
ELSEVIER

Parallel Computing 24 (1998) 989–1004

PARALLEL COMPUTING

# Coordination languages for parallel programming

F. Arbab [a,*], P. Ciancarini [b], C. Hankin [c]

[a] *CWI, Software Engineering Department, Kruislaan 413, 1098 SJ Amsterdam, Netherlands*

[b] *Dipartimento di Scienze dell'Informazione, Università di Bologna, Mura Anteo Zamboni 7, I-40127 Bologna, Italy*

[c] *Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK*

**Abstract**

A number of interesting models have been proposed and used to support coordination languages and systems. In this introductory paper, we first present a number of important concepts that form a context for classification and comparison of various coordination models and languages, and their applications. Next, we review three models and their associated languages, representing three different approaches to coordination. We illustrate the application of each model and language by using it to solve the classical dining philosophers problem. This paper ends with an overview of the rest of the papers that appear in this special issue. 

*Keywords:* Coordination languages; Parallel programming; Linda; Gamma; Manifold; Models of communication; Philosophers problem; Overview

## 1. Introduction

Humans often collaborate to achieve some shared objective. In such situations it is quite usual for one of the collaborators to be appointed or to emerge as a leader. One important role of the leader is to coordinate the activities of the other collaborators to ensure that the objective is achieved efficiently, possibly with minimum effort. Increasingly, we see an analogous situation in computing. The emergence of high bandwidth

* Corresponding author.

0167-8191/98/$ - see front matter 
PII: S0167-8191(98)00039-8

network technology, and the trend toward reusing whole applications as components of larger software configurations have fuelled the development of distributed computing and concurrent programming.

Coordination languages [1] are a new class of parallel programming languages which offer a solution to the problem of managing the interaction among concurrent programs. They offer language support for composing and controlling software architectures made of parallel or distributed components, and as such, can be thought of as the linguistic counterpart of software libraries and platforms, like MPI [2] or PVM [3], which offer extra-linguistic support for parallel programming.

A number of interesting models have been proposed and used to support coordination languages and systems. Examples include 'generative tuple spaces' as in Linda [4], various forms of 'parallel multiset rewriting' or 'chemical reactions' as in Gamma [5], and models with explicit support for coordinators [6]. A significant number of these models and languages are based on a few common notions, such as pattern-based, associative communication [7], to complement the name-oriented, data-based communication of traditional languages for parallel programming.

Coordination languages have been applied to the parallelization of computation intensive sequential programs in the fields of: simulation of fluid dynamics systems, matching of DNA strings, molecular synthesis, parallel and distributed simulation, monitoring of medical data, computer graphics, analysis of financial data integrated into decision support systems, and game playing (chess). See Refs. [8–10] for some concrete examples.

Carriero and Gelernter [1] coined the term *Coordination* in the following slogan:

$$\text{Programming} = \text{Computation} + \text{Coordination}$$

The authors formulated this equation discussing the coordination language Linda (see below). The intent is that there should be a clear separation between the *components of the computation* and their *interaction* in the overall program or system. On the one hand, this separation facilitates the reuse of components; on the other hand, the same patterns of interaction occur in many different problems - so it might be possible to reuse the coordination component as well!

In the rest of this article we identify some basic principles of coordination languages, describe some examples, and introduce the rest of the papers in this special issue.

## 2. Basic principles

In defining coordination languages, there are a number of issues which must be addressed:

1. What is to be accomplished by coordination?
2. What is being coordinated?
3. What are the media for coordination?
4. What are the protocols and rules used for coordination?

Before expanding each of these, it is important to make some general observations.

- Coordination languages usually are not general purpose programming languages; rather, they are often defined as language extensions or scripting languages and they are exclusively concerned with coordination issues.
- Shared memory multi-processor machines (e.g., SUN and SGI) and cluster architectures (e.g., IBM SP) of today offer practical and cost effective parallelism to applications that can take advantage of coarser-grain concurrency. Coordination models and languages are equally pertinent for such parallel application as they are for distributed computing.
- Coordination languages are most relevant in the context of open systems, where the coordinated entities are not fixed at the outset; here they have much in common with object-based approaches. In order to operate in an open system, entities must be *encapsulated* (their implementation details should be hidden from other entities) and they should persist beyond a single transaction (some authors describe such entities as *reactive* [8]). In the past, such considerations have led to the development of object-based modeling techniques; the design of coordination languages should also address these issues.
- The natural description of the activities and the history of the computation carried out in a certain class of applications tends to center around a substantial shared body of data; we call them *data-oriented* applications. Such an application is essentially concerned with what happens to *the data*. Examples of data-oriented applications include database applications and transaction systems such as banking and airline reservation systems. There is another class of applications, called *process-oriented* or *control-oriented* where it is unnatural to view their activities and their history as centered around a shared body of data. Indeed, often, the very notion of *the data*, as such, simply does not exist in these applications. For instance, it is unnatural for compiler writers to consider a central body of data in a compiler, and describe the compilation process in terms of a collection of activities that transform this data; it is more natural to view a compiler as a collection of activities that genuinely consume their input data, and subsequently produce, remember, and transform 'new data' that they generate by themselves.

### 2.1. Coordination purpose

The use of a coordination model or language in an application influences its design and its structure in various significant ways. From a software engineering perspective, depending on the nature and the life cycle of an application, some of these models and their influences may be more or less congruent with what is intended to be accomplished through coordination. In this respect, we distinguish three issues.

1. Coordination languages can be used to separate the control issues from the computation concerns in the design and development of parallel programs derived from high-level specifications [11]. As a consequence, the correctness concerns (i.e., the computation) and efficiency issues (i.e., the coordination) can be dealt with separately [12]. The use of coordination for this purpose shares similar concerns with

formal specifications, automatic derivation of programs from higher-level specifications, and functional programming.

2. Coordination languages can be used as a means to separate the 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. The applications most directly amenable to this approach are the data-oriented applications. The notion of shared data spaces used in many coordination models and languages has obvious conceptual similarities with relational databases and with blackboard systems in AI [13]. This makes them a particularly suitable match for many data-oriented applications.
3. Coordination languages can be used as a means to describe the communication protocols necessary for the cooperation of the active entities in a parallel/distributed application, introducing independent coordination modules. The very large class of applications most directly amenable to this approach are the control-oriented applications. Most coordination models and languages targeted for this purpose use a message passing paradigm as their base and modify it by introducing such additional notions as synchronizers, contracts, constraints, and events.

### 2.2. Coordinated entities

The coordinated entities are usually active agents or processes. Coordination of agents should not require reprogramming of the agents; the coordination mechanism is a wrapper around the existing, independent agents. The agents themselves may have been programmed in a variety of different programming languages.

### 2.3. Coordination media

In many coordination languages, coordination is accomplished via a shared data space. In such models, communication is *generative*: agents communicate by 'generating' data in the shared space. This data is then available to any other agent that has access to the space - this contrasts with the message-passing paradigm where communication is usually a private act between the participating agents. In a heterogeneous system, in which the agents are written in different languages, the data must be stored in a common format.

Alternatively, some coordination models and languages use point-to-point communication channels that are constructed and dismantled by coordination protocols. In these models coordination is event-driven rather than data-driven.

### 2.4. Coordination rules

The Linda proposal identifies a set of coordination primitives which may be used to access a shared data space - the primitives are normally implemented as library routines

which are called from some host language such as C or Prolog. In contrast to Linda, many of the recent proposals have been for rule-based languages; one consequence of this shift to a more declarative view of coordination is increased reasoning power. In either case the coordination 'rules' provide a level of abstraction which hides much of the complexity of coordination from the programmer.

## 3. Examples

### 3.1. Linda

Linda must be combined with a sequential programming language (called the *host language*) to offer a complete language for parallel programming.

The central feature of Linda [14] is the *tuple space*. This is the coordination medium - it is a shared data structure which contains tuples (records). Tuples may be passive (data) or active (processes). A tuple is active if one or more of its fields are function calls. Tuples are created and manipulated by processes using a small set of operations. Only passive tuples can be accessed; tuple selection involves pattern matching.

Tuples are finite sequences of fields; the number of fields is the *arity* of the tuple. Every field has a value and a type drawn from the host language. The type of a tuple is the cross product of the types of its fields.

**Example**. In C-Linda, `("array",1,3)` is a tuple of arity 3. The first field has type `string`, the second and third fields have type `int`. The type of the tuple is `string` × `int` × `int`.

The tuple space is a multiset of tuples, i.e., identical tuples may exist in the tuple space. Processes communicate by inserting, removing and examining tuples in the tuple space. Thus the tuple space is shared: all processes have access to all tuples in it. Access is associative, i.e., processes use pattern matching to access tuples. Pattern matching is based on the concept of an *anti-tuple*, that is similar to a tuple, except that some fields can be typed variables; these are prefixed by a '?' symbol. We say that a tuple $t$ and an anti-tuple $a$ match if and only if:

1. both $t$ and $a$ have the same arity;
2. values in corresponding fields are identical;
3. a variable in $a$ and some corresponding value in $t$ have the same type.

The result of a successful matching operation is that variables in anti-tuple $a$ obtain the values contained in the corresponding fields of tuple $t$.

**Example.** The anti-tuple ("array",?x,?y) matches the tuple `("array",1,3)`; the anti-tuple ("array",?x,?x) does not match the tuple `("array",1,3)`.

Several Linda implementations also allow variables in tuples (i.e., in the tuple space), but they do not match a corresponding variable in an anti-tuple. Thus variables in tuples play the role of wild cards in matching any value in an anti-tuple, and no side effect is intended.

The operations on tuples are:

`in(t)` : looks for a tuple matching `t`; if found then the tuple is deleted, otherwise the process waits until matching succeeds.

`out(t)` : creates a new passive tuple whose contents are specified by `t`.

`eval(t)` : creates an active tuple whose contents are specified by `t`. An active tuple must have at least one field which is a function to be computed.

Both of the last two operations are guaranteed to succeed; the issuing process continues immediately. The selected operations are needed for the example; the full set of operations includes a non-destructive read and predicates for testing the state.

To be more concrete we will illustrate an example in C-Linda. We give a solution to the Dining Philosopher's problem - a classic concurrency problem. We will describe a version of the problem involving five philosophers. The philosophers sit around a circular table with a bowl of spaghetti in the middle and five forks. Each philosopher alternately thinks and then eats; in order to eat spaghetti, the philosopher requires two forks. The situation is modeled by the C-Linda program in Fig. 1.

The philosophers are numbered from 0 to 4, and so are the forks. The program (`real_main()`) generates five passive tuples representing forks and five active tuples representing the philosophers. The program also generates four meal tickets; a philosopher must have a meal ticket before attempting to eat - the fact that there are only four tickets means that at least one philosopher has access to two forks and the system does

```
#define TRUE 1

philosopher(int i)
{
   while(TRUE) {
      think();
      in("meal ticket"); in("fork", i); in("fork", (i+1)%5);
      eat();
      out("fork", i); out("fork", (i+1)%5); out("meal ticket");
   } }

real main()
{
   int i;
   for (i=0, i<5, i++){
      out("fork", i);
      eval(philosopher(i));
      if (i<4) out("meal ticket");
   }
}
```

Fig. 1. The dining philosophers in C-Linda.

not deadlock. When a philosopher is ready to eat he/she must pick up the fork with the same number and an adjacent fork (% is the modulus operation in C).

### 3.2. Combining Linda with another language

Linda has been combined with several programming languages, like for instance C, FORTRAN, Prolog, and Java. When designing a combination, the language designer has to specify the following issues.

- The type system for data values, and the matching rules between tuples and anti-tuples. Linda has neither a type system nor a set of basic data values. Tuples are defined as ordered sequences of data values inherited from the host language. Not all data types of the host language are easily embedded in Linda. For instance, in C-Linda tuples cannot include pointer values because it would be meaningless to pass such references from one process to another. This issue is especially important because each data type can be analyzed and implemented using specific strategies which optimize the run-time behavior of the Linda program.
- The control constructs allowed to combine Linda tuple operators. Control constructs are inherited from the host language, and not all constructs are compatible with Linda operators on tuples. For instance, it is difficult to combine the tuple operators with backtracking, as in Prolog.
- The semantics and possible constraints on active tuples, i.e., on `eval`. For instance, in C-Linda all fields of an active tuple can be function calls. However, in some implementations only one process (namely only one `eval`) per processor is allowed.
- The closures for active tuples. When an active tuple is put in the tuple space, it must be specified which is the environment assumed for variables in the code that it executes. For instance, in C-Linda under Unix the closure is empty.
- Syntax and semantics of operations for multiple tuple spaces. In the original Linda definition only one tuple space is allowed. A natural extension consists of relaxing this constraint, defining a language based on Multiple Tuple Spaces [15].

Linda has been implemented on all the major parallel architectures, like SP/2 and Cray T3, and on several network operating systems, like SunOS, Solaris, Linux, Silicon Graphics, and Windows NT. The commercial implementations by SCA usually offer Linda combined with C or FORTRAN.

### 3.3. Gamma

The coordination medium in Gamma [16] is a multiset - a set-like collection which may contain many copies of the same element. In the basic model, the multiset is untyped and so it is rather similar to the Linda tuple space but, in contrast to Linda, all of the elements of the multiset are passive. Simple agents are represented as pairs consisting of a reaction condition and an action, written:

$$\text{action} \Leftarrow \text{reaction}$$

An action is a rewrite rule:

$$\text{lhs} \rightarrow \text{rhs}$$

The action selects some elements (which match the left hand side of the rule and satisfy the reaction condition) from the multiset and rewrites them according to the rule (replacing them by the elements listed on the right hand side of the rule). Most papers on Gamma assume a simple functional language for the actions [5]; however, the Gammalög language [17] is an instance of Gamma built on the logic programming language Gödel.

Gamma has been used in a number of application areas. The original papers contain many, small programming examples. More substantial applications include image processing and biological modeling. Discussion of these applications and citations to the relevant literature may be found in Ref. [8].

The philosopher function might be expressed by the following two rules in Gamma:

```
("fork", i), ("fork", j) → ("eat", i) ⇐ j = (i+1)%5
("eat", i) → ("fork", i), ("fork", (i+1)%5) ⇐ true
```

Since the selection of elements is an atomic action, the "meal ticket" is not required in this solution.

As another example, consider the problem of sorting a multiset of values. The parallel sort program in Gamma is defined by:

```
("index", i), x → (i,x), ("index", i+1) ⇐ true
(i, x), (j, y) → (j, x), (i, y) ⇐ (x > y) and (i < j)
```

We have assumed that the input to the program is a (multi)set of values and a distinguished element ("index", 0). We further require that there is an ordering relation, $\geq$, on the values. The first rule associates an index with each value; the initial assignment of indices is non-deterministic. The second rule reorders indices to ensure that larger values have higher indices. Notice that the semantics of Gamma allow for parallel execution of the two rules as well as parallel applications of the second rule: indices can be reordered as soon as they are generated. The first rule describes an essentially sequential (but non-deterministic) process - there is only one index element. On termination, the multiset contains a set of (index, value) pairs where the index indicates the position of the value in the sorted order of the input values; there will also be an element ("index", n) where n is the number of values that were sorted.

There have been several proposals for extensions to this basic model. Higher-order extensions [18] allow active configurations to be stored in the multiset - this extended framework has been used to study the properties of other coordination languages [19]. LeMétayer has recently proposed a typed variant of Gamma, called Structured Gamma [20], which has been used in the study of software architectures.

There have been a number of prototype implementations of Gamma: on the Connection Machine, Intel iPSC2, MasPar, Sequent and specialized parallel hardware [21]. Detailed bibliographic references for these implementations may be found in the references of Ref. [22].

### 3.4. MANIFOLD

MANIFOLD is a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes [23]. The processes that comprise an application are either computation or coordinator processes. Computation processes can be written in any conventional programming language. Coordinator processes are clearly distinguished from the others in that they are written in the MANIFOLD language. The purpose of a coordinator process is to establish and manage the communications among other (computation or coordinator) processes. MANIFOLD is a control-driven or process-oriented coordination language based on the IWIM model of communication [24].

IWIM stands for *Idealized Worker Idealized Manager* and is a generic, abstract model of communication that supports the separation of responsibilities and encourages a weak dependence of workers (processes) on their environment. Two major concepts in IWIM are separation of concerns and anonymous communication. Separation of concerns means that computation concerns are isolated from the communication and cooperation concerns into, respectively, worker and manager (or coordinator) modules. Anonymous communication means that the parties (i.e., modules or processes) engaged in communication with each other need not know each other. IWIM-sanctioned communication is either through broadcast of events, or through point-to-point channel connections that, generally, are established between two communicating processes (who do not know each other's identity) by a third party coordinator process.

MANIFOLD is a strongly-typed, block-structured, event driven language, meant for writing coordinator program modules. As modules written in MANIFOLD represent the idealized managers of the IWIM model, strictly speaking, there is no need for the constructs and the entities that are common in conventional programming languages; thus, semantically, there is no need for integers, floats, strings, arithmetic expressions, conditional statements, loops, etc. [1] The only entities that MANIFOLD recognizes are processes, ports, events, and streams, and the only control structure that exists in MANIFOLD is an event-driven state transition mechanism. Programming in MANIFOLD is a game of dynamically creating (coordinator and/or worker) process instances and dynamically (re)connecting the ports of some of these processes via streams, in reaction to observed event occurrences. 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 in most other coordination languages and models.

[1] For convenience, however, some of these constructs, syntactically, do exist in the MANIFOLD language. Currently, only the front-end of the MANIFOLD language compiler knows about such 'syntactic sugar' and translates them into processes, state transitions, etc., so that as far as the run-time system (or even the code generator of the MANIFOLD compiler) is concerned, these familiar constructs 'do not exist' in MANIFOLD.

MANIFOLD encourages a discipline for the design of concurrent software that results in two separate sets of modules: pure coordination, and pure computation. This separation disentangles the semantics of computation modules from the semantics of the coordination protocols. The coordination modules construct and maintain a dynamic data-flow graph where each node is a process. These modules do no computation, but only change the connections among various processes in the application as prescribed, which changes only the topology of the graph. The computation modules, on the other hand, cannot possibly change the topology of this graph, making both sets of modules easier to verify and more reusable. The concept of reusable pure coordination modules in MANIFOLD is demonstrated, e.g., by using (the object code of) the same MANIFOLD coordinator program that was developed for a parallel/distributed bucket sort algorithm, to perform function evaluation and numerical optimization using domain decomposition [25,26].

The MANIFOLD system consists of a compiler, a run-time system library, a number of utility programs, libraries of built-in and pre-defined processes, a link file generator called MLINK, and a run-time configurator called CONFIG. The system has been ported to several different platforms (e.g., SGI 5.3, SGI 6.3, Solaris 5.2, Linux, and IBM SP/1 and SP/2).

*3.4.1. Processes*

In MANIFOLD, the atomic workers of the IWIM model are called atomic processes. Any operating system-level process can be used as an atomic process in MANIFOLD. Furthermore, a regular C function running as an independent thread can be used as an atomic process too. Atomic processes can only produce and consume units through their ports, broadcast and receive events, and compute. In this way, the desired separation of computation and coordination is achieved.

Coordination processes are written in the MANIFOLD language and are called manifolds. A manifold definition defines a process type and consists of a header and a body. The header of a manifold gives its name, the number and types of its parameters, and the names of its input and output ports. The body of a manifold definition is a block. A block consists of a finite number of states. Each state has a label and a body. The label of a state defines the condition under which a transition to that state is possible. It is an expression that can match observed event occurrences in the event memory of the manifold instance. The body of a simple state defines the set of actions that are to be performed upon transition to that state. The body of a compound state is either a (nested) block, or a call to a parameterized subprogram known as a *manner* in MANIFOLD. A manner consists of a header and a body. As for the subprograms in other languages, the header of a manner essentially defines its name and the types and the number of its parameters. A manner is either atomic or regular. The body of a regular manner is a block. The body of an atomic manner is a C function that can interface with the MANIFOLD world through an interface library.

*3.4.2. Ports*

A *port* is a regulated opening at the boundary of a process, through which the information produced and/or consumed by the process is exchanged with other processes. Regulated means that the information can flow in only one direction through a

port: it either flows into or out of the process. The information exchanged between a process and other processes through its ports is quantized in discrete bundles called units. A *unit* is a packet containing an arbitrary number of bits that are produced, transferred, and consumed in an integral fashion; i.e., there are no partial units.

Ports through which units flow into a process are called the *input* ports of the process. Similarly, ports through which units flow out of a process are called the *output* ports of the process.

*3.4.3. Streams*

All communication in MANIFOLD is asynchronous. In MANIFOLD, the asynchronous IWIM channels are called streams. A stream is a communication link that transports units. A stream represents a reliable and directed flow of information from its *source* to its *sink*. As in the IWIM model, the constructor of a stream between two processes is, in general, a third process. Once a stream is established between (a port of) a producer process and (a port of) a consumer process, it operates autonomously and transfers the units from its source to its sink. When the process at the sink of a stream requires a unit through its port connected to that stream, it is suspended only if no units are available in any of the streams connected to the arrival side of that port. The suspended process resumes as soon as the next unit becomes available for its consumption. The source of a stream is never suspended because the infinite buffer capacity of a stream is never filled.

*3.4.4. Events and state transitions*

In MANIFOLD, once an event is *raised* by a process, the latter continues, while the event occurrence propagates through the environment independently. Any receiver process that is interested in such an event occurrence will automatically receive it in its *event memory*. An observer process can react to an event occurrence in its event memory at its own leisure. In reaction to such an event occurrence, a manifold instance can make a transition from one labeled state to another.

The only control structure in the MANIFOLD language is an event-driven state transition mechanism. Upon transition to a state, the primitive actions specified in its body are performed atomically in some non-deterministic order. Then, the state becomes *pre-emptable*: if the conditions for transition to another state are satisfied, the current state is pre-empted. Pre-empting a state can dismantle the streams (depending on their types) that were connected upon transition to that state. The most important primitive actions in a simple state body are (i) creating and activating processes, (ii) generating event occurrences, and (iii) connecting streams to the ports of various processes.

*3.4.4.1. Example.* Our implementation of the dining philosophers problem in MANIFOLD models philosophers and forks as separate process types: manifolds `Philosopher` and `Fork`, as shown in the listing. The `Main` program (contained in a separate source file and not shown here) simply creates and activates five instances of `Philosopher` and five instances of `Fork`, and arranges them in a circular configuration around the virtual table. `Main` accomplishes these introductions by making each instance of philosopher and fork sensitive to the events raised by the other processes it must know.

```
1  #define WAIT (preemptall, terminated(self))
2
3  event request, done.
4  manner Eat(process, process, process) import.
5  manner Think(process) import.
6  manner GetTicket() import.
7  manner ReturnTicket() import.
8
9  export Fork()
10 {
11   begin: while true do {
12            begin: WAIT.
13
14            request.*phil & *ready.*phil: {
15              save *.
16              begin: (raise(ready), WAIT).
17              done.phil:.
18              }.
19          }.
20 }
21
22 export Philosopher()
23 {
24   event ready.
25
26   begin: while true do {
27            begin: Think(self);
28                   GetTicket();
29                   (raise(request, ready), WAIT).
30
31            ready.*lfork & ready.*rfork: Eat(self, lfork, rfork).
32
33            end: raise(done);
34                 ReturnTicket().
35          }.
36 }
```

Line 1 in the listing defines `WAIT` as a preprocessor macro. What `WAIT` expands into is in fact a common programming idiom that hangs the executing process, waiting for an event from any of its known event sources to cause a state transition.

Line 3 defines `request` and `done` as events in the global scope of this source file. Any module within this source file that uses either of these identifiers without redefining it, refers to the corresponding event defined on this line.

Line 4 declares the prototype of a subprogram (manner) called `Eat` that takes three process-type parameters. The keyword `import` states that the body of this subprogram is defined in another source file. In reality, this subprogram may be a piece of MANIFOLD code in the other source file, or it may be, e.g., a C function. We do not

care about the details of Eat: whenever a philosopher manages to obtain the two forks it requires to eat, it calls Eat to engage in 'eating' and passes its own identity plus the identities of its two forks as its parameters (line 31). Similarly, line 5 defines Think as another imported subprogram, which is used by a philosopher to do its 'thinking' (line 27).

Lines 6 and 7 declare two other imported manners that together implement a 'dining ticket' mechanism used to prevent deadlocks (lines 28 and 34). These manners can easily be written in MANIFOLD, but we skip their detail here.

The manifolds Fork and Philosopher are our main interest here. An instance of Fork is sensitive to the two philosophers who can potentially use it. Upon activation, a Fork instance enters an infinite loop (lines 11–19) of waiting for a pair of event occurrences (line 12) and reacting to them (lines 14–18).

The only way [2] an instance of Fork can make a transition out of its wait state (line 12) is if it observes two event occurrences from the same process, one of which must be an occurrence of the event request. The request event is defined in this source file and because it does not have the extern attribute, this event is not known in any other source file in any application. Within this source file, request can be raised only by instances of Philosopher (line 29). Thus, the source of the request event occurrence (on line 14) can only be an instance of Philosopher. The identity of this Philosopher instance will be bound to the identifier phil, due to the *phil construct in the label of this state. This binding restricts the event occurrences that can match the rest of the label: *ready can match any event raised by the same source, phil, that has also raised request. The only thing that can possibly match *ready on line 14 is an occurrence of ready raised by an instance of Philosopher on line 29. Note that there can be two pairs of event occurrences raised by different instances of Philosopher, in which case, a Fork instance picks one pair non-deterministically.

After transition, a Fork instance raises the same ready event it has received, and waits for another event (line 16). Although the ready event is broadcast by Fork, because each philosopher has its own private ready event (see below), no one other than the philosopher who raised it and caused a transition in Fork to line 14 can react to it. Once a Fork instance is in this wait state (line 16), no event occurrences other than an occurrence of the event done from phil (i.e., the same instance of Philosopher that caused the transition to line 14) can cause it to make a state transition; this is due to the save statement on line 15.

To an instance of Philosopher, receiving its own private ready event means that one of the forks it is potentially entitled to use is now exclusively at its disposal. In return, the Philosopher instance must raise the event done to inform the forks it has used to 'eat' that it is done with them. Thus, a committed Fork ends its wait in each iteration (line 17) when it receives an occurrence of done raised by the same philosopher it had committed itself to on line 14. At this point, the next iteration starts.

[2] We ignore here the predefined events terminate and abort to which all process instances respond by terminating.

Because the event `ready` is declared inside the body of `Philosopher` (line 24), every instance of `Philosopher` will have its own unique, private `ready` event. Analogous to `Fork`, an instance of `Philosopher` enters an infinite loop upon its activation (lines 26–35). In each iteration of this loop, a `Philosopher` instance first does its thinking (line 27), then waits to obtain a dining ticket (line 28), and finally, declares its intention to eat by raising the events `request` and its private `ready`, and goes into a wait state. If and when two instances of `Fork` declare their exclusive commitment to this `Philosopher` instance, `it` will have two occurrences of its own `ready` event raised by them in its event memory. This allows it to make a transition to the state on line 31, where the identities of the two forks will be bound to `lfork` and `rfork`. Now the `Philosopher` instance eats, and once done, it raises the event `done` to release its committed forks (line 33), returns its dining ticket (line 34), and goes on to its next iteration.

There will be 11 processes running in this application: an instance of `Main`, 5 instances of `Philosopher`, and 5 instances of `Fork`. Each of these 11 processes will actually be a light-weight process (preemptively scheduled thread) in some task instance (heavy-weight process). The actual number of task instances that house these processes can range from 1 to 11, and they can run on the same actual (single or multi-processor) host, or on several (homogeneous or heterogeneous) such hosts over a network. None of this detail is relevant at the level of the source code, and the same compiler produced object code can be linked with different specifications in the MLINK and CONFIG input to tailor the desired run-time configuration.

## 4. About this special issue

The rest of this issue consists of six papers. The papers were submitted in response to an open call for submissions. They use and extend the three coordination languages presented in Section 3. A major criterion in selecting these papers was that they all demonstrate the role of coordination languages in parallel applications.

- *An implementation of Linda for a NUMA machine* by N. Carriero: The paper presents a new software architecture for Linda's run-time support. It also reports on empirical results from implementations on the Cray T3D/E. The results indicate that the performance is competitive with low-level coordination using message passing (MPI and PVM).
- *The formal derivation of parallel triangular system solvers using a coordination-based design method* by M.R.V. Chaudron and A.C.N. van Duin: The paper uses Chaudron and De Jong's schedules and a suitable notion of refinement to develop a parallel solver for systems of linear equations. The initial specification is a Gamma program. Schedules are used to organize the computation into components which correspond to different levels of the BLAS hierarchy. The initial specification and refinement steps are formally verified.
- *Generative coordination environments supporting parallel discrete event simulation* by L. Donatiello and A. Fabbri: The paper identifies a number of difficult problems posed by parallel discrete event simulation (partitioning, message flow control, global

virtual time computation, etc.). The authors propose a new methodology for parallel simulation which they call Active-Events. They describe an implementation based on Linda.

- *Using coordination to parallelize sparse-grid methods for 3-D CFD Problems* by C.T.H. Everaars and B. Koren: This paper describes the use of MANIFOLD to parallelize a sequential Fortran CFD code. The code is restructured into a master/slave architecture. The latter is implemented using MANIFOLD for the coordination—the Fortran subroutines are enclosed in C wrapper functions which are called from MANIFOLD. The paper includes empirical results which demonstrate a nearly linear speed-up.
- *Behavior specification of parallel active objects* by T. Holvoet and T. Kielmann: The paper presents Objective Linda, a new model for parallel object-oriented programming. The authors use a Petri Net formalism for specifying behaviors. Type checking amounts to testing for liveness of certain transitions. They apply their formalism to the study of a master/slave architecture; generic master and slave agents are designed and type checking is used to verify correct interaction.
- *Distributed and parallel systems engineering in MANIFOLD* by G.A. Papadopoulos: This paper investigates software engineering aspects of coordination programming using the MANIFOLD language. The paper catalogues a number of techniques that the author has developed.

## References

[1] N. Carriero, D. Gelernter, Coordination languages and their significance, Commun. ACM 35 (2) (1992) 97–107.
[2] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, MPI: The complete reference, MIT Press, 1995.
[3] G. Geist, V. Sunderam, Network-based concurrent computing on the PVM system, Concurrency—Practice and Experience 4 (4) (1992) 293–311.
[4] D. Gelernter, Generative communication in Linda, ACM Trans. Programming Languages Syst. 7 (1) (1985) 80–112.
[5] J.P. Banatre, D. LeMetayer, Programming by multiset transformation, Commun. ACM 36 (1) (1993) 98–111.
[6] F. Arbab, I. Herman, P. Spilling, An overview of Manifold and its implementation, Concurrency—Practice and Experience 5 (1) (1993) 23–70.
[7] J.M. Andreoli, P. Ciancarini, R. Pareschi, Interaction abstract machines, in: G. Agha, P. Wegner, A. Yonezawa (Eds.), Trends in Object-Based Concurrent Computing, MIT Press, Cambridge, MA, 1993, pp. 257–280.
[8] J.M. Andreoli, C. Hankin, D. LeMetayer (Eds.), Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996.
[9] P. Ciancarini, C. Hankin (Eds.), 1st Int. Conf. on Coordination Languages and Models (COORDINATION), Vol. 1061 of Lecture Notes in Computer Science, Cesena, Italy, Springer-Verlag, Berlin, April 1996.
[10] D. Garlan, D. LeMetayer (Eds.), 2nd Int. Conf. on Coordination Languages and Models (COORDINATION), Vol. 1282 of Lecture Notes in Computer Science, Berlin, Germany, Springer-Verlag, Berlin, September 1997.
[11] N. Brown, Correctness-preserving transformations for the design of parallel programs, in: P. Ciancarini, O. Nierstrasz, A. Yonezawa (Eds.), Object-Based Models and Languages for Concurrent Systems, Vol. 924 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1995, pp. 29–48.

[12] M. Chaudron, E. deJong, Towards a compositional method for coordinating Gamma programs, in: P. Ciancarini, C. Hankin (Eds.), Proc. 1st Int. Conf. on Coordination Models and Languages, Vol. 1061 of Lecture Notes in Computer Science, Cesena, Italy, April 1996, Springer-Verlag, Berlin, pp. 107–123.
[13] H. Nii, Blackboard systems, in: A. Barr, P. Cohen, E. Feigenbaum (Eds.), The Handbook of Artificial Intelligence, Vol. 4, Addison-Wesley, 1989, pp. 1–82.
[14] N. Carriero, D. Gelernter, T. Mattson, A. Sherman, The Linda alternative to message-passing systems, Parallel Comput. 20 (1994) 633–655.
[15] D. Gelernter, Multiple tuple spaces in Linda, in: E. Odijk, M. Rem, J. Syre (Eds.), Proc. Conf. on Parallel Architectures and Languages Europe (PARLE 89), Vol. 365 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1989, pp. 20–27.
[16] J.P. Banatre, D. LeMetayer, The Gamma model and its discipline of programming, Sci. Comput. Programming 15 (1990) 55–77.
[17] P. Ciancarini, D. Fogli, M. Gaspari, A logic language based on multiset rewriting, in: J.M. Andreoli, C. Hankin, D. LeMetayer (Eds.), Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996, pp. 323–348.
[18] D. LeMetayer, Higher order multiset programming, in: G. Blelloch et al. (Eds.), Specification of Parallel Algorithms (DIMACS Workshop), Vol. of DIMACS, May 1994, pp. 179–200.
[19] M. Bourgois, Specifying a reflective and distributed implementation of LO in higher order Gamma, in: J.M. Andreoli, C. Hankin, D. LeMetayer (Eds.), Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996, pp. 3–41.
[20] P. Fradet, D. LeMetayer, Type checking for a multiset rewriting language, in: M. Dam (Ed.), Analysis and Verification of Multiple-Agent Languages, Vol. 1192 of Lecture Notes in Computer Science, Stockholm, Springer-Verlag, Berlin, June 1996, pp. 126–140.
[21] J.P. Banatre, A. Coutant, D. LeMetayer, A parallel machine for multiset transformation and its programming style, Future Generation Comput. Syst. 4 (1988) 133–144.
[22] J.P. Banatre, D. LeMetayer, Gamma and the chemical reaction model: Ten years after, in: J.M. Andreoli, C.Hankin, D. LeMetayer (Eds.), Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996.
[23] F. Arbab, Coordination of massively concurrent activities. Technical Report CS-R9565, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, November 1995, available on-line http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z.
[24] F. Arbab, The IWIM model for coordination of concurrent activities, in: P. Ciancarini, C. Hankin (Eds.), Proc. 1st Int. Conf. on Coordination Models and Languages, Vol. 1061 of Lecture Notes in Computer Science, Cesena, Italy, April 1996, Springer-Verlag, Berlin, pp. 34–56.
[25] F. Arbab, C. Blom, F. Burger, C. Everaars, Reusable coordinator modules for massively concurrent applications, in: L. Bouge, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Proceedings of Euro-Par '96, Vol. 1123 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, August 1996. pp. 664–677.
[26] C. Everaars, F. Arbab, Coordination of distributed/parallel multiple-grid domain decomposition, in: A. Ferreira, J. Rolim, Y. Saad, T. Yang (Eds.), Proc. Irregular '96, Vol. 1117 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, August 1996, pp. 131–144.