending units in transit within C and a channel is always associated with a complete producer-consumer pair.
- Both ends of C have type K (keep connections). In this case the channel is not disconnected from either end if it is disconnected from the other end.
- Both ends of C have type B (break connections). In this case once the channel is disconnected from one end it will automatically also get disconnected from the other end.
- The source of C has a B and its sink has a K type connection. If a BK channel is disconnected from its consumer then it is also automatically disconnected from its producer but not vice versa.
- The source of C has a K and its sink has a B type connection. If a KB channel is disconnected from its producer then it is also automatically disconnected from its consumer but not vice versa.

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

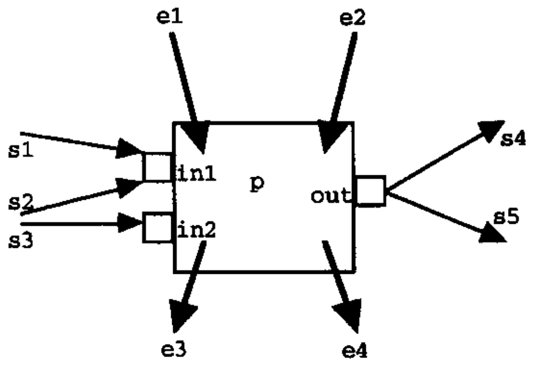

FIG. 1.

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 IWIM model has been realised by means of a concrete coordination language, namely MANIFOLD [5]. However, the purpose of this paper is not to introduce this language; this is done extensively elsewhere [4–6]. Instead, we argue that IWIM is in fact independent of its concrete realisation and provides a complete — and tested! (due to the very existence of MANIFOLD) — methodology for using any data-oriented coordination language for control-oriented applications. Nevertheless, for illustrative purposes, we present below the MANIFOLD version of a 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.

More information on MANIFOLD can be found in [4–6]; note that MANIFOLD has already been implemented on top of PVM and has been successfully ported to a number of platforms including Sun, Silicon Graphics, Linux, and IBM SP1 and SP2.

3. Comparison of IWIM with 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. Some of them are Gamma [7] based on the Chemical Abstract Machine and LO [3] based on Linear Logic as well as the models presented in [19,24]. Probably the most well known realisation of this notion is the Tuple Space as used primarily by Linda [2], and other similar models such as Swarm [21]. 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. The Tuple Space is a centrally managed dataspace which contains all pieces of information that processes want to communicate. A process in Linda is a black box and the Tuple Space exists outside of these black boxes which, effectively, do the real computing. The semantics of the tuples is independent of the underlying programming language used, thus rendering Linda a true coordination model where processes can be written in any programming language; indeed there are a number of such successful marriages [2,11] such as C-Linda, Fortran-Linda and Prolog-Linda [9].

The Linda formalism (and more generally the Shared Dataspace one) shares the same purpose with IWIM, namely addressing the deficiencies of the TSR model, and achieves that by providing symmetric, anonymous means of communication between parties. Nevertheless, there are a number of fundamental differences between Linda and IWIM. First of all, 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 computation 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. Thus, the simplicity of a Linda program is somewhat deceptive in that the program in question may not address issues such as security, optimisation techniques when point to point or selective broadcasting communication is required, reliability, etc.

For instance, if more than one process can obtain a tuple from the Tuple Space, a protocol is needed to avoid undesirable effects. This presents a problem reflecting the natural hierarchy of applications (such as divide-and-conquer or domain decomposition) that involve independent groups of processes, in their implementation. Ideally, each such group should communicate through its own private medium, to avoid accidental conflicts among the communications of unrelated parties. However, secure, private communication between two processes is not directly possible in Linda. Folding all these private communication media into one shared Tuple Space involves some non-trivial protocols and programming tricks leading to solutions that are not simultaneously elegant, simple and efficient. Furthermore, rather than being general, such solutions seem to be ad hoc extensions of the basic model derived from the need to address specific scenarios faced by their designers [8,16,18,19].

This comment should not be taken as a criticism of Linda (or, indeed, of its aforementioned extensions) but rather as a justification for the extra complexity that must be added when we want to build more sophisticated coordination patterns on top of the basic one offered by Linda. In fact, we explore (and exploit) this aspect of comparison in the rest of the paper, essentially by deriving an IWIM-Linda model which we feel leads to an alternative Linda-based coordination framework. In this respect, our work is along the lines of many other researchers who have extended the basic Linda formalism in one way or another [8,16,18,19]. But, contrary to the rather ad hoc extensions that some of these approaches suggest, ours is following a concrete, if abstract, model (namely IWIM — it is concrete because of the realisation of the model in terms of a specific coordination language). The very fact that the proposed framework has already been realised by means of a specific language suggests that it has been tried and proved useful as well as implementable; furthermore, it minimises the danger of containing "notional gaps" in the sense that it supports a "complete" coordination functionality (no important or necessary features are left out) in a consistent way (supported features do not have conflicting functionality).

Furthermore, the fact that these add on primitives must be fitted into a computation language leads to intermixing of computation and coordination/communication activities within individual processes. On the other hand, IWIM encourages a clean separation of computation and coordination concerns into different program modules. As we will see later on, this allows the reuse of coordination patterns by different computation modules.

3.1. Attempting to express IWIM's features in Linda. 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 a MANIFOLD'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/MANIFOLD 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 or MANIFOLD 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 non-deterministic). 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 [18]. Alternatively, the notion of multisets as in Bauhaus Linda [13] 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.

4. Realising IWIM in shared dataspace — the IWIM-Linda coordination framework. 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 in terms of the functionality offered by the Shared Dataspace. To make the discussion complete while producing concrete implementations of our ideas, we concentrate on Linda and present what we have already called IWIM-Linda. Nevertheless, we stress the point that the proposed framework is inherently independent of which Shared Dataspace coordination model is being used and thus can be applied to other such models [3,7,15,19,21,24] in order to provide IWIM extensions of them.

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 Fig. 1. 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's 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 computation processes by communicating with the respective monitoring processes.

Figure 2 below presents a possible scenario based on the framework just described. There are four computation processes (P1-P4) which communicate with their environment through an equal number of monitor processes (M1-M4), and also a number of coordination processes (C1-C3). C1 is used for the broadcasting of events while C2 and C3 are used for setting up and monitoring stream connections between the computation processes (via their respective monitor processes). So, for instance, P1 is the producer of data consumed by P3 by means of the stream connection S1 while P2 produces data through two streams, S2 and S3, the first one connected to an input port of P4 and the second to another input port of P3.

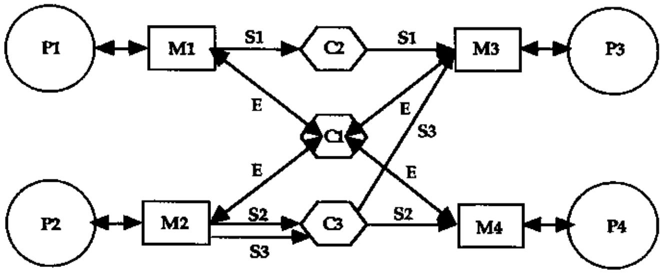

FIG. 2.

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 in Section 3.1 was written in C. Nevertheless, we have found that it is more natural to use a symbolic language formalism, namely the concurrent logic language one [22]. 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" [23] for adding IWIM-based coordination to other Linda-like coordination models. This can be achieved merely by modifying those pieces of code that use the Linda primitives and substituting them with equivalent code using the primitives of the other formalism; the rest of the apparatus however should remain mostly intact. We admit that to a certain extent the choice of the concurrent logic notation is somewhat biased (one of the authors has been pursuing research in this area for years) but, along with other researchers [18] we believe that when "laws", "rules" or "constraints" must be expressed, a symbolic notation is easier to use and modify. Finally, we note that the code written in a symbolic notation can be partially evaluated and transformed to more efficient versions [23].

We now describe in more detail the IWIM-Linda framework. Some rather lower level functionality is not covered here, primarily for reasons of clarity and brevity. The framework to be described is currently being implemented; the monitor and coordination processes are written in the concurrent logic language KL1 [14] and that system is interfaced to a C-Linda package running on a network of Unix machines (IBM RS/6000 running AIX 4.2). From the discussion so far it should be apparent that in implementing IWIM-Linda, three types of activities must be realised: (i) establishing and monitoring stream connections between computation processes via their respective monitor processes, (ii) actual sending and receiving of data, (iii) raising and reacting to events. The next three subsections cover each of the above topics while the following two present concrete examples.

4.1. Setting up and managing stream connections. The following clauses are used by a coordination process to request from a pair of monitor processes to set up a stream connection, thus realising an IWIM type communication channel between a producer and a consumer process along some output (respectively input) port.

```
channel(TS,Prod,Out_Port,Cons,In_Port,Type,...)
  <-  Str_Id='str_id_unique',                    /* some unique id generated */
      out("estbl_str",Prod,Out_Port,Str_Id),
      out("estbl_str",Cons,In_Port,Str_Id),
      receive_ack(TS,Prod,Out_Port,Cons,In_Port,Str_Id,Type,...).

receive_ack([in("ack_estbl_str",Prod,Str_Id),
             in("ack_estbl_str",Cons,Str_Id)|TS],
            Prod,Out_Port,Cons,In_Port,Str_Id,Type,...)
  <- forward(TS,Prod,Out_Port,Cons,In_Port,Str_Id,Type,...).
```

The above piece of code `outs` two tuples requesting the setting up of a stream connection between the processes `Prod` and `Cons`, along the ports `Out_Port` and `In_Port` respectively; the stream connection should be of type `Type` (we recall from Section 2 that in IWIM there are four types of stream connections, excluding the synchronous channel). Note that all clauses to be presented in the rest of the paper have two things in common regarding the header: (i) The first argument is always a communication stream with the environment of the clause; if the clause belongs to a coordinator process then the environment is just the Tuple Space but if it belongs to a monitor process then it is both the Tuple Space and the corresponding computation process. (ii) The symbol " ... " indicates that the clause may make use of more arguments which are not shown either because they are not relevant to the context of the discussion or because they are related to the handling of lower level issues. Thus, as far as this last point is concerned, the position of the arguments in the head of the clause should be taken for granted only in the context of the paragraph a clause is presented and discussed.

Upon receiving back the acknowledgments that the two monitor processes involved in setting up the requested stream connection have done so, a `forward` process (described in detail in the next subsection) is spawned to manage the transfer of data units through the stream. We end this subsection with the code executed by a monitor process which has been requested by a coordinator process in the fashion just described to assist in the setting up of some stream connection.

```
m_process([in("estbl_str",Self,Port_Id,Str_Id)|TS],Port_Strs,...)
  <-  New_Port_Strs=[(Str_Id,0)|Port_Strs],
      out("ack_estbl_str",Self,Str_Id),
      m_process(TS,New_Port_Strs,...).
```

Upon receiving a request for setting up a stream connection, the monitor process adds a new entry (`Str_Id,Index`) to the list of streams connected to the involved port held by the argument `Port_Strs`. We recall here that a port is either an input or an output one and consequently all attached streams either produce or receive data. Note also that `Index` (whose initial value is 0) is used to retain the partial ordering of messages received or sent by the monitor process along the same stream. Finally, note that `Self` is a special variable referring to the id of the dual process entity monitor-computation.

4.2. Sending and receiving data. Accomplishing the communication between monitor processes on behalf of their respective computation processes through some stream connection involves two sets of activities. The first is performed by some monitor process and involves the interception of messages intended to be sent by its corresponding computation process to the Tuple Space or received from it. These tuples must be modified in such a way that are presentable to the intended recipients; more details follow promptly as we examine the relevant code. In the code that follows we assume that the processes involved use only the default ports that always exist for communication, namely `in` for receiving data and `out` for sending data; it is straightforward to add similar rules for other ports (see code in Sections 4.4 and 4.5).

```
m_process([in("unit",Self,Port_Id,Str_Id,Data,Index)|TS],In_Strs,...)
  <-  Port_Id==in,
      (Str_Id,Index+1)@In_Strs
      |
      process!out("unit",in,Data),
      New_In_Strs=In_Strs.(Str_Id,Index+1),
      m_process(TS,New_In_Strs,...).

m_process([Self?out("unit",Port_Id,Data)|Rest],Out_Strs,...)
  <-  Port_Id==out
```

```
      |
      send_out(Data,Self,Port_Id,Out_Strs,New_Out_Strs,...),
      m_process(Rest,New_Out_Strs,...).

send_out(Data,Prod,Port_Id,[(Str_id,Index)|Rest],New_Port_Strs,...)
  <-  out("unit",Prod,Port_Id,Str_Id,Data,Index),
       New_Port_Strs=[(Str_id,Index+1)|New_Port_Strs_Rest],
       send_out(Data,Prod,Port_Id,Rest,New_Port_Strs_Rest,...).
send_out(_,_,_,[],New_Port_Strs,...) <- New_Port_Strs=[].
```

The first rule intercepts a unit sent by some coordinator process to the `in` port of `process`. The respective monitor process (`m_process`) first establishes that the stream connection does indeed exist; note here that the expression `(Stream_Id,Index+1)@X_Strs` which uses a special set operator '`@`' succeeds if the pair `(Stream_Id,Index+1)` is an element of the list `X_Strs` (in the process it may instantiate variables in the LHS of the operator if they are non-ground). Upon the successful evaluation of the guard, the rule commits to the body where an "ordinary" version of the tuple is sent to `process`. Here note the operator '`!`' which forwards the tuple to `process`. The rule updates the index entry kept for the stream and recurses. Also, note the update operator '`.`'; `New=Old.New_El` replaces some element `El` of a list `Old` with `New_El`. `El` is found by unifying the first argument (playing the role of an index) of the structure `El` with the corresponding argument in each of the arguments in `Old`. All these special operators are simply syntactic sugaring of calling rather straightforward-to-implement utility procedures.

The next two rules are used to intercept outgoing units from the computation process to the Tuple Space via some output port (in this case the default one `out`). Upon receiving such a unit the monitor process `m_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 which must be forwarded to the Tuple Space.

The above set of activities is responsible for sending and receiving unit tuples in a manner as dictated by IWIM, namely that neither receivers nor senders of data units are aware of each other. However, somewhere along the line the tuples must be modified and enhanced with the necessary information needed so that they can be delivered to the intended destination. This is accomplished by the second set of activities which is performed by a sufficient number of coordinator processes. The relevant code follows promptly.

```
forward([in("unit",Prod,Out_Port,Str_Id,Data,New_Index)|TS],
        Prod,Out_Port,Cons,In_Port,Str_Id,Base,Index,Type,...)
  <-  out("unit",Cons,In_Port,Str_Id,Data,Index+Base),
      forward(TS,Prod,Out_Port,Cons,In_Port,Str_Id,Base,New_Index,Type,...).
```

This rule intercepts a tuple sent by a monitor process on behalf of some computation process and modifies it so that the id of the sender is removed and instead the id of the receiver is added. `Base` and `Index` are used to retain the partial ordering between units sent down the same stream. In particular, `Index` retains the partial ordering during the lifespan of a certain stream connection. If the connection breaks but the stream is reconnectable then upon reestablishing the flow of data units, `Base` (which is initially 0 and subsequently has as value the last one `Index` had just before the connection broke) is added to the new index. Thus, we retain the partial ordering of messages flowing down a certain stream connection independent of how many times the stream has been broken and reconnected. We recall here that more than one stream can be connected to either an output or an input port. In the first case a data unit to be sent is replicated to all streams connected to the port; in the second case data units arriving at an input port connected to many incoming streams are received by the consumer process in a non-deterministic fashion.

We also recall that there are four types of stream connections, depending on whether and how a stream connection may be broken and reconnected. `forward` handles these activities as well but we discuss them in the following subsection since they are related to the handling of events.

4.3. Raising, detecting and reacting to events. IWIM is a control oriented coordination model. After establishing stream connections, data units flow through them to the intended destinations. This set up is retained until some event is raised by some process in which case, typically, a number of activities take place such as the breaking off of existing stream connections, the establishing of new ones, termination of processes or spawning of new ones. As in the case of sending and receiving data, the activities to be performed can be classified into two sets, the first performed by the monitor processes and the second by the coordination processes.

The monitor processes are again responsible for intercepting events intended to be broadcast or observed by their corresponding computation process.

```
m_process([rd("event",Type,Source,Self,Index)|TS],...,Events_Received,...)
  <-  (Source,Index)@Events_Received
      |
      Self!in("event",Type),
      New_Events_Received=Events_Received.(Source,Index+1),
      m_process(TS,...,New_Events_Received,...).

m_process([Self?out("event",Type)|Rest],,Events_Raised,...)
  <-  (Type,Index)@Events_Raised
      |
out("event",Type,Self,Index+1),
New_Events_Raised=Events_Raised.(Type,Index+1),
m_process(Rest,New_Events_Raised,...).
```

Before a monitor process forwards an event tuple to the corresponding computation process (if it has been received from the Tuple Space) or to the Tuple Space (if it has been received from the computation process), it first checks to see whether the following conditions are satisfied: (i) The event can indeed be observed or raised — the relevant information is kept in arguments; thus, `Events_Received` is a list of all sources from which raising of events can be observed by this process and `Events_Raised` is a list of all types of events that this process can raise. (ii) The partial ordering of observing events is retained. This last point is important because if some process raises two events, another process which is tuned to observe the raising of events from the previous process must do so in the order the events were raised. Here again, an index is used for that purpose, updated as the need arises.

Note that, unlike what is happening with data tuples, event tuples are `rd` rather than `in`ed; this is because a raised event is actually broadcast to a number of processes tuned to observe its occurrence rather than just one such process.

Some simple but useful variations of the first clause above (the one responsible for detecting the raising of events) are possible. We recall that in IWIM and MANIFOLD the expression `e.p` refers to an event `e` raised by some source `p`. In addition to detecting the raising of an event from a specific source, it is also possible to use "wildcards" and refer to `e.*` (i.e., has event `e` been raised from any source) or, indeed, `*.p` (i.e., has source `p` raised any event). This functionality can be easily expressed by modifying slightly the head of the relevant clause:

```
m_process([rd("event",Type,_,Self,Index)|TS],...)
  <-  ...

m_process([rd("event",_,Source,Self,Index)|TS],...)
  <-  ...
```

The second set of activities performed by the coordination processes involves the distribution of event tuples from the processes raising them to those that must observe them. This is implemented by means of the following clauses.

```
detect_event([in("event",Type,Source,Index)|TS],Observ_Procs,...)
  <-  distribute_event(e(Type,Source,Index),Observ_Procs,...),
      detect_event(TS,Observ_Procs,...).

distribute_event(_,[],...).
distribute_event(E,[(Type,Proc)|Rest_Observ_Procs],...)
```

```
  <-  E=e(Type,Source,Index)
      |
      out("event",Type,Source,Proc,Index),
      distribute_event(E,Rest_Observ_Procs,...).
distribute_event(E,[_|Rest_Observ_Procs],...)
  <-  distribute_event(E,Rest_Observ_Procs,...).
```

In the above code note that `Observ_Procs` holds the information related to which process can observe what event. Thus, an event is forwarded to some process only if its relevant entry can be found. Depending on how IWIM-Linda is implemented, this list could be updated dynamically or formed statically by means of suitable syntax annotations. Here for simplicity we adopt the second approach; however, our on going implementation uses the first one because it is more expressive.

We can now show how `forward`, introduced in the previous subsection, handles the breaking off and possible reconnection of streams that are already set up. If either the producer or the consumer process wants to break a stream connection it raises via its corresponding monitor process a relevant event which is detected by the coordinator process responsible for that stream connection. What happens afterwards is indicated by the following code.

```
forward([in("event",[break,Str_Id],Prod,_)|TS],
        Prod,Out_Port,Cons,In_Port,Str_Id,Base,Index,[T,k],...)
  <-  wait_new_connection1(TS,Cons,In_Port,Str_Id,Index,[T,k],...).

forward([in("event",[break,Str_Id],Prod,_)|TS],
        Prod,Out_Port,Cons,In_Port,Str_Id,Base,Index,[T,b],...)
  <-  out("event",[disconnect,Str_Id],Cons,0).

forward([in("event",[break,Str_Id],Cons,_)|TS],
        Prod,Out_Port,Cons,In_Port,Str_Id,Base,Index,[k,T],...)
  <-  wait_new_connection2(TS,Prod,Out_Port,Str_Id,Index,[k,T],...).

forward([in("event",[break,Str_Id],Cons,_)|TS],
        Prod,Out_Port,Cons,In_Port,Str_Id,Base,Index,[b,T],...)
  <-  out("event",[disconnect,Str_Id],Prod,0).

wait_new_connection1([in("event",[connect,Out_Port,Str_Id],Prod,0)|TS],
                     Cons,In_Port,Str_Id,Index,Type,...)
  <-  forward(TS,Prod,Out_Port,Cons,In_Port,Str_Id,Index,0,Type,...).

wait_new_connection2([in("event",[connect,In_Port,Str_Id],Cons,0)|TS],
                     Prod,Out_Port,Str_Id,Index,Type,...)
  <-  forward(TS,Prod,Out_Port,Cons,In_Port,Str_Id,Index,0,Type,...).
```

Each clause for `forward` handles one of the four types of stream connection, i.e., B $\rightarrow$ K, B $\rightarrow$ B, K $\rightarrow$ B or K $\rightarrow$ K. The `Type` argument of `forward` is in fact a binary list `[T1,T2]` indicating the type of connection from both the sender's and the receiver's point of view. Upon receiving an event to break connection `forward` decides as to whether simply wait until the stream gets reconnected again (if the other side of the stream is a keep connection) or totally dismantle the channel connection (if the other side of the stream is a break connection) in which case it sends an appropriate control tuple to the process handling the other end of the stream. For brevity we do not show what happens when a process receives a `disconnect` event although we get a glimpse of it in the next section describing the Fibonacci program.

4.4. A concrete example. We now use the above described apparatus to produce an IWIM-Linda version of the Fibonacci program of Section 2. This is a concrete, albeit not particularly realistic, example whose purpose is to put everything we covered in the previous sections into perspective. The next section discusses the implementation of a more realistic test case.

The only computation process is the one performing the addition. For simplicity we assume that it is IWIM-Linda compliant. Its code can be written as follows.

```
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 (some in-line optimisation of code has been performed for the sake of brevity and clarity). Note that the rest of the rules are written in KLIC and effectively implement the IWIM functionality.

```
m_sigma([in("estbl_str",Self,x,Str_Id)|TS],
        New_X_Strs,Y_Strs,Out_Strs,Events,)
  <-  New_X_Strs=[(Str_Id,0)|X_Strs],
      out("ack_estbl_str",Self,Str_Id),
      m_sigma(TS,New_X_Strs,Y_Strs,Out_Strs,Events,...).

m_sigma([in("estbl_str",Self,y,Str_Id)|TS],
        X_Strs,New_Y_Strs,Out_Strs,Events,...)
  <-  New_Y_Strs=[(Str_Id,0)|Y_Strs],
      out("ack_estbl_str",Self,Str_Id),
      m_sigma(TS,X_Strs,New_Y_Strs,Out_Strs,Events,...).

m_sigma([in("estbl_str",Self,out,Str_Id)|TS],
        X_Strs,Y_Strs,New_Out_Strs,Events,...)
  <-  New_Out_Strs=[(Str_Id,0)|Out_Strs],
      out("ack_estbl_str",Self,Str_Id),
      m_sigma(TS,X_Strs,Y_Strs,New_Out_Strs,Events,...).

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

m_sigma([in("unit",Self,Port_Id,Stream_Id,Data,Index)|TS],...,Y_Strs,...)
  <-  Port_Id==y,
      (Stream_Id,Index+1)@Y_Strs
      |
      sigma!out("unit",y,Data),
      Index'=Index+1,
      New_Y_Strs=[(Stream_Id,Index')|Y_Strs],
      m_sigma(TS,...,New_Y_Strs,...).

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

m_sigma([Self?out("event",Type)|Rest],...,EventsRaised,...)
  <-  (Type,Index)@EventsRaised
      |
      out("event",Type,Self,Index+1),
      New_Events_Raised=Events_Raised.(Type,Index+1),
      m_sigma(Rest,...,New_Events_Raised,...).

/* token clause to illustrate the handling of events;
   in practice there are many clauses here */
m_sigma([rd("event",halt,Process_Id,_)|TS],X_Strs,Y_Strs,Out_Strs,...)
  <-  Self!in("event",halt).
```

The next set of rules implement an IWIM "variable" needed in the example; 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)|TS],In_Strs,Out_Strs,Events)
  <-  New_In_Strs=[(Str_Id,0)|In_Strs],
      out("ack_estbl_str",Self,Str_Id),
      variable(TS,New_In_Strs,Out_Strs,Events).
variable([in("estbl_str",Self,out,Str_Id)|TS],In_Strs,Out_Strs,Events)
  <-  New_Out_Strs=[(Str_Id,0)|Out_Strs],
      out("ack_estbl_str",Self,Str_Id),
      variable(TS,In_Strs,New_Out_Strs,Events).

variable([in("unit",Self,in,Stream_Id,Data,_)|TS],In_Strs,Out_Strs,Events)
  <-  (Stream_Id,_)@In_Strs
      |
      send_out(Data,Self,out,Out_Strs,New_Out_Strs),
      variable(TS,In_Strs,New_Out_Strs,Events).

variable([in("event",halt,Process_Id,_)|TS],In_Strs,Out_Strs,Events)
  <-  out("event",ack_halt,Self,_).
```

We end this section with the description of the coordinator process responsible for setting up the whole apparatus. 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 */
            out("estbl_str",variable1,out,StrId2),
            out("estbl_str",sigma,y,StrId2),              /* v1->sigma.y */
            out("estbl_str",variable1,out,StrId3),
            out("estbl_str",variable0,in,StrId3),         /* v1->v0 */
            out("estbl_str",sigma,out,StrId4),
            out("estbl_str",variable1,in,StrId4),         /* sigma->v1 */
            out("estbl_str",sigma,out,StrId5),
            out("estbl_str",print,in,StrId5),             /* sigma->print */
            main1(TS,variable0,variable1,sigma,print,
                  StrId1,StrId2,StrId3,StrId4,StrId5).

main1([in("ack_estbl_str",variable0,StrId1),
       in("ack_estbl_str",variable0,StrId3),...|TS],...)  /* rest of acks */
   <-  forward(TS,variable0,out,sigma,x,StrId1,0,0,[b,k],...),
       forward(TS,variable1,out,sigma,y,StrId2,0,0,[b,k],...),
       forward(TS,variable1,out,variable0,in,StrId3,0,0,[b,k],...),
       forward(TS,sigma,out,variable,in,StrId4,0,0,[b,k],...),
       forward(TS,sigma,out,print,in,StrId5,0,0,[b,k],...),
       detect_event(TS,[(main,halt),(sigma,overflow),...],...).
```

`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 and a `detect_events` process having the functionality described in the previous subsections. For instance, once an `overflow` event is raised by `sigma` `detect_events` sends out a `halt` control tuple to signal termination of the stream connections and thus the whole spectrum of computation and coordination activities. 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 according to the procedure described in the previous section (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).

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 computation 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 (i.e., the set of concurrent logic programming rules) remains unchanged (we assume only that `detect_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* [6]. In that way, we can reuse the concurrent logic programming component for many similar coordination patterns. We have more to say about this in the next section.

A couple of points must be addressed at this stage however. The first has to do with comparing the MANIFOLD version of Section 2 with the IWIM-Linda version just described. One can hardly miss noticing the complexity of the latter and therefore wonder as to whether the approach advocated in this paper is really worth pursuing. We believe it is for the following reasons: (i) The simplicity of the first version is deceptive; the MANIFOLD version is syntactically simple but it invokes the underlying abstract machine which takes care of all these details which precisely come out in the open when a high-level realisation of IWIM is sought. (ii) Furthermore, as we discussed at some length in Section 3 and we will discuss again in Section 5, if one wants to employ sophisticated coordination patterns on top of the vanilla Linda model (a purpose, we believe, worth pursuing), one must be ready to accept the extra overhead incurred. However, we admit that one should also be concerned about expressiveness vs efficiency tradeoffs.

Which takes us to the second point in question: there is no reason why our IWIM rules cannot be interfaced to a general purpose Linda optimiser (such as the ones reported in [1,10,25]) which can draw upon its functionality and optimise certain operations. For instance, the system can recognise that outed tuples which constitute an IWIM channel or stream will be ined by a single process (by virtue of the IWIM functionality) and act appropriately.

Or, that a pair such as <in("estbl_str", Source, Target, StrId1), out("estbl_str", Source, NewTarget, StrId2)> which is used in the example of the following section to reconfigure stream connections, can be treated like a shared variable. Much of the functionality of the IWIM rules can already be mapped to predefined and more efficiently treated idioms or templates of tools such as the Linda Program Building Tool [1]; the rest can be taken care of by extending this or other similar tools.

4.5. A realistic case. The previous example served primarily to provide a detailed description of our framework. In this section we present the most important pieces of code for a rather more realistic example, its purpose being to highlight the reusability aspect of the coordination subcomponent of a non-trivial application. However, we also take the opportunity to introduce some additional important functionality such as the reconfiguration of streams required in dynamically evolving patterns of communication.

Our example is an adaptation and simplification of the one presented in [6] and involves the development of a bucket sorting program. There exists a sufficiently large number of atomic sorters, each atomic sorter a_i being able to sort very efficiently a bucket of k_i units (where, in fact, the number k_i may vary dynamically from one sorter to the next). Each atomic sorter receives the units to sort in its (default) input port, sorts them and produces the sorted sequence through its (default) output port. If its receives more than the k_i units it can handle efficiently, it raises the event `filled` to signify that its bucket is now full. Upon detecting the presence of `filled`, a coordinator process spawns another atomic sorter to handle the rest of the stream of units to be sorted as well as an atomic merger process which will merge the partial results of the old and the new sorter. Figure 3(a) shows the configuration with one sorter and 3(b) how the configuration evolves when a second sorter is added to the apparatus. Note that the whole procedure recurs and it is thus possible to have a number of sorters, all of them feeding their output to mergers and the latter (via a number of intermediate mergers) to the output process.

The psedo C-code for an IWIM-Linda compliant atomic sorter could be something like the following:

```
bucket_sort()
{
 int num_read=0, next_num=0, nums[LIMIT];

 while (next_num<LIMIT)              /* sorts efficiently LIMIT nums */
 {
  in("unit",input,?num_read);
  nums[next_num]=num_read;
  next_num++;
 }
```

```
  out("event",filled);                /* bucket is full */
  <sort numbers>
  for (next_num=0; next_num<LIMIT; next_num++;)
    out("unit",output,nums[next_num]);
}
```

Note that <sort numbers> refers to that part of the code which actually performs the sorting (it could in fact be a call to a purely computation function); it is not included here since the emphasis of the example is on how the various major components of the application get coordinated rather than on what they precisely do or how they do it.

We will not show the respective monitor process for bucket_sort() which is very similar to the ones presented for the previous example. Instead, we move directly to the presentation of the coordination process and we show the top clause.

```
coord_sorter(Ts,Prod_data,[Params1],       /* input process in fig. 3 */
             Comp_res,[Params2],           /* sorter process in fig. 3 */
             Arrange_res,[Params3],        /* merge process in fig. 3 */
             Cons_res,[Params4],           /* output process in fig. 3 */
             [Events_List],                /* includes 'filled' */
             ...) <-
    call(Prod_data,Params1), call(Comp_res,Params2),
    call(Arrange_res,Params3), call(Cons_res,Params4),
    out("estbl_str",Prod_data,out,StrId1),
    out("estbl_str",Comp_res,in,StrId1),      /* input -> sorter */
    out("estbl_str",Comp_res,out,StrId2),
    out("estbl_str",Cons_res,in,StrId2),      /* sorter -> output */
    coord_sorter1(...).

coord_sorter1([in("ack_estbl_str",Prod_data,out,StrId1),
               in("ack_estbl_str",Comp_res,in,StrId1)
               in("ack_estbl_str",Comp_res,out,StrId2)
               in("ack_estbl_str",Cons_res,in,StrId2),...|TS],...)
    <-    forward(TS,Prod_data,out,Comp_res,in,StrId1,0,0,[b,k],...),
          forward(TS,Comp_res,out,Cons_res,in,StrId2,0,0,[b,k],...),
          detect_event(TS,[(Comp_res,filled),...],...).
```

As for the case of the Fibonacci series example above, the top level coordinator spawns and activates the application components and sets up and activates the stream connection. Note that initially there is no merger process; this will appear once the raising of the event filled by the sorter process has been detected by the coordinator.

The careful reader may have already noticed that we do not "hardwire" into the code the processes involved (i.e., input, sorter, merge and output) but rather we pass them as arguments to the coordinator process and the latter spawns them as processes by means of using a metacall facility available in all logic programming frameworks. In fact, the same is done

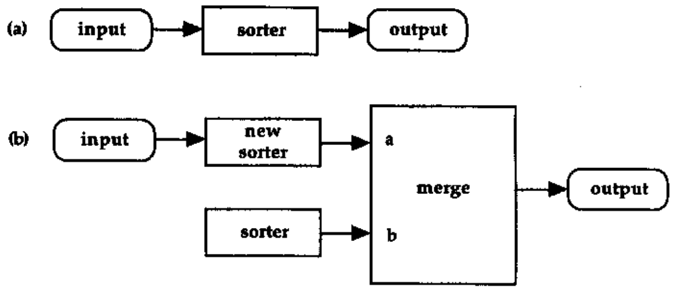

FIG. 3.

for the events involved in the scenario (namely `filled`). The reason for doing so will become apparent later on. We now show how the dynamic evolution of the configuration from the one shown above (which represents the scenario of Fig. 3(a)) to the one involving the spawning of another atomic sorter and a merger (which represents the scenario of Fig. 3(b)) is achieved.

```
detect_event(TS,(Comp_res,filled),          /* detect raising of 'filled' */
          Prod_data,[Params1],
          New_Comp_res,[Params2],
          Arrange_res,[Params3],
          Cons_res,[Params4],
          ...) <-
     call(New_Comp_res,Params2),                 /* spawn new sorter */
     call(Arrange_res,Params3),                  /* and a merger */

     in("estbl_str",Comp_res,in,StrId1),         /* break off existing */
     in("estbl_str",Comp_res,out,StrId2),        /* stream configurations */
     in("estbl_str",Cons_res,in,StrId2),

                                                /* and establish new ones */
     out("estbl_str",New_Comp_res,in,StrId1),    /* input -> new_sorter */
     out("estbl_str",New_Comp_res,out,StrId3),
     out("estbl_str",Arrange_res,a,StrId3),      /* new_sorter -> merge.a */
     out("estbl_str",Comp_res,out,StrId4),
     out("estbl_str",Arrange_res,b,StrId4),      /* sorter -> merge.b */
     out("estbl_str",Arrange_res,out,StrId5),
     out("estbl_str",Cons_res,in,StrId5),        /* merge -> output */
     coord_sorter2(...).

coord_sorter2([in("ack_estbl_str",...),...],...|TS],...)
     <-   forward(TS,...),...,
          detect_event(...).                           /* recur */
```

Once the raising of the event filled by `Comp_res` has been detected, a new `Comp_res` and an `Arrange_res` (namely merge) processes are spawned, the old stream connections are broken off and new ones are established.

The functionality of `coord_sorter2` is similar to that of `coord_sorter1` and there is no point in presenting its code in detail.

We notice once more that the coordination pattern just described is completely independent of the computation processes involved. In fact, as far as our coordination "laws" are concerned, little matters if the (atomic computation) processes involved perform sorting, merging or anything else. The benefits accruing from this complete decoupling of the computation from the communication/coordination component that IWIM encourages and enforces become apparent if we consider the following optimisation problem:

$$\max z = x^2 + y^2 - 0.5 * \cos(18 * y) \text{ with } (x,y) \text{ in the range } [-1.0, 1.0]$$

These types of problems are usually solved by means of domain decomposition techniques where a grid on the domain of the function is imposed splitting it into a number of sub-domains, as determined by the size of the grid. Next, good rough estimates for the highest values of z are obtained, they are further decomposed into smaller sub-domains and the whole procedure recurs until a sufficiently good estimate for z has been obtained.

In order to implement the above technique we need three main computation processes as follows. A `split` process receives as its parameters the specification of a grid (say 6×6) and, through its input port, a unit that describes a (sub-) domain and produces through its output port units describing the sub-domains obtained by imposing the grid on this input domain before terminating. The *i*th instance of a third process, `eval`, reads a bucket of $k_i > 0$ sub-domains via its input port and raises a specific event that it receives as a parameter, to inform other processes that it has filled up its input bucket with some sub-domains descriptions. It then finds the best estimate for the optimum z value in each of its sub-domains, producing an ordered sequence of units describing the best solutions it has found through its output port and terminates. Finally, a third process `merge` reads from its input ports `a` and `b` two ordered sequences of units describing sub-domains and their best estimates, and produces a sequence of one or more of its best sub-domains on its output port.

We need a coordination rule to coordinate the cooperation of `eval` and `merge` to solve the optimisation problem in a parallel/distributed fashion. It should feed units describing (sub-) domains from a `split` to an `eval` process up to the latter's limit and feed the rest to a new `eval` process before merging the two outputs and forwarding them to an output process. It is obvious that this functionality is almost identical to the one performed by `coord_sorter` above. Thus the coordination protocol of the domain decomposition example can be realised simply as follows:

```
coord_eval(TS) <- coord_sorter(split,[6,6],eval,[],merge,[],output,[],...),
                  forward(TS,[],[],split,in,_,0,[1,-1.0,-1.0,1.0,1.0],...).
```

5. Conclusions — related and further work. Conceptually, IWIM and Linda are very different models of coordination. At some fundamental level, they can be considered to be "equivalent" in the sense that each can emulate the other. They both provide symmetric primitives for anonymous communication. Linda provides a single shared dataspace as the universal medium of (asynchronous) communication among all processes, whereas IWIM provides (both synchronous and asynchronous) private channels. One major difference between the two is that Linda is data-oriented, whereas IWIM is a control-oriented model. The main advantage of IWIM is that it supports a clean separation of computation and coordination concerns into different program modules and encourages the decomposition of both computation and coordination tasks into hierarchies of small, reusable pure computation modules and small, reusable pure coordination modules. Furthermore, IWIM explicitly models the fate of "pending messages" in an asynchronous communication: a practically significant aspect of the behaviour of some real systems that is often neglected or made implicit in other models.

Whereas the generative communication advocated by Linda and variants is well suited to data-oriented applications, it is cumbersome to use in control-oriented ones [12]. Thus, the software engineering advantages inherent in the IWIM model makes it worthwhile to consider its implementation on top of Linda (and, more generally, in terms of the Shared Dataspace family of coordination models). Essentially, this involves overcoming and programming around most of the very same deficiencies in the vanilla Linda model faced by other researchers and practitioners reported in the literature. The derivation of what we refer to as the IWIM-Linda formalism has been the purpose of this paper.

As such, our work is directly related to a whole spectrum of research activities regarding the development of coordination models and languages [11]. Although we share the basic principles on which the other coordination frameworks are based, we pay particular attention to the issue of a clear and complete separation between computation and coordination/communication activities. We believe that an application should consist of a number of modules that clearly separate these two categories of activities; coordinator modules can in fact be reused with different computation ones. From this point of view, one can notice some major differences between Linda (or any other similar paradigm for that matter) on the one hand and IWIM-based formalisms such as IWIM-Linda on the other. The Linda model addresses only part of the underlying concerns of the IWIM family of coordination models. There is a symmetry between the communication primitives in Linda, and the communication between processes is accomplished anonymously through the Tuple Space. However, there is nothing to prevent complete mixing of communication concerns with computation. There is no clear separation of workers and managers, as in the IWIM family. Unlike Linda and similar models, the IWIM philosophy encourages programmers to develop "pure coordination modules" in their applications. This manifests the result of the substantial effort invested in the coordination component of an application in a tangible form as modular "pure coordinators" which can be reused by other applications.

Another significant difference between the underlying models of Linda and IWIM is that the former can be characterised as a more data-oriented one whereas the latter is a more control-oriented approach to coordination of the cooperation among concurrent processes. Because there is (conceptually) one flat Tuple Space in Linda, it is difficult to write meta-coordinators (i.e., coordinators that coordinate coordinators). Meta-coordinators however can be supported also by IWIM-Linda by means of techniques reminiscent of the ones used in creating multiple (and private to subsets of processes) Tuple Spaces (see relevant discussion below).

Our work is also in the spirit of [15] where the separation between computation and coordination/communication is also recognised as useful and enforced by means of *synchroniser* objects which are responsible for the coordination of ordinary computation ones. However, there are also some important differences: (i) we emphasize the complete lack of knowledge that some coordinated process need have about other fellow processes; (ii) our model is event driven whereas the one proposed in [15] is constraint based; (iii) the work presented here is effectively an abstraction of an already existing (and therefore tested) concrete programming environment [5] whereas [15] presents a high-level proposal whose implementation is under way. Another major difference between the two models is that synchronisers are different from the associated computation objects which means they cannot be used as normal objects, whereas in IWIM-Linda coordinators and worker processes behave in the same way and no external observer can distinguish between the former and the latter. Thus, it is a lot easier to compose hierarchies of IWIM-Linda coordinators than similar hierarchies of synchronisers. However, both schemes offer a compositional approach in the (re)use of coordinator code.

Our work is also somewhat similar in nature to Law-Governed Linda [18] 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 Architectures [17] in general, and in Law-Governed Linda in particular, and their role in coordination is also similar to the event mechanism supported by IWIM-Linda. 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* [13]. 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.

All in all, we notice that some of the ingredients of what constitutes IWIM-Linda can already be found in other proposed extensions of the vanilla model, even if the IWIM philosophy is not highlighted, explicitly enforced or encouraged. Thus, one can use the already developed know-how, coupled with the concepts contributed in this paper to develop a practical and efficient concrete IWIM-Linda environment. As we have said already, the proposed framework is currently under development using C-Linda and KLIC [14] to write the monitor and coordination processes. This implementation will be used not only to assess the practicality of our approach in the Linda context but also to compare it with a concrete realisation of IWIM, namely MANIFOLD. In the process, we will examine ways to derive a minimal IWIM subset which provides the required functionality; for instance, in practice we may not need in the IWIM-Linda context all five kinds of stream connections. In addition, we recall that the code written in KLIC, which is independent of any computation language as well as coordination model, effectively forms a set of coordination skeletons [23]. These skeletons can in fact be used as add-on IWIM coordinators in other models, thus providing the basis to extend the implementation work in ways that the more theoretical comparison described below may dictate.

This paper is only the tip of the iceberg regarding the relationship between IWIM and other coordination models and languages. We are currently pursuing an extensive study of other coordination models and languages [3,7,11,15,19,21,23,24] 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. Also, a reverse study of how the basic functionality of other coordination models and languages can be modelled within the IWIM philosophy is also interesting in assessing the expressiveness of the latter.

Finally, one notices that there are some interesting similarities between IWIM and the family of Module Interconnection Languages [20]. An interesting line of research would be to examine whether IWIM can be used as the basis for a formal analysis of MILs' functionality.

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

REFERENCES

[1] S. Ahmed, N. Carriero and D. Gelernter, "A program building tool for parallel applications", *DIMACS Workshop on Specifications of Parallel Algorithms*, Princeton Univ., May 1994.

[2] S. Ahuja, N. Carriero and D. Gelernter, "Linda and friends", *IEEE Computer* **19(8)**, 26 (1986).

[3] J.-M. Andreoli and R. Pareschi, "Linear objects: logical processes with built-in inheritance", *New Generation Computing* **9(3-4)**, 445 (1991).

[4] 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, 1996), pp. 34–56.

[5] F. Arbab, I. Herman and P. Spilling, "An overview of manifold and its implementation", *Concurrency: Practice and Experience* **5(1)**, 23 (1993).

[6] 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 (Springer Verlag, 1997), pp. 664–677.

[7] J.-P. Banatre and D. Le Metayer, "The GAMMA model and its discipline of programming", *Science of Computer Programming* **15**, 55 (1990).

[8] S. Bijnens, W. Joosen and P. Verbaeten, "Sender-initiated and receiver-initiated coordination in a global object space", *Object-Based Models and Languages for Concurrent Systems*, Bologna, Italy, 5 July 1994, LNCS 924 (Springer Verlag, 1995), pp. 14–28.

[9] A. Brogi and P. Ciancarini, "The concurrent language shared-prolog", *ACM Trans. on Programming Languages and Systems* **13(1)**, 99 (1991).

[10] N. Carriero and D. Gelernter, "New optimization strategies for the Linda pre-compiler", in *Linda-Like Systems and their Implementation*, Edinburgh Parallel Computing Centre, Technical Report 91-13, 1991, pp. 74–83.

[11] N. Carriero and D. Gelernter, "Coordination languages and their significance", *Commun. ACM* **35(2)**, 97 (1992).

[12] N. Carriero, D. Gelernter and S. Hupfer, "Collaborative applications experience with the Bauhaus coordination language", *30th Hawaii International Conference on System Sciences (HICSS-30)*, Maui, Hawaii, 7–10 Jan. 1997 (IEEE Press, 1997), pp. 310–319.

[13] 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, 1995), pp. 66–76.

[14] T. Chikayama, "KLIC User's Manual", ICOT, Japan, Oct. 1994, software obtained from http://www.icot.or.jp/ICOT/IFS/IFS-abst/ ifs-catalogue.html.

[15] S. Frølund and G. Agha, "A language framework for multi-object coordination", *European Conference on Object-Oriented Programming (ECOOP'93)*, Kaiserslautern, Germany, 26–30 July 1993, LNCS 707 (Springer Verlag, 1994), pp. 346–360.

[16] T. Kielmann, "Designing a coordination model for open systems", *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cesena, Italy, 15–17 April 1996, LNCS 1061 (Springer Verlag, 1996), pp. 267–284.

[17] N. H. Minsky, "The imposition of protocols over distributed systems", *IEEE Transactions on Software Engineering* **17(2)**, 183 (1991).

[18] 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, 1995), pp. 125–145.

[19] M. Rem, "Associons: A program notation with tuples instead of variables", *ACM Transactions on Programming Languages and Systems* **3(3)**, 251 (1981).

[20] M. D. Rice and S. B. Seidman, "A formal model for module interconnection languages", *IEEE Transactions on Software Engineering* **20**, 88 (1994).

[21] G.-C. Roman and H. C. Cunningham, "Mixed programming metaphors in a shared dataspace model of concurrency", *IEEE Transactions on Software Engineering* **16(12)**, 1361 (1990).

[22] E. Y. Shapiro, "The family of concurrent logic programming languages", *Computing Surveys* **21(3)**, 412 (1989).

[23] 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, 1996), pp. 78–85.

[24] A. Tanenbaum, F. Kaashouek and H. Bal, "Parallel programming using shared objects and broadcasting", *IEEE Computer* **25(8)**, 10 (1992).

[25] G. Wilson, "Improving the performance of generative communication systems by using application-specific mapping functions", in *Linda-Like Systems and their Implementation*, Edinburgh Parallel Computing Centre, Technical Report 91-13, 1991, pp. 129–142.