
ELSEVIER

Parallel Computing 24 (1998) 1137–1160

PARALLEL COMPUTING

Distributed and parallel systems engineering in MANIFOLD

George A. Papadopoulos *

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

Received 15 May 1997; revised 10 October 1997

Abstract

A rather recent approach in programming parallel and distributed systems is that of coordination models and languages. Coordination programming enjoys a number of advantages such as the ability to express different software architectures and abstract interaction protocols, supporting multilinguality, reusability and programming-in-the-large, etc. However, most of the proposed models and languages are data-driven in the sense that changes in the behaviour of the formalism are triggered by detecting the presence and examining the nature of data values. In addition, more often than not, the formalism does not clearly separate the computation components from other related components, namely (and primarily) communication ones, but also synchronisation components, etc. In this paper, we use a coordination model (IWIM) and language (MANIFOLD) which are control-driven in the sense that the formalism's change of behaviour is modelled as state transitions triggered by means of raising events and detecting their presence, i.e., without involving the actual data being manipulated. We illustrate the main features of this formalism and we show how it can be used in supporting a variety of activities related to distributed and parallel software engineering, and software architectures. Throughout, we place emphasis on the control-driven nature of this formalism, discussing how that has helped us in modelling a variety of scenarios. Finally, we also compare the formalism with other such formalisms highlighting the differences between them.

Keywords: Coordination languages and models; Distributed and parallel computing models and tools; Software engineering for distributed and parallel systems; Modelling software architectures

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

0167-8191/98/$ - see front matter
PII: S0167-8191(98)00045-3

1. Introduction

Massively distributed and parallel systems open new horizons for large applications and present new challenges for software technology. Many applications already take advantage of the increased raw computational power provided by such parallel systems to yield significantly shorter turn-around times. However, the availability of so many processors to work on a single application presents a new challenge to software technology: coordination of the cooperation of large numbers of concurrent active entities. Classical views of concurrency in programming languages that are based on extensions of the sequential programming paradigm are ill-suited to meet this challenge.

Furthermore, it has been recently recognised within the Software Engineering community, that when systems are constructed of many components, the organisation or architecture of the overall system presents a new set of design problems. It is now widely accepted that an architecture comprises, mainly, two entities: *components* (which act as the primary units of computation in a system) and *connectors* (which specify interactions and communication patterns between the components).

Exploiting the full potential of massive parallel systems requires programming models that explicitly deal with the concurrency of cooperation among very large numbers of active entities that comprise a single application. Furthermore, these models should make a clear distinction between individual components and their interaction in the overall software organisation. In practice, the concurrent applications of today essentially use a set of ad hoc templates to coordinate the cooperation of active components. This shows the need for proper *coordination languages* [1] or *software architecture languages* [2] that can be used to explicitly describe complex coordination protocols in terms of simple primitives and structuring constructs.

In particular, a coordination language should be able to support, among other things, the following: distribution (no intrinsically global features), open endedness (dynamic creation and connection of services), activity, reactivity and proactivity, multiparadigm interoperability, fault tolerance, reusability and programming-in-the-large (i.e., provision of constructs able to express software architectures). The above mentioned targets should be met in ways that allow both time and space coordination to be expressed; clearly separate the computation parts of an application from others (such as coordination or synchronisation ones); and support multilingual programming and perform optimisations at all possible levels (including the coordination one).

A number of such coordination models and languages have evolved over the last few years such as Linda and related formalisms based on the Shared Dataspace approach [3–7], Gamma based on Multiset Rewriting [8], Linear Objects [9] based on Linear Logic and the framework of Interaction Abstract Machines [10], UNITY and (various types of) Skeletons based on functional programming [11–13], PCN and related formalisms based on (concurrent) logic programming [14], the Programmer's Playground based on I/O abstractions [15], etc. As it should be expected, these formalisms differ in a number of ways in their attempt to meet the above mentioned targets. However, almost all of these formalisms share two common characteristics:

1. They are *data-driven*, in the sense that changing the current state of the computation involves detecting the presence of data values and, possibly, examining their actual

contents. Thus, data play the dual role of being information exchange channels between the concurrently executing agents, as well as control triggering mechanisms which change the state of the computation. This duality makes it often hard to build programs that are easy to understand, implement and optimise and compromises many of the above mentioned goals.

2. Furthermore, and to a certain extent as a consequence of the above problem, these models do not offer a crystal clear separation between the computation component and other types of components (such as communication and synchronisation ones) that all together make up an application environment. In particular, both at a syntactic (linguistic) and semantic level, the computation code is interspersed throughout a program or a collection of associated modules with communication, coordination and synchronisation code. As it should be expected, this further compromises the goals that coordination models and languages have been set out to meet.

Many applications are by nature event-based (rather than data-driven) where software components interact with each other by posting and receiving events, the presence of which triggers some activity (e.g., the invocation of a procedure). Events provide a natural mechanism for system integration and enjoy a number of advantages such as: (i) waiving the need to explicitly name components, (ii) making easier the dynamic addition of components (where the latter simply register their interest in observing some event(s)), and (iii) encouraging the complete separation of computation from communication concerns by enforcing a distinction of event-based interaction properties from the implementation of computation components. Event-based paradigms are natural candidates for designing coordination rather than programming languages; a 'programming language based' approach does not scale up to systems of event-based components, where interaction between components is complex and computation parts may be written in different programming languages.

Thus, there exists a second class of coordination models and languages, which is control-driven and state transitions are triggered by raising events and observing their presence. Typical members of this family are MANIFOLD [16–18], which will be the primary focus of this paper, but also ConCoord [19] and TOOLBUS [20]. Furthermore, by advocating a somewhat liberal notion of what coordination actually is, we can also include in this family *software architecture* or *configuration* languages such as Conic [21], Durra [22], Darwin/Regis [23], POLYLITH [24] and Rapide [25]. Contrary to the case of the data-driven family where coordinators directly handle and examine data values, here, processes are treated as black boxes; data handled within a process is of no concern to the environment of the process. Processes communicate with their environment by means of clearly defined interfaces, usually referred to as *input* or *output ports*. Producer–consumer relationships are formed by means of setting up *stream* or *channel* connections between output ports of producers and input ports of consumers. By nature, these connections are *point-to-point*, although *limited broadcasting* functionality is usually allowed by forming $1-n$ relationships between a producer and n consumers and vice versa. Certainly though, this scheme contrasts with the Shared Dataspace approach usually advocated by the coordination languages of the data-driven family. A more detailed description and comparison of these two main families of coordination models and languages can be found in Ref. [26].

In this paper, we use the generic model of communication Ideal Worker Ideal Manager (IWIM) and a specific *control-oriented event-driven* coordination language (MANIFOLD) based on this model [16–18]. The important characteristics of IWIM include compositionality, inherited from the data-flow model, anonymous communication for both producers and consumers, evolution of coordination frameworks by observing and reacting to events and complete separation of computation from communication and other concerns. These characteristics lead to clear advantages in large concurrent applications.

In particular, we present by means of examples the way MANIFOLD can be used in developing software environments for distributed and parallel systems. The rest of the paper is organised as follows. Section 2 is a brief presentation of the coordination model IWIM and associated language MANIFOLD. Sections 3 and 4 illustrate the usefulness of the framework: Section 3 is focused on using MANIFOLD for distributed and parallel programming, while Section 4 is concerned with software architecture and engineering issues. The paper ends with some conclusions, comparison with related work and short reference to our future activities.

2. The IWIM model and the language MANIFOLD

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: (1) 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. (2) 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 hard-wired 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 communication model 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 intercommunication 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 interconnectionsbetween 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.

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.

Note that the last four channel types effectively model variations of asynchronous communication. Note also that any units pending within a channel which has been disconnected from either end will remain there and resume their flow once the channel is reconnected to some other process.

MANIFOLD is a coordination language which can be seen as a concrete version of the IWIM model just described where:

1. each of the basic concepts of process, port, event and channel corresponds to an explicit language construct;
2. communication is asynchronous and raising and reacting to events (or signals) enforces no synchronisation between the processes involved. Thus, MANIFOLD realises all but the SS (synchronous) types of IWIM channels.
3. The asynchronous channels of IWIM are called streams in the language and represent a reliable and directed flow of information. Once a stream is established between a producer process and a consumer process, it operates autonomously and transfers the units from its source to its sink. The sink of a stream is suspended only if no units are available for its consumption and resumes once the next unit becomes available for consumption. The source of a stream is never suspended because the infinite buffer capacity of a stream is never filled.
4. The separation between communication and computation, i.e., the distinction between workers and managers, is more strongly enforced.

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

In this section, we have deliberately presented MANIFOLD from the point of view of its underlying model (namely IWIM). More information on the actual language can be found in Refs. [16,17,27,18] and in the introduction of this special issue. We will also have the opportunity to present a number of features that the language supports, while describing the examples in the main part of this paper.

The purpose of the next two sections is to present a number of techniques which model a variety of activities related to the development of distributed and parallel systems using MANIFOLD. The aim here is twofold: (i) to further illustrate the capabilities of this coordination formalism in playing the role of a coordination

framework for distributed and parallel systems (at least as we perceive such a role to be), and (ii) to present a number of techniques that we are developing in using the model to design and implement non-trivial but also typical applications such as fault tolerant systems and distributed multimedia environments.

We believe that a control-driven or event-driven coordination language such as MANIFOLD can be used in application domains where, traditionally, either data-driven or control-driven coordination languages are being used, but not both. The former domain is concerned primarily with parallelising programs, whereas the latter one is more interested with issues related to modelling software architectures. To illustrate this dual capability of MANIFOLD, we split the main part of the paper in the above mentioned two sections.

3. Coordinating activities in distributed and parallel systems

3.1. Fault tolerance

Fault tolerance is an important aspect of distributed and parallel systems. A number of techniques have evolved over the years in providing applications with fault tolerance such as N modular redundancy, backward error recovery, etc. However, one may notice that the issues related to developing and testing an environment for fault tolerance are really orthogonal to those related with the actual computation the environment will produce. Nevertheless, when an ordinary language is used those issues are intermixed, making the development of both the computation and the fault tolerant parts more difficult.

Using a coordination language like MANIFOLD for fault tolerance is ideal. The elaborated constructs that MANIFOLD supports, the philosophy that port-to-port connections are secure, the event control-based state transitions the language advocates and its dynamic reconfiguration features are well suited to the development of generic fault tolerant frameworks that can cover all dimensions of fault tolerance: software failures, processor (device) failures, communications failure, etc. In the sequel, we are presenting the principles of such a fault tolerant generic framework based on the well known TMR (Triple Modular Redundant) algorithm [28]. The basic idea is the following: Each one of the three modules executes separately and passes its result to a voter module which, after getting results from all three modules, decides on the actual outcome of the computation. The most important parts of the code follow promptly.

```
manifold Producer( ) import.
manifold AtomicVoter( ) import.

manifold Vote( )
{
  event rec_from_1, rec_from_2, rec_from_3,
        tuple_rec, not_rec, get_from_1, get_from_2,
        get_from_3, repeat.
```

```
  process prod1 is Producer(manifold, event get_from_1).
  process prod2 is Producer(manifold, event get_from_2).
  process prod3 is Producer(manifold, event get_from_3).
  process Voter is AtomicVoter( ).

stream KB reconnect prod1 → voter.
stream KB reconnect prod2 → voter.
stream KB reconnect prod3 → voter.

  begin: (activate(prod1,prod2,prod3,voter), post(repeat)).

  repeat: (prod1 → tout1, prod2 → tout2, prod3 → tout3,
           guard(tout1,transport,rec_from_1), alarm(tout1,
           30, not_rec),
           guard(tout2,transport,rec_from_2), alarm(tout2,
           30, not_rec),
           guard(tout3,transport,rec_from_3), alarm(tout3,
           30, not_rec)).

  rec_from_1&
  rec_from_2&
  rec_from_3: ((tout1 → , tout2 → , tout3 → ) → voter,
               raise(get_from_1, get_from_2, get_from_3),
               terminated(self)).

not_rec: ⟨exception handling⟩.

tuple_rec.voter: post(repeat).
}

manifold Producer(manifold Module, event get_from).
{
  event received, repeat.
  process produce is Module.

  stream KB Producer → produce.

  begin: (activate(produce),
          post(repeat)).

  repeat: (produce → Producer,
           guard(input,transport,received)).

  received: (input → output,
             terminated(self)).

  get_from: post(repeat).
}
```

In our framework, all four entities (the three modules and the voter) are black boxes communicating with their environment by means of port connections. In particular, there exists one manifold for each entity plus a global coordinator. Each manifold monitoring a module receives the next datum and forwards it to its output port. It then suspends (by means of executing the `terminated(self)` operation which never terminates) until it receives a message `get_from_1` (or 2 or 3) from the voter in which case it repeats the process. The voter after setting up the required initial stream connections suspends waiting to receive three values from the respective modules or the elapse of 30 s (arbitrary value which may change at will) signifying that something has gone wrong at either the software level or the hardware level (e.g., breaking up of some communication link). This is achieved by means of setting appropriate guards which monitor activity in a port (e.g., the expression `guard(port_id,transport,passed)` posts the event passed if a unit has passed through the port `port_id`). In the latter case, some appropriate exception handling takes place; this may involve removing the faulty component and/or activating a new one, possibly on another processor (see also next example). Upon receiving the three values, the voter manifold passes them on to the actual voter for processing. The actual voter then sends a `rec_received` message to the manifold voter and the latter repeats the process. Note here that streams have been set up as KB reconnect since they are broken from and reconnected back to the voter every time a triplet of data must be received (other more efficient techniques for communicating data, than the 'wave forwarding' one used here, can also be supported).

Note the following: (i) being black boxes, the three modules and the voter may be written in any language, or indeed in different ones; (ii) changes in the actual computation performed by these modules do not affect in any way the coordination and synchronisation parts of our apparatus - thus, it is possible to change the logic, of the voter, incorporate deliberately wrong values in the modules ('fault injection'), etc., (iii) the framework presented above is itself fault tolerant - communication is secured (thanks to MANIFOLD's philosophy), any breaking of communication links or hardware faults can be detected (by means of incorporating suitable device drivers which raise error signals to whose presence the MANIFOLD apparatus reacts appropriately) and, in fact, all modules run on different machines to minimise the possibility of total collapse resulting from hardware failure. This last functionality is achieved by means of suitable directives set in an associated 'config.map' file like the following:

```
{host host1 afrodite.cs.ucy.ac.cy venus.cs.ucy.ac.cy}
{host host2 zeus.cs.ucy.ac.cy}
{host host3 atlas.cs.ucy.ac.cy}
{host host4 venus.cs.ucy.ac.cy}
{arch anyRS RS/6000}
{locus voter_task $host1}
{locus prod1_task $host2}
{locus prod2_task $host3}
{locus prod3_task $anyRS}
```

where task instances (i.e., executable files), one for each incarnation of a voter or a producer process, run on a number of IBM RS/6000 machines. In particular, the

scenario above says that voter_task will run on the machine afrodite of the domain cs.ucy.ac.cy. and if this machine is not available, voter_task will run on venus. In contrast, prod3_task will run on any available machine of the given architecture, whereas the other tasks will run on the particular machines as specified.

In similar ways, we can model other fault tolerant schemes. The second example shows the principles of implementing backward error recovery, another popular fault tolerant technique.

```
manifold variable import.
manifold Producer( ) import.
manifold Consumer( ) import.

manifold Main( )
{
  event error.
  auto process state is variable.
  auto process producer is Producer.
  auto process consumer1(port out sstate, event error)
                         is Consumer.
  auto process consumer2(port in rstate) is Consumer.

  stream KB reconnect producer → *.

  begin: (producer → consumer1, consumer1.sstate → state,
          terminated(self)).
  error.consumer1: (producer → consumer2,
                    state → consumer2.rstate).
}
```

In the above program, a producer initially sends its output to a consumer for further processing. The latter one periodically sends in the form of a tuple its state to the variable state (which is another manifold as far as the language is concerned). Upon detecting an error (whether software or hardware), consumer1 raises an appropriate signal and the main coordinator switches the output of consumer1 to the backup consumer2. At the same time, the most recently saved state of consumer1 is fed into consumer2 and computation falls back to the last error-free point. Again, as in the previous example, producer is completely unaware of such dynamic changes and, more generally, this fault tolerant policy does not affect, in any way, the actual computation work performed by the three processes and, thus, it can be changed without the need to modify the internals of the processes involved. The program can be made more tolerant to hardware and communication faults by choosing to place every manifold involved (including the 'variable' state) on different processors; this can be achieved as indicated in the description of the previous example.

The above scenario supports *single fault tolerance* in the sense that only one catastrophic failure of a consumer process can be handled. However, it is also possible to model a more dynamic scenario involving many levels of fallback. It is beyond the scope of this paper to address such issues as the tradeoff between the frequency of state savings and the incurred overhead, or ways to handle the so called *domino effect* whereby, the fallback of one process to a previous state causes other processes to do the same. However, we note that these issues are primarily coordination rather than computation issues. Thus, we believe that it would be more appropriate to model them at the MANIFOLD level and in this way have the ability to modify the fault tolerant policy without affecting the computation code in any way.

3.2. Incremental software development

Traditionally, coordination languages are used to 'glue together' existing components, thus, enhancing reusability. Although MANIFOLD can of course play this central role, it can do even more. The well-defined elaborate interfaces between the components that can be specified by means of the plethora of primitives the language supports, and the control-driven change of state it advocates, make the language suitable for 'creative development' of completely new software environments. The main concern in such a development is how to resolve quickly and efficiently issues such as proper communication and synchronisation between the concurrently executing components; usually the pure computation part of the code is not that hard to develop and debug if it is free from other concerns.

MANIFOLD can assist this process of development by separating completely the computation from all the other parts of the code. Such a to-be-created environment can be viewed as a number of initially large black boxes. MANIFOLD's constructs like port connections and raising of events can be used to set up well-defined interfaces between the boxes which, initially, can be just mock-ups of the actual computation parts of the application. Gradually, the black boxes split into smaller ones and again the same process of first establishing well-defined communication and synchronisation patterns before producing the computation part is followed. Indeed, optimisation tests (as, for instance, in determining communication costs) can also be performed by modifying these patterns without any negative effects arising from the computation part being still unknown. Eventually, or in parallel with this process, the black boxes are filled with the actual computation part which would be mainly sequential and thus, easier to debug.

As an example, we consider the case of a control module processing the output produced by a set of workstations which is transferred from the latter to the former via a communication subsystem. Results of this processing are displayed by the workstations after their transfer back to them. A possible specification of the three main entities comprising this apparatus is the following:

```
CONTROL          MESSAGE_HANDL      WORKSTATION
Input Unit       Input Unit         Manager
Output Unit      Output Unit        Tools
Proc Unit        Handler            Viewer
```

An initial configuration in MANIFOLD is the following one:

```
manifold CONTROL( ) import.
manifold MESSAGE_HANDL( ) import.
manifold WORKSTATION( ) import.

process control is CONTROL( ).
process handler is MESSAGE_HANDL( ).

process ws1 is WORKSTATION( ).
process ws2 is WORKSTATION( ).
process ws3 is WORKSTATION( ).

manifold Main( )
{
   begin: (activate(control,handler,ws1,ws2,ws3),
           control → handler → ( → ws1, → ws2, → ws3),
           (ws1 → ,ws2 → ,ws3 → ) → handler → control).

   ⟨set up event communication to define precisely the
    communication patterns to be employed⟩
}
```

Later on, the module WORKSTATION can be further refined as follows:

```
manifold MANAGER import.
manifold TOOLS import.
manifold VIEWER import.

process manager is MANAGER( ).
process tools is TOOLS( ).
process viewer1 is VIEWER( ).
process viewer2 is VIEWER( ).

export manifold WORKSTATION( )
{
   begin: (activate(manager,tools,viewer1,viewer2),
           manager → tools → ( → viewer1, → viewer2),
           (viewer1 → ,viewer2 → ) → tools → manager).
   ⟨set up event communication to define precisely the
    communication patterns to be employed⟩
}
```

Etc. for the other components. Note that any subcomponent can be defined in more or less detail at will. Note also that different connections between replicated components (such as workstations or viewers) can be employed to determine what is the most efficient and effective configuration.

4. Software architecture and software engineering issues

4.1. Anonymous communication via S / W multiplexers

MANIFOLD supports point-to-point communication between concurrently executing agents. However, often it is desirable for a number of agents to communicate with each other in a conferencing mode, i.e., whatever output one produces is replicated to all other agents. This functionality is useful in applications such as teleconferencing. In coordination models which use a common shared communication medium, this can be done trivially by having each agent post information to the medium while the rest of the agents retrieve it. However, this simple scenario is not without its disadvantages: the information is usually physically replicated to the private memory of all agents accessing the public forum (whether they are actually participating in this particular data exchange or not); furthermore, since the shared medium is public and insecure, there is no guarantee that the information to be exchanged is not lost, intercepted, forged or altered in any other undesirable way. To solve these problems, sophisticated techniques must be employed [7]. MANIFOLD however can address these issues without encountering the above mentioned problems.

We now present a solution to the following scenario. An agent wants to join a conferencing session and be able to both talk and listen to all other agents. However, we do not require each agent to know the 'addresses' or 'ids' of the other agents. Furthermore, we want to route information directly to the agents concerned without unnecessary re-routing and use of centralised information relay nodes. In addition, this goal should be met in a way that would not force each agent to repeat the connection/disconnection process for every other agent engaged in the conferencing session.

The following code implements the scenario just described.

```
event leave.

manifold Session(event) import.

manifold Connect(process p1, process p2)
{
  begin: (p1 → p2, p2 → p1,
          terminated(self)).
  leave.p1|leave.p2: .
}
```

```
manifold Participate(proess me)
{
  ignore join.me.

  begin: while true do
           {
             begin: terminated(self).
             join.*other: Connect(me,other).
           }.
  leave.me: .
}

manifold User( )
{
  event remove_me.

  begin: (Participate(self),
          raise(join),
          Session(remove_me),
          terminated(self)).
  remove_me: raise(leave).
}
```

A few remarks are again in order. `Session` represents the actual conferencing session between each participant and the rest of the world; its definition is described elsewhere and, for all that matters, it can be implemented in MANIFOLD itself or some other programming language. The only aspect of `Session` relevant to the rest of this discussion is that it takes as a parameter, an event which it will raise when it wishes to terminate the session between the participant and the rest of the conferencing group. `Connect` is a simple manifold, which takes a pair of processes as parameters, and connects the output port of the first process to the input port of the second process and vice versa, thus establishing to and from communication between the two processes. Upon detecting the presence of a `leave` signal, raised by either of the two processes on behalf of which it has established mutual connections, and signifying the wish of the signal's producer to leave the session, `Connect` breaks off the stream connections between these two processes. `Participate` is a monitoring manifold; there exists one such manifold for every participant in a conferencing session. It is responsible for detecting the arrival of new participants and creating a `Connect` process to set up additional stream connections between every newly arrived participant and the process that `Participate` monitors. More to the point, it recurs within a loop, initially suspended, waiting for the arrival of a `join` signal raised by some participant-to-be process, other than the one that it monitors. Upon detecting such a signal, it spawns a `Connect` process passing to the latter as parameters the ids of both its own process that

it monitors and the process that raised join. `Participate` will exit the loop and terminate when it detects the presence of the signal `leave`, raised by the process it monitors. Finally, `User` represents a participating party. Upon commencing execution, it will activate a `Participate` process which will be responsible for connecting this party to any other participant that will possibly arrive in the future. It will also raise the signal `join`, thereby making the other participants connect to this party, and enter the conferencing session. When the `remove_me` signal is raised by `Session`, the corresponding `User` process itself raises the signal `leave`, for the benefit of its monitoring process `Participate`, and terminates.

Note that the joining and leaving of participants is done in a distributed and asynchronous fashion. There is no centralised control, either for the setting up of the stream connections or the forwarding of data between the participants, and each member of the conferencing group is responsible for connecting itself to the newly arriving processes. Furthermore, the actual conferencing part of the code, namely `Session`, is completely unaware of the dynamic changes that take place in its environment. Elementary graceful withdrawal of participants is guaranteed by the default behaviour of a stream connection which is BK, meaning that even when the producer breaks up a stream connection, the stream will still remain connected to the arrival side, thus allowing the consumer to retrieve any units of information pending within the stream. More sophisticated control over the data distributed via the set up streams is possible by installing guards on the input/output ports of the processes involved, and thus monitoring the status of stream connections in order to act accordingly. Finally, we note that data travel directly to the participating parties, and the only broadcasting operations are those for raising a couple of events, which is a very cheap operation in MANIFOLD.

4.2. Hybrid coordination of S / W And H / W components

The next example shows how MANIFOLD can be used to coordinate objects which are not necessarily software modules, i.e., they could be hardware devices, and which do not necessarily produce discrete data, i.e., they could produce video or audio streams of continuous data. Imagine the following rather generic scenario: A control module is connected to a sensors module and both are connected to a display device. The sensors module outputs simulated changes in various readings produced by sensor devices and the control module, based on these readings, regulates the operation of certain devices. Both modules send some data to the display device for presentation in suitable forms. The outline of part of the code is shown below.

```
manifold SENSORS port in heating, flow.
                 port out temperature, volume, concent.
                 elsewhere.

manifold CTRL_MOD port in temperature, volume, concent.
                  port out value,valve.
                  elsewhere.
```

```
manifold DISPLAY port in heating, flow, concentration,
                 level, temperature.
                 elsewhere.

manifold Main( )
{
  process comp_sensors is SENSORS.
  process comp_control is CTRL_MOD.
  process comp_display is DISPLAY.

  begin: (activate(comp_sensors, comp_control,
                   comp_display),
          comp_sensors.temperature →
                   (→ comp_control.temperature,
                    → comp_display.temperature),
          comp_sensors.volume → ( → comp_control.volume,
                                  → comp_display.volume),
          comp_sensors.concent →
                   (→ comp_control.concent,
                    → comp_display.concentration),
          comp_control.value → ( → comp_sensors.heating,
                                 → comp_display.flame),
          comp_control.valve → ( → comp_sensors.flow,
          terminated(self)).     → comp_display.flow),
}
event send_values.

manifold Compute_Sensors port in i1, i2.
                         port out o1, o2, o3.
                         {atomic. event send_values}.

manifold SENSORS( )
{
  process sensors ⟨i1, i2|o1, o2, o3⟩ is Compute_Sensors.
  begin: (activate(sensors),
          heating → sensors.i1,
          flow → sensors.i2,
          sensors.o1 → temperature,
          sensors.o2 → volume,
          sensors.o3 → concent
          terminated(self)).
  ...⟨detect various events and react accordingly⟩...
  death.sensors: halt.
}
```

```
#include "AP_interface.h"
void Compute_Sensors( )
{
  AP_Event send_values;
  int flow,volume,concent;
  float heating,temperature;
  int i1,i2,o1,o2,o3;

  i1=AP_PortIndex( "i1");
  i2=AP_PortIndex( "i2");
  o1=AP_PortIndex( "o1");
  o2=AP_PortIndex( "o2") "
  o3=AP_PortIndex( "o3");

  AP_InitHeaderEvent(send_values, "send_values");

  while (1)
  {
    heating=AP_PortGetUnit(i1);
    flow=AP_PortGetUnit(i2);

    ⟨compute temperature, volume and concentration⟩

    AP_Raise(send_values);
    AP_PortPutUnit(o1,temperature);
    AP_PortPutUnit(o2,volume);
    AP_PortPutUnit(o3,concentration);

    ⟨handle events⟩

    if (termination_condition_satisfied) return;
}
⟨similar code for the other two modules⟩
```

The main points that the above code attempts to highlight are the following. There is a main coordinator which sets up the stream connections between the three modules. It is completely irrelevant as far as the coordinator is concerned, what type of values these modules produce or what their own nature actually is. For every such module we have two entities: a MANIFOLD monitoring process (like SENSORS) responsible (among other things) for the module's proper connection with the rest of the world and the actual computation process (like Compute_Sensors) which in this case is a C program. Note that this program is 'MANIFOLD compliant' meaning that by using the AP_interface.h environment it is able at the C (or indeed any other language) level to receive from or send to ports data, raise events or detect their presence, etc. This need

not be the case; an ordinary program (i.e., any Unix-level process) would still be able to communicate with its environment, albeit with a significantly less rich functionality.

Note that our framework can evolve by adding or removing components without affecting the state of affairs. For instance, assume that at some point in time we want to reforward some of the displayed data to a printing device. The user could execute a manifold process like the following one:

```
manifold send_to_printer(port out source, manifold
                 printer_id)
{
  begin: source → printer_id.
}
```

and invoke it as follows:

```
send_to_printer(comp_display.level,printer_id),
send_to_printer(comp_display.temperature,printer_id).
```

This would create additional streams to the printing device; upon termination of printing, the connections can be broken up without affecting the rest of the information flow between the other modules.

5. Conclusions. Related and further work

Malone and Crowston [29] characterise coordination as an emerging research area with an interdisciplinary focus. They observe that coordination has been and is a key issue in many diverse disciplines other than Computer Science. Although tackling coordination problems in operating systems, parallel programming, and databases (to name but a few) has a long history in Computer Science, the notion of coordination as a research area and coordination languages as a serious topic are rather recent developments. Nevertheless, a number of models and systems have already appeared for coordination. Many of them deal with some particular aspect of coordination, or coordination in a specific and somewhat limited context.

One of the best known coordination languages is Linda [3,5] which uses the so called *generative communication* model based on the Tuple Space (which is itself an instance of the Shared Dataspace model). The Tuple Space is, at least conceptually, centrally managed and contains all pieces of information that processes want to communicate. The Tuple Space exists outside the lifespan of the processes constituting a computation which are treated as black boxes. Accessing the Tuple Space is achieved by means of only four simple primitives which effectively constitute a 'coordination assembly language'. However, this simplicity is also Linda's weak point. The 'vanilla' model does not support point-to-point communication nor does it guarantee any form of security while accessing the Tuple Space which is a public forum. This has led to a

number of extensions [4,7] that try to address these issues and ameliorate the associated problems. Furthermore, the programming style encouraged by Linda and associated models suggests using shared dataspace access primitives directly in the computation code. This mixes communication code with computation code. Finally, there is no clear separation between workers and managers as in IWIM.

The metaphor of Interaction Abstract Machines [10] and its underlying formal computational model of Linear Objects [9], present a paradigm for abstract modelling of concurrent agent-oriented computation. The operational semantics of the agents and their interactions are given in terms of the proof theory of Linear Logic whereas the 'property driven communication' the model employs is analogous to MANIFOLD's port connections.

Gamma [8] is another coordination model based on non-deterministic multiset rewriting. It provides a framework in which programs can be expressed with a minimum of explicit control and where, ideally, efficient execution schedules for such high-level program specifications can be found automatically.

All the above models have in common the use of a shared, open, unrestricted data structure (whether it is called Tuple Space, Blackboard, MultiSet, etc.) as an appropriate medium for communication in a distributed or parallel data-driven system. But, this shared medium also becomes a synchronisation mechanism during the concurrent execution of the processes involved in a computation. Whatever coordination primitives each model uses are mixed throughout the program with computation code. Communication via the shared common medium is by no means secured (at least as far as the 'vannila' versions of each model are concerned) and it is also difficult to optimise these models so that data go only to the destinations that actually need them.

In Ref. [15], while describing another coordination mechanism based on I/O abstractions, a number of desirable properties that coordination models should possess are listed. These properties are active and reactive communication, connection-oriented and user-specifiable configuration and support for a variety of communication schemes such as implicit, direct, multiway, and use of continuous streams. It is worth mentioning here that IWIM and MANIFOLD support all these schemes as first class citizens. In addition, IWIM and MANIFOLD support complete separation of computation from coordination concerns and a control-driven specification of system transformations, which unlike I/O abstractions, is, in our opinion, more appropriate for coordination programming.

Note that in IWIM and MANIFOLD, unlike in many other coordination models and languages, a component is oblivious not only to bindings produced by other components but also to whether or not communication is taking place at all or what type of communication this is. This frees the programmer from having to establish when it is the best moment to send and/or receive messages. And of course, the language enjoys the ability for dynamic system reconfiguration without the need to disrupt services or the components having mutual knowledge of structure or location - point-to-point or multicast communications can be configured independently of the computation activity and mapped appropriately onto the underlying architecture.

Furthermore, the stream or channel connections that IWIM and MANIFOLD support as the basic mechanism for communication between computation components, provide a

natural abstraction for supporting continuous datatypes such as audio or video and make this coordination model and its associated language ideal for coordinating the activities in, say, distributed multimedia environments. We are currently exploiting this characteristic of MANIFOLD in a recently commenced research project where the language will be used to manage and coordinate, among other activities, the data produced or consumed by media servers.

MANIFOLD is a typical member of the control-driven family of coordination models [26]. Although many of the concepts found in MANIFOLD have been used in other control-oriented coordination languages, MANIFOLD generalises them into abstract linguistic constructs, with well-defined semantics that extends their use. For instance, the concept of a port as a first-class linguistic construct representing a 'hole' with two distinct sides, is a powerful abstraction for anonymous communication: normally, only the process q that owns a port p has access to the 'private side' of p, while any third party coordinator process that knows about p, can establish a communication between q and some other process by connecting a stream to the 'public side' of p. Arbitrary connections (from the departure sides to the arrival sides) of arbitrary ports, with multiple incoming and multiple outgoing connections are all possible and have well-defined semantics. Also, the fact that computation and coordinator processes are absolutely indistinguishable from the point of view of other processes, means that coordinator processes can, recursively, manage the communication of other coordinator processes, just as if they were computation processes. This means that any coordinator can also be used as a higher-level or meta-coordinator, to build a sophisticated hierarchy of coordination protocols. Such higher-level coordinators are not possible or are hard to define in many other coordination languages and models.

MANIFOLD advocates a liberal view of dynamic reconfiguration and system consistency. Consistency in MANIFOLD involves the integrity of the topology of the communication links among the processes in an application, and is independent of the states of the processes themselves. Other languages, such as Conic [21], limit the dynamic reconfiguration capability of the system by allowing evolution to take place only when the processes involved have reached some sort of a safe state (e.g., quiescence). MANIFOLD does not impose such constraints; rather, by means of a plethora of suitable primitives, it provides programmers the tools to establish their own safety criteria to avoid reaching logically inconsistent states. Furthermore, primitives such as guards, installed on the input and/or output ports of processes, inherently encourage programmers to express their criteria in terms of the externally observable (i.e., input/output) behavior of (computation as well as coordination) processes.

Structuring of programs is another area where MANIFOLD is prepared to allow the programmer to have more freedom (but also responsibility) than other members of the control-driven family of coordination languages. For instance, ConCoord [19] enforces a hierarchical evolution of coordinator and computation processes, where each coordinator process is responsible for monitoring the activities of a group of computation processes forming a domain. Interprocess communication is distinguished between being either interdomain or intradomain and the ports of a coordinator are only used to define bindings between computation processes belonging to different domains. MANIFOLD on the other hand, does not impose any such specific hierarchy; instead, the programmer

has the ability (but also the responsibility) to define different forms of process hierarchies by means of restricting observability of raised events to specific groups of processes.

In contrast to a rather extensive repertoire of coordination constructs, MANIFOLD does not support ordinary computational entities such as data structures (with the exception of tuples), variables, conditional or loop statements, etc. (although for programming convenience, they actually do exist in the language as syntactic sugar).

We are currently evaluating further the model for a number of areas related to coordination programming. In Ref. [30], we extend the model to handle real-time constraints. The derived model is used along with some of the techniques described in this paper in the development of a distributed multimedia environment and also in the specification of software architectures [2]. In Ref. [31], we show how MANIFOLD's underlying computational model (namely IWIM) can be applied to the Shared Dataspace family of coordination languages. Furthermore, in Ref. [32], we explore the possibility of applying MANIFOLD to the fields of groupware and workflow management.

Finally, one notices that there are some interesting similarities between IWIM and the family of Module Interconnection Languages [33]. However, MILs are less liberal than IWIM and MANIFOLD (or most other coordination languages for that matter). They require considerable prior agreement between the developers of different modules - e.g., all modules may have to be written in the same programming language or module interfaces should describe the other modules with which they interact. Furthermore, the interrelationships formed in MILs between components are essentially static, whereas in IWIM and MANIFOLD the interconnections change dynamically under program control. Nevertheless, an interesting line of research would be to examine whether IWIM can be used as the basis for a formal analysis of MILs' functionality.

Acknowledgements

This work has been partially supported by the INCO-DC KIT (Keep-in-Touch) program 962144 'Developing Software Engineering Environments for Distributed Information Systems' financed by the Commission of the European Union, and by project ERIS financed by the University of Cyprus.

References

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

[2] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, G. Zelesnik, Abstractions for software architecture and tools to support them, IEEE Trans. Software Eng. 21 (4) (1995) 314–335.

[3] S. Ahuja, N. Carriero, D. Gelernter, Linda and friends, IEEE Comput. 19 (8) (1986) 26–34.

[4] A. Brogi, P. Ciancarini, The concurrent language shared-prolog, ACM Trans. Programming Languages Syst. 13 (1) (1991) 99–123.

[5] N. Carriero, D. Gelernter, T. Mattson, A. Sherman, The Linda alternative to message passing systems, Parallel Comput. 20 (1994) 633–655.

[6] S. Frølund, G. Agha, A language framework for multi-object coordination, 7th European Conference on Object-Oriented Programming (ECOOP'93), Kaiserslautern, Germany, 26–30 July, 1993, LNCS 707, Springer-Verlag, pp. 346–360.
[7] N.H. Minsky, 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.
[8] J.-P. Banatre, D. Le Metayer, The GAMMA model and its discipline of programming, Sci. Comput. Programming 15 (1990) 55–70.
[9] J.-M. Andreoli, R. Pareschi, Linear objects: logical processes with built-in inheritance, New Generation Comput. 9 (3–4) (1991) 445–473.
[10] J.-M. Andreoli, P. Ciancarini, R. Pareschi, Interaction abstract machines, Trends in Object Based Concurrent Systems, MIT Press, 1993, pp. 257–280.
[11] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison Wesley Publs., 1989.
[12] J. Darlington, Y. Guo, H.W. To, J. Yang, Functional skeletons for parallel coordination, EUROPAR'95, Stockholm, Sweden, 23–31 Aug., 1995, LNCS 966, Springer-Verlag, pp. 55–68.
[13] D.B. Skillicorn, Towards a higher level of abstraction in parallel programming, Programming Models for Massively Parallel Computers (MPPM'95), IEEE Press, Berlin, Germany, 9–12 Oct., 1995, pp. 78–85.
[14] I. Foster, R. Olson, S. Tuecke, Productive parallel programming: the PCN approach, Scientific Programming 1 (1) (1992) 51–66.
[15] K.J. Goldman, B. Swaminathan, T.P. McCartney, M.D. Anderson, R. Sethuraman, The programmer's playground: I/O abstractions for user-configurable distributed applications, IEEE Trans. Software Eng. 21 (9) (1995) 735–746.
[16] F. Arbab, The IWIM model for coordination of concurrent 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.
[17] F. Arbab, I. Herman, P. Spilling, An overview of manifold and its implementation, Concurrency: Practice and Experience 5 (1) (1993) 23–70.
[18] MANIFOLD home page, URL: http://www.cwi.nl/ ~ farhad/manifold.html.
[19] A.A. Holzbacher, A software environment for concurrent coordinated programming, First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, 15–17 April, 1996, LNCS 1061, Springer-Verlag, pp. 249–266.
[20] J.A. Bergstra, P. Klint, The TOOLBUS coordination architecture, First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, 15–17 April, 1996, LNCS 1061, Springer-Verlag, pp. 75–88.
[21] J. Kramer, J. Magee, A. Finkelstein, A constructive approach to the design of distributed systems, Tenth International Conference on Distributed Computing Systems (ICDCS'90), Paris, France, 26 May–1 June, 1990, IEEE Press, pp. 580–587.
[22] M.R. Barbacci, C.B. Weinstock, D.L. Doubleday, M.J. Gardner, R.W. Lichota, Durra: a structure description language for developing distributed applications, Software Eng. J., IEE, March 1996, pp. 83–94.
[23] J. Magee, N. Dulay, J. Kramer, Structured parallel and distributed programs, Software Engineering Journal IEE, March 1996, pp. 73–82.
[24] J.M. Purtilo, The POLYLITH software bus, ACM Trans. Programming Languages Syst. 16 (1) (1994) 151–174.
[25] D.C. Luckham, Specification and analysis of system architecture using rapide, IEEE Trans. Software Eng. 21 (4) (1995) 336–355.
[26] G.A. Papadopoulos, F. Arbab, Coordination models and languages, Advances in Computers 46, Academic Press, 1998 (to appear).
[27] F. Arbab, C.L. Blom, F.J. Burger, 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.
[28] P. Nixon, C. Birkinshaw, P. Croll, D. Marriott, Rapid prototyping of parallel fault tolerant systems, 2nd Euromicro Workshop on Parallel and Distributing Processing (PDP'94), Malaga, Spain, Jan., 1994, IEEE Press, pp. 202–210.

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

[30] G.A. Papadopoulos, F. Arbab, Coordination of systems with real-time properties in manifold, Twentieth Annual International Computer Software and Applications Conference (COMPSAC'96), Seoul, Korea, 19–23 Aug., 1996, IEEE Press, pp. 50–55.

[31] G.A. Papadopoulos, F. Arbab, Coordination of distributed activities in the IWIM model, Int. J. High Speed Comput., World Scientific, 1997 9 (2) 127–160.

[32] G.A. Papadopoulos, F. Arbab, Control-based coordination of human and other activities in cooperative information systems, Second International Conference on Coordination Models and Languages, 1–3 Sept., 1997, Berlin, Germany, LNCS, Springer-Verlag, pp. 422–425.

[33] M.D. Rice, S.B. Seidman, A formal model for module interconnection languages, IEEE Trans. Software Eng. 20 (1994) 88–101.