COORDINATION MODELS AND LANGUAGES

GEORGE A. PAPADOPOULOS

Department of Computer Science
University of Cyprus
Nicosia, Cyprus

FARHAD ARBAB

Department of Software Engineering
CWI
Amsterdam, The Netherlands

Abstract

A new class of models, formalisms and mechanisms has recently evolved for describing concurrent and distributed computations based on the concept of "coordination". The purpose of a coordination model and associated language is to provide a means of integrating a number of possibly heterogeneous components together, by interfacing with each component in such a way that the collective set forms a single application that can execute on and take advantage of parallel and distributed systems. In this chapter we initially define and present in sufficient detail the fundamental concepts of what constitutes a coordination model or language. We then go on to classify these models and languages as either "data-driven" or "control-driven" (also called "process-" or "task-oriented"). In the process, the main existing coordination models and languages are described in sufficient detail to let the reader appreciate their features and put them into perspective with respect to each other. The chapter ends with a discussion comparing the various models and some conclusions.

1. Introduction

1.1 Background and Motivation

Massively parallel and distributed 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 the challenge.

Exploiting the full potential of massively 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. This has led to the design and implementation of a number of coordination models and their associated programming languages. Almost all of these models share the same intent, namely to provide a framework which enhances modularity, reuse of existing (sequential or even parallel) components, portability and language interoperability. However, they also differ in how they precisely define the notion of coordination, what exactly is being coordinated, how coordination is achieved, and what are the relevant metaphors that must be used.

The purpose of this survey chapter is twofold: (i) to present a thorough view of the contemporary state-of-the-art research in the area of coordination models and languages, and (ii) to provide enough information about their historical evolution, design, implementation and application to give the reader an appropriate understanding of this important and rapidly evolving research area in Computer Science to appreciate its potential.

In the process, we argue that these coordination models and languages can be classified into two main categories: those that are “data-driven”, where the evolution of computation is driven by the types and properties of data involved in the coordination activities, and those that are “control-driven” (or process-oriented) where changes in the coordination processes are triggered by events signifying (among other things) changes in the states of their coordinated processes.

1.2 Organisation of the Article

The rest of this chapter is organised as follows: Part 2 provides a historical perspective on coordination programming and explains how and why it has evolved into its current form. Part 3 describes in detail the most important coordination models and languages using the above mentioned classification. Part 4 presents a general comparison and classification of the models described in the previous part and part 5 ends the chapter with some conclusions.

2. From Multilingual and Heterogeneous Systems to Coordination Models

2.1 Need for Multilinguality and Heterogeneity

With the evolution of distributed and parallel systems, new programming paradigms were developed to make use of the available parallelism and the, often massive, number of processors comprising a system. These languages were able to exploit parallelism, perform communication but also be fault tolerant. They differed in the granularity or unit of parallelism they offered (e.g., sequential process, object, parallel statements, etc.) and the communication mechanism employed (e.g., message passing models such as rendezvous or remote procedure calls, data sharing models such as distributed data structures or shared variables, etc.). The increased availability of massively parallel and open distributed systems lead to the design and implementation of complex and large applications such as vehicle navigation, air traffic control, intelligent information retrieval and multimedia-based environments, to name but a few. Gradually, it became apparent that no unique programming language or machine architecture was able to deal in a satisfactory way with all the facets of developing a complex and multifunctional application. Furthermore, issues such as reusability, compositionality and extensibility became of paramount importance.

Thus, in order to deal with all these requirements of programming-in-the-large applications, the notions of multilinguality and heterogeneity came into play. *Multilingual* or *multiparadigm* programming ([73]) is able to support a number of diverse paradigms and provide inter-operation of these paradigms while at the same time isolate unwanted interactions between them. Furthermore, a multilingual or multiparadigm programming environment aims at accommodating the diverse execution models and mechanisms of the various paradigms, manage the resources required for implementing them, and offer intuitive ways for combining code written in a mixture of paradigms while at the same time providing orthogonal programming interfaces to the involved paradigms. There are basically two ways to produce multilingual or multiparadigm languages: either design a new superlanguage that offers the facilities of all paradigms intended to be used or provide an interface between existing languages. The first approach has the advantage of usually providing a more coherent combination of different paradigms. However, it has also the disadvantages of introducing yet another programming language that a programmer must learn plus the fact that such a single language cannot possibly support all the functionality of the languages it aims to replace. The second approach can in fact be realised with different degrees of integration ranging from using the operating system's communication primitives for inter-component collaboration to providing concrete integration of the various languages involved, as in the case of, say, Module Interconnection Languages ([63]). Multilinguality is closely related to *heterogeneity* ([70]) since heterogeneous systems (whether metacomputers or mixed-mode computers) demand that a programming language used must be able to express many useful models of computation. It is, however, usually impossible to find a single language able to deal satisfactorily with an extensive variety of such models; a mixture of language models may have to be employed.

Over the years, a number of models and metaphors were devised, their purpose being (partially) to abstract away and encapsulate the details of communication and cooperation between a number of entities forming some computation from the actual computational activities performed by these entities. A typical example is the *blackboard* model ([53]), developed for the needs of Distributed Artificial Intelligence, where a blackboard is a common forum used by a number of autonomous agents forming a multi-agent system to solve a problem in a cooperative manner. Another typical example is the *actor* model ([1]), based on the principles of concurrent object-oriented programming, where actors represent self contained computational entities, using only message passing (and not direct manipulation of each other's internal data structures) to coordinate their activities towards finding the solution for some problem. All these examples are concerned primarily with the development of parallel or distributed systems. For (mostly) sequential systems, one can mention the various component interconnection mechanisms that have been developed ([52]), with MILs (mentioned before) being an instance of the general model.

2.2 The Coordination Paradigm

The coordination paradigm offers a promising way to alleviate the problems and address some of the issues related to the development of complex distributed and parallel computer systems as these were outlined above. *Programming* a distributed or parallel system can be seen as the combination of two distinct activities: the actual *computing* part comprising a number of processes involved in manipulating data and a *coordination* part responsible for the communication and cooperation between the processes. Thus, coordination can be used to distinguish the computational concerns of some distributed or parallel application from the communication ones, allowing the separate development but also the eventual amalgamation of the these two major development phases.

The concept of coordination is closely related to those of multilinguality and heterogeneity. Since the coordination component is separate from the computational one, the former views the processes comprising the latter as black boxes; hence, the actual programming languages used to write computational code play no important role in setting up the coordination apparatus. Furthermore, since the coordination component offers a homogeneous way for interprocess communication and abstracts away the machine-dependent details, coordination encourages the use of heterogeneous ensembles of architectures.

The concept of coordination is by no means limited to Computer Science. In a seminal paper ([49]), Malone and Crowston characterise coordination as an emerging research area with an interdisciplinary focus, playing a key issue in many diverse disciplines such as economics and operational research, organisation theory and biology. Consequently, there are many definitions of what is coordination ranging from simple ones such as:

> *Coordination is managing dependencies between activities.*

to rather elaborate ones such as:

> *Coordination is the additional information processing performed when multiple, connected actors pursue goals that a single author pursuing the same goals would not perform.*

In the area of Programming Languages, probably the most widely accepted definition is given by Carriero and Gelernter ([21]):

> *Coordination is the process of building programs by gluing together active pieces.*

Consequently:

> *A coordination model is the glue that binds separate activities into an ensemble.*

A *coordination model* can be viewed as a triple **(E,L,M)**, where **E** represents the entities being coordinated, **L** the media used to coordinate the entities, and **M** the semantic framework the model adheres to ([27,71]). Furthermore, a *coordination language* is the linguistic embodiment of a coordination model, offering facilities for controlling synchronisation, communication, creation and termination of computational activities.

Closely related to the concept of coordination is that of *configuration* and *architectural description*. Configuration and architectural description languages share the same principles with coordination languages. They view a system as comprising *components* and *interconnections*, and aim at separating structural description of components from component behaviour. Furthermore, they support the formation of complex components as compositions of more elementary components. Finally, they understand changing the state of some system as an activity performed at the level of interconnecting components rather than within the internal purely computational functionality of some component.

Thus, if one adopts a slightly liberal view of what is coordination, one can also include configuration and architectural description languages in the category of coordination languages. Reversely, one can also view coordination as dealing with architectures (i.e., configuration of computational entities). In the sequel, we present a number of coordination models and languages, where by “coordination” we also mean “configuration” or “architectural description”. (However, we will not cover those cases where ordinary languages are used for coordination, such as the framework described in [57].) Other collections of work on coordination models and languages are [4,28,29], and [27] is a survey article where the focus is on the first family of models and languages according to the classification presented below.

3. Coordination Models and Languages

3.1 Data- vs. Control-Driven Coordination

The purpose of this (third) main section of the chapter is to present the most important coordination models and languages. There are a number of dimensions in which one can classify these models and languages, such as the kind of entities that are being coordinated, the underlying architectures assumed by the models, the semantics a model adheres to, issues of scalability, openess, etc. ([27,71]). Although we do provide a classification and a summary comparison of the models and languages in section 4, here we argue that these models fall into one of two major categories of coordination programming, namely either *data-driven* or *control-driven* (or task- or process-oriented).

The main characteristic of the data-driven coordination models and languages is the fact that the state of the computation at any moment in time is defined in terms of both the values of the data being received or sent and the actual configuration of the coordinated components. In other words, a coordinator or coordinated process is responsible for both examining and manipulating data as well as for coordinating either itself and/or other processes by invoking the coordination mechanism each language provides. This does not necessarily mean that there does not exist a useful clear separation between the coordination functionality and the purely computational functionality of some process. But it usually does mean that, at least stylistically and linguistically, there exists a mixture of coordination and computation code within a process definition. A data-driven coordination language typically offers some coordination primitives (coupled with a coordination metaphor) which are mixed within the purely computational part of the code. These primitives do encapsulate in a useful way the communication and configurational aspects of some computation, but must be used in conjunction with the purely computational manipulation of data associated with some process. This means that processes cannot easily be distinguished as either coordination or computational processes. It is usually up to the programmer to design his program in such a way that the coordination and the computational concerns are clearly separated and are made the responsibility of different processes; however, most of the time such a clear separation is not enforced at the syntactic level by the coordination model.

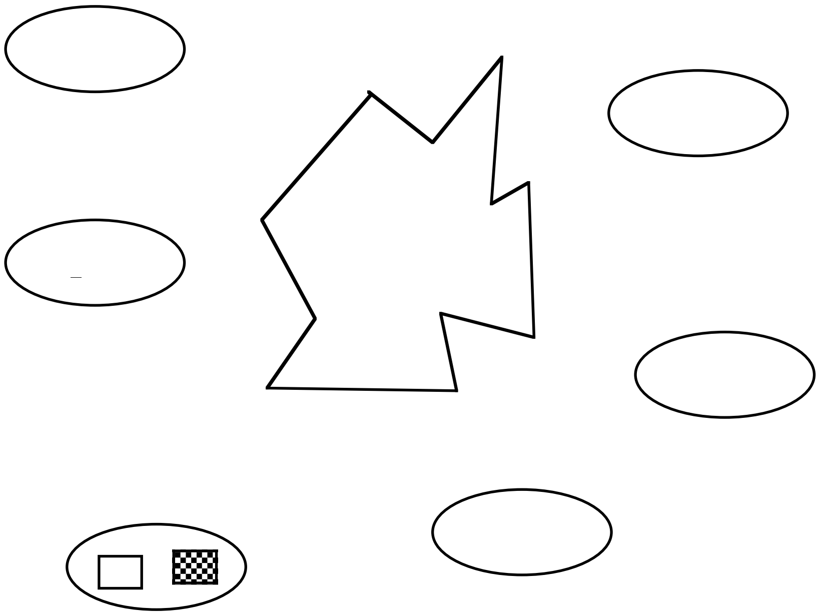
The main difference between the models of the data-driven category and those of the control-driven category is that in the latter case we have an almost complete separation of coordination from computational concerns. The state of the computation at any moment in time is defined in terms of only the coordinated patterns that the processes involved in some computation adhere to. The actual values of the data being manipulated by the processes are almost never involved. Stylistically, this means that the coordination component is almost completely separated from the computational component; this is usually achieved by defining a brand new coordination language where the computational parts are treated as black boxes with clearly defined input/output interfaces. Consequently, whereas in the case of the data-driven category, the coordination component is usually a set of primitives with predefined functionality which is used in connection with some “host” computational language, in the control-driven category the coordination component is usually a fully-fledged language. This also means that it is easier in the second case (in fact, it is usually being enforced by the model) to syntactically separate the processes (or at least program modules) into two distinct groups, namely purely computational ones and purely coordination ones.

Aside from the stylistic differences between the two categories which affect the degree of separation between computational and coordination concerns, each category also seems to be suitable for a different type of application domain. The data-driven category tends to be used mostly for parallelising computational problems. The control-driven category tends to be used primarily for modelling systems. This may be attributed to the fact that from within a configuration component a programmer has more control over the manipulated data in the case of data-driven coordination languages, than in the case of control-driven ones. Thus, the former category tends to coordinate data whereas the latter tends to coordinate entities (which, in addition to ordinary computational processes, can also be devices, system components, etc.).

We should stress the point here that the data- vs. control-driven separation is by no means a clear cut one. For instance, regarding the application domains, the data-driven coordination language LAURA is used for distributed systems whereas the control-driven coordination languages MANIFOLD and ConCoord are also used for parallelising data-intensive programs. Furthermore, regarding the degree of syntactic decoupling between the computational and the coordination components, Ariadne does have a concrete and separate coordination component, although it belongs to the data-driven category. However, the main difference between these two general categories does hold: i.e., in the data-driven category the coordination component "sees" the manipulated data whereas in the control-driven category the actual structure and contents of data is of little or no importance. In the sequel, we describe in more detail the major members of each one of these two main categories of coordination models and languages.

3.2 Data-Driven Coordination Models

Almost all coordination models belonging to this category have evolved around the notion of a *Shared Dataspace*. A Shared Dataspace ([64]) is a common, content-addressable data structure. All processes involved in some computation can communicate among themselves only indirectly via this medium. In particular, they can post or broadcast information into the medium and also they can retrieve information from the medium either by actually removing this information out of the shared medium or merely taking a copy of it. Since interprocess communication is done only via the Shared Dataspace and the medium's contents are independent of the life history of the processes involved, this metaphor achieves decoupling of processes in both space and time. Some processes can send their data into the medium and then carry or doing other things or even terminate execution while other processes asynchronously retrieve this data; a producer need not know the identity of a consumer (and vice versa) or, indeed, if the data it has posted into the medium has been retrieved or read by anyone. The following figure illustrates the general scenario advocated by most of the coordination languages in this category.



Strictly speaking, not all coordination models in this category follow the above pattern of coordination (although even for the cases of using, say, shared variables, one could treat these variables as a restricted version of a Shared Dataspace). For instance, synchronisers (section 3.1.13) use a message-passing (rather than a Shared Dataspace) based mechanism. However, all models are data-driven in the sense defined in section 3.1.

3.1.1 Linda

Linda ([2,20]) is historically the first genuine member of the family of coordination languages. It provides a simple and elegant way of separating computation from communication concerns. Linda is based on the so-called *generative communication* paradigm: if two processes wish to exchange some data, then the sender generates a new data object (referred to as a *tuple*) and places it in some shared dataspace (known as a *tuple space*) from which the receiver can retrieve it. This paradigm decouples processes in both space and time: no process need to know the identity of other processes, nor is it required of all processes involved in some computation to be alive at the same time. In addition to passive tuples containing data, the tuple space can also contain active tuples representing processes which after the completion of their execution, turn into ordinary passive tuples.

Linda is in fact not a fully fledged coordination language but a set of some simple coordination primitives. In particular, `out(t)` is used to put a passive tuple `t` in the tuple space, `in(t)` retrieves a passive tuple `t` from the tuple space, `rd(t)` retrieves a copy of `t` from the tuple space (i.e., `t` is still there) and `eval(p)` puts an active tuple `p` (i.e., a process) in the tuple space. The primitives `rd` and `in` are blocking primitives and will suspend execution until the desired tuple has been found. The primitives `out` and `eval` are non-blocking primitives. A process that executes `eval(p)` will carry on executing in parallel with `p`, which will turn into a passive tuple when it completes execution. Over the years, a number of additional primitives were introduced into the basic model; for instance `rdp(t)` and `inp(t)` are non-blocking variants of `rd(t)` and `in(t),` respectively, which when the desired tuple is not found in the tuple space will return `FALSE`.

Tuples are actually sequences of typed fields. They are retrieved from the tuple space by means of *associative pattern matching*. More to the point, the parameter `t` of the primitives `in`, `inp`, `rd` and `rdp` is actually a *tuple schemata* containing formal parameters; pattern matching of `t` with an actual tuple `ta` in the tuple space will succeed provided that the number, position, and types of `t`'s fields match those of `ta`.

The Linda primitives are indeed completely independent of the host language; thus, it is possible to derive natural Linda variants of almost any programming language or paradigm (imperative, logic, functional, object-oriented, etc.). Linda's "friends" are C, Modula, Pascal, Ada, Prolog, Lisp, Eiffel and Java to name but a few ([2,17,20,61]). The following example is an implementation of the Dining Philosophers in C-Linda. (In the Dining Philosophers problem, a number (typically 5) of philosophers are seated around a table, with a plate of food (typically spaghetti) in front of each one of them, and a fork between each plate. To eat, they must use both forks adjacent to them. But if all act independently, then each may pick up the left fork first and all enter a "deadlocked" state waiting forever for the right fork to become available. This is a standard problem for developing coordination solutions among independently executing processes.

```
#define NUM 5

philosopher(int i)                   main()
{                                    {
 while (1)                            int i;
 {                                    for (i=0, i<=NUM, i++)
  think();                            {
  in("room ticket");                   out("fork",i);
  in("fork",i);                        eval(philosopher(i));
  in("fork",(i+1)%NUM);                if (i<(NUM-1))
```

```
    eat();
    out(“fork”,i);
    out(“fork”,(i+1)%NUM);
    out(“room ticket”);
  }
}
```

```
      out(“room ticket”);
    }
  }
```

Although the Linda model is appealing, it is also deceptively simple when it comes to implementing it, especially if distributed (as it is usually the case) environments are to be considered. There are a number of issues that a Linda implementor must address such as where precisely the tuples are stored and how are they retrieved, how load balancing is achieved, choices in implementing the `eval` primitive, etc. Furthermore, since programmers usually adhere to specific communication and coordination patterns, often used protocols should be optimised. There are a number of different approaches in implementing Linda ([34,72]). Piranha ([41]) for example is an execution model for Linda, particularly suited to networks of workstations. Piranha features *adaptive parallelism* where processor assignment to executed processes changes dynamically. A typical Piranha program is usually a variant of the master-slave paradigm. In particular, there exists a *feeder* process responsible for distributing computations and selecting results and a number of *piranhas* which perform computations. Piranhas are statically distributed over the available nodes in a network (i.e., they do not migrate at run-time). They remain *dormant* as long as the node they reside on is *unavailable*; when the node becomes *available* they get activated. If the node is claimed back by the system and is removed from the list of available nodes then the piranha residing on it must *retreat*. Work on the current task is stopped and the retreating piranha posts to the tuple space enough information to allow some other piranha to take up the rest of the work. Typical applications suitable for the Piranha paradigm are Montecarlo simulations and LU decompositions. The general structure of a Piranha program is as follows.

```
#define DONE -999
int index;

feeder()
{
  int count;
  struct Result result;

  /* put out the tasks */
  for (count=0; count<TASKS; count++)
    out(“task”,count);

  /* help compute results */
  piranha();

  /* collect results */
  for (count=0; count<TASKS; count++)
    in(“result”,count,?result_data);
}

retreat()
{
  /* replace current task */
  out(“task”,index);
}
```

```
piranha()
{
  struct Result result;

  while (1)
  {
    in(“task”,?index);
    if (index==DONE)
    {
      /* all tasks are done */
      out(“task”,index);
    }
    else
    {
      /* do the task */
      do_work(index,&result);
      out(“result”,index,result);
      in(“tasks done”,?index);
      out(“tasks done”,i+1);
      if ((i+1)==TASKS)
          out(“task”,DONE);
    }
  }
}
```

Linda has inspired the creation of many other similar languages — some are direct extensions to the basic Linda model but others differ significantly from it. These derivatives aim to improve and extend the basic model with multiple tuple spaces, enforcement of security and protection of the data posted to the tuple space, etc. Some of these proposals will be described promptly.

3.1.2 Bauhaus Linda

Bauhaus Linda ([22]) is a direct extension of the "vanilla" Linda model featuring multiple tuple spaces implemented in the form of *multisets* (msets). Bauhaus Linda does not differentiate between tuples and tuple spaces, tuples and anti-tuples (i.e., tuple templates) and active and passive tuples. Instead of adding tuples to and reading or removing tuples from a single flat tuple space, Bauhaus Linda's `out`, `rd` and `in` operations add multisets to and read or remove multisets from another multiset. Consequently, Linda's *ordered* and position dependent associative pattern matching on tuples is replaced by *unordered* set inclusion.

Assuming the existence of the multiset `{a b b {x y Q} {{z}} P}` where elements in capital letters denote processes and the rest denote ordinary tuples, then if `P` executes `out {x->R}` the multiset will become `{a b b {x y Q R} {{z}} P}`. If the multiset has the form `{a b b {x y} {{z}} P}` and `P` executes `mset m := rd {x}` then `m` will get assigned the structure `{x y}`. Finally, if the multiset has the form `{a b b {x y Q} {R {z}} P}` and `P` executes `mset m := in {x}`, then `m` is assigned the structure `{x y Q}` (and thus it becomes a live mset due to the presence of the element `Q`).

Furthermore, the language introduces the new primitive `move` which allows tuples to move up and down the levels of a multiset. For instance, if the multiset has the form `{a b b {x y Q} {w {z}} P}` and `P` executes `move {w}` then the result is `{a b b {x y Q} {w {z} P}}`. There also exist two other variants of `move`: `up`, which causes the issuing process to go one level up the structure, and `down m` (equivalent to `move -> {m}` where `m` is a multiset) which causes the issuing process to go down to a sibling node that contains `m`.

It is thus possible to organise data into useful hierarchies, such as:

```
{"world"
  {"africa" …}
  {"antarctica" …}
  …
  {"north america"
    {"usa" …
      {"ct" …
        {"new haven" …
          {"yale" …
            {"cs" …}
          …}
        …}
      …}
    …}
  …}
}
```

and assuming that the vector of strings `Path[]` represents a travelling course, we can move around by means of commands such as:

```
for (i=0; Path[i]; i++) down {Path[i]};
```

3.1.3 Bonita

Bonita ([65]) is a collection of new Linda-like primitives aiming at enhancing the functionality of the basic model as well as improve its performance. The first goal is achieved by providing the functionality for multiple tuple spaces and aggregate tuple manipulation. The second goal is achieved by providing a finer grain notion of tuple retrieval where a request by some process for finding a tuple in the tuple space is treated separately from checking whether it has actually been delivered to the requesting process; thus, these activities can be done in parallel and consequently the related overhead is minimised.

In particular, Bonita supports (among others) the following coordination primitives:

`rquid=dispatch(ts, tuple)`
: Non-blocking; puts `tuple` in `ts` and returns a tuple id to be used by other processes which may want to retrieve the tuple.

`rquid=dispatch(ts, template, d|p)`
: Non-blocking; retrieves a tuple from `ts` matching `template`, either removing it (if `d` is specified) or getting a copy of it (if `p` is specified), and returns a request id as before.

`rquid=dispatch_bulk(ts1, ts2, template, d|p)`
: Non-blocking; moves (`d`) or copies (`p`) from `ts1` to `ts2` all tuples matching `template`.

`arrived(rquid)`
: Non-blocking; tests *locally* in the environment of the issuing process whether the tuple with the indicated id has arrived, and returns true or false accordingly.

`obtain(rquid)`
: Blocking; suspends until the tuple with the indicated id is available.

The rather subtle differences between the basic Linda model and Bonita can be understood by considering the retrieval of three tuples from a single tuple space (in general, more than one tuple space may be involved):

```
Linda                          Bonita

                               int rqid1, rqid2, rqid3;

in("ONE");                     rqid1=dispatch(ts,"ONE",d);
in("TWO");                     rqid2=dispatch(ts,"TWO",d);
in("THREE");                   rqid3=dispatch(ts,"THREE",d);

                               get(rqid1);
                               get(rqid2);
                               get(rqid3);
```

Whereas in Linda, the first `in` operation must complete execution before the second one commences, Bonita follows a finer grain scenario: the three requests for the sought tuples are dispatched and the underlying system starts executing them in parallel. Thus, the time taken to retrieve one tuple can be overlapped with the time taken to retrieve the other tuples. In Bonita, it is also possible to express more efficiently (than in the basic Linda model) a non-deterministic selection construct:

```
Linda                          Bonita

                               int rqid1, rqid2;

                               rqid1=dispatch(ts,"ONE",d);
                               rqid2=dispatch(ts,"TWO",d);

while(1)                       while(1)
{                              {
 if (inp("ONE"))                if (arrived(rqid1))
    { do_first(); break; }         { do_first(rqid1); break; }
 if (inp("TWO"))                if (arrived(rqid2))
    { do_second(); break; }        { do_second(rqid2); break; }
}                              }
```

The Linda version repeatedly polls the global tuple space in order to check as to whether the requested tuples have appeared; furthermore, execution of the two `inp` operations is serialised. The Bonita version dispatches the two requests, which the underlying system is now able to serve in parallel, and then keeps checking locally in the environment of the process whether the requested tuples have arrived. The literature on Bonita shows that in many cases the finer grain notion of tuple handling that this model has introduced produces substantial gains in performance.

3.1.4 Law-Governed Linda

Whereas both Bauhaus Linda and Bonita extend the basic language by actually modifying the model and its underlying implementation, Law-Governed Linda ([51]) superimposes a set of "laws" which all processes wishing to participate in some exchange of data via the tuple space must adhere to. In particular, there exists a *controller* for every process in the system, and all controllers have a copy of the law. A controller is responsible for intercepting any attempted communication between the process it controls and the rest of the world. The attempted communication is allowed to complete only if it adheres to the law.

A Law-Governed Linda system is understood as a 5-tuple `<C,P,CS,L,E>` where `C` is the tuple space, `P` is a set of processes, `CS` is a set of control states (one associated with each process), `L` is the law which governs the system, and `E` is the set of controllers that enforce the law. Although the law can be formulated in any language (and a natural choice would be the host language that a Linda system is using), the designers of Law-Governed Linda have chosen a restricted subset of Prolog enhanced with the following primitives: `complete` actually carries out the operation being invoked; `complete(arg′)` does the same but the original argument of the operation is replaced with `arg′`; `return` is used in conjunction with the Linda primitives `rd` and `in` and its effect is to actually deliver the requested tuple to the issuing process; `return(t′)` does the same but `t′` is delivered instead of the matched (from a previous `rd` or `in` operation) tuple `t`; `out(t)` is the conventional Linda primitive; `remove` removes the issuing process from the system. Furthermore, tuple terms may contain the following attributes: `self(i)` where `i` is the unique id of the process; `clock(t)` where `t` is the current local time of the process; `op(t)` where `t` is the argument of the latest Linda operation invoked by the issuing process. In addition, `do(p)` adds `p` (a sequence of one or more primitive operations) to a *ruling list* R which is executed after the ruling of the law is complete, and `+t` and `-t` add and remove respectively a term `t` from the control state of a process. Finally, every process is in a certain *state* represented by the global variable `CS` and examined by means of the operator '`@`'.

A user may now enhance the basic Linda functionality by formulating a suitable law. For instance, the following Prolog clauses implement secure message passing, where a message is a tuple of the form `[msg,from(s),to(t),contents]` where `msg` is a tag identifying the tuple as a message, `s` is the sender process, `t` is the receiver process, and `contents` represents the actual contents of the message.

```
R1. out([msg,from(Self),to(_)|_]) :- do(complete).
R2. in([msg,from(…),to(Self)|_]) :- do(complete) :: do(return).
R3. out([X|_]) :- not(X=msg), do(complete).
R4. in/rd([X|_]) :- not(X=msg), do(complete) :: do(return).
```

R1 allows a process to `out` messages only with its own id in the sender field (thus a message cannot be forged). R2 allows a process to remove a tuple from the tuple space only when its own id is in the receiver field. R3 and R4 simply send to or read/retrieve from the tuple space, respectively, any other tuple which need not adhere to the above law.

It is also possible in Law-Governed Linda to establish multiple tuple spaces by means of laws like the following:

```
R1. out([subspace(S)|_]) :- hasAccess(S)@CS, do(complete).
R2. in/rd([subspace(S)|_]) :-  actual(S), hasAccess(S)@CS,
                               do(complete) :: do(return).
```

where initially every process has any number of `hasAccess(s)` terms in its control state, identifying the subspaces it can access. R1 allows a process to `out` a tuple to the subspace `s` provided the term `hasAccess(s)` is in its control state. R2 uses the primitive `actual` to make sure that the variable representing the name of a subspace is actually instantiated to some ground value (a process cannot attempt to query multiple subspaces at once) before proceeding to retrieve or read a tuple from the subspace.

3.1.5 Objective Linda

Objective Linda ([45]) is another direct variant of the basic Linda model, influenced by Bauhaus Linda and particularly suited to modelling open systems. Objective Linda introduces an *object model* suitable for open systems and independent of the host programming language (objects in Objective Linda are described in the *Object Interchange Language* (OIL) — a language-independent notation). Objects make use of the object space by means of suitably defined object-space operations which address the requirements of openess. Furthermore, the language supports hierarchies of multiple object spaces and ability for objects to communicate via several object spaces. Object spaces are accessible through *logicals* (i.e. object space references passed around between objects); logicals can be `out`ed by some object to the tuple space and then retrieved by another space via a special `attach` operation.

In particular, Objective Linda supports, among others, the following operations which are variants of Linda's basic coordination primitives:

`bool out(MULTISET *m, double timeout)`
: Tries to move the objects contained in `m` into the object space. Returns true if the attempted operation is successful and false if the operation could not be completed within timeout seconds.

`bool eval(MULTISET *m, double timeout)`
: Similar to the previous operation but now the moved objects are also activated for execution.

`MULTISET *in(OIL_OBJECT *o, int min, int max, double timeout)`
: Tries to remove multiple objects $o'_1 \ldots o'_n$ matching the template object `o` from the object space and returns a multiset containing them if at least `min` matching objects could be found within `timeout` seconds. In this case, the multiset contains at most `max` objects, even if the object space contained more. If `min` objects could not be found within `timeout` seconds, `NULL` is returned instead.

`MULTISET *rd(OIL_OBJECT *o, int min, int max, double timeout)`
: Similar to the previous operation but now clones of multiple objects $o'_1 \ldots o'_n$ are returned.

`int infinite_matches`
: Constant value interpreted as infinite number of matching objects when provided as a `min` or `max` parameter.

`double infinite_time`
: Constant value interpreted as infinite delay when provided as a `timeout` parameter.

The following C++ example models a collision avoidance scenario for cars driving in a cyclic grid.

```
class Position: public OIL_OBJECT
{
 private: bool match_position;    // switching the matching mode
 public:  int x, y;               // the grid position
          int car;                // the car's id
 bool match(OIL_OBJECT* obj)
```

```
  {
   if (match_position)
     return ( (((Position*)obj)->x == y) && (((Position*)obj->y==y) );
   else return ((Position*)obj)->car==car;
  };
  set_match_position() { match_position=true; }
  set_match_car()      { match_position=false; }
};

class Car: public OIL_OBJECT
{
  private: int x, y;               // the grid position
           int car;                // the car's id
           direction dir;          // the direction to move
  void wait(){};  // wait for an arbitrary interval
  void evaluate()
  {
   MULTISET m = new MULTISET;
   Position *p; int nx, ny, px, py;
   while(true)
   {
     m->put(new Position(id,x,y));
     (void)context->out(m,context->infinite_time);
     wait();
     // store next position to move to in nx and ny
     nx = … ; ny = … ;
     p = new Position(id,nx,ny); p->set_match_position();
     m = context->rd(p,1,1,0);
     if (m)   // there is a car in front of us
       { delete m; delete p; }
     else {  delete p;
             // store position with priority in px and py
             px = … ; py = … ;
             p = new Position(id,px,py); p->set_match_position();
             m = context->rd(p,1,1,0);
             if (m)   // there is a car with priority
               { delete(m; delete p; }
             else  {  x = nx; y = ny   // move
                      delete p;
                   }
          }
     }
     p = new Position; p->car = id; p->set_match_car();
     m = context->in(p,1,1,context->infinite->time);
     p = m->get(); delete p;
   }
  }
}
```

Agents responsible for steering cars communicate via the object space. Every agent puts an object of type `Position` into the object space which carries the agent's id as well as its position and grid. When changing its position, an agent consumes (`in`s) the `Position` object with its own id and replaces it by a new one with the updated position. Before an agent changes position, it checks whether it can `rd` a `Position` object directly in front of it, in which case it will wait. Also, it checks as to whether some other car is approaching from another direction, in which case again it waits. This functionality is modelled by `Car`.

3.1.6 LAURA

LAURA ([69]) is another approach of deriving a Linda-like coordination model suitable for open distributed systems. In a LAURA system, *agents* offer *services* according to their functions. Agents communicate via a *service-space* shared by all agents and by means of exchanging forms. A *form* can contain a description of a *service-offer*, a *service-request* with arguments or a *service-result* with results. Suitable primitives are provided by the language with the intention to put and retrieve offer-, request- and result-forms to and from the service-space. More to the point, `SERVE` is used by clients to ask for service, `SERVICE` is used by servers to offer service and `RESULT` is used by servers to produce a result-form. Identification of services is not done by means of naming schemes but rather by describing an interface signature consisting of a set of operation signatures. Operation signatures consist of a name and the types of arguments and parameters. As in the case of Objective Linda, a host language-independent *interface description language* (STL for Service Type Language) is used for this purpose. Furthermore, the service-space is monolithic and fault tolerant.

The following code models the activities of a travel ticket purchase system.

```
SERVE large-agency operation
 (getflightticket  :  cc * <day,month,year> * dest ->
                      ack * <dollar,cent>;
 getbusticket      :  cc * <thedate.day,thedate.month,thedate.year>
                      * dest -> ack * <dollar,cent> * line;
 gettrainticket    :  cc * <day,month,year> * dest ->
                      ack * <dollar,cent>).
SERVE

RESULT large-agency operation
 (getflightticket  :  cc * <day,month,year> * dest ->
                      ack * <dollar,cent>;
 getbusticket      :  cc * <thedate.day,thedate.month,thedate.year>
                      * dest -> ack * <dollar,cent> * line;
 gettrainticket    :  cc * <day,month,year> * dest ->
                      ack * <dollar,cent>).
RESULT

SERVICE small-agency
 (getflightticket  :  cc * <thedate.day,thedate.month,thedate.year>
                      * dest -> ack * <dollar,cent>;)
SERVICE
```

A service with the interface `large-agency` is offered and the code for the selected operation should be bound to `operation`. Depending on which one of the three services offered by `large-agency` requested, the respective program variables (such as `cc`, `day`, etc.) will be bound with the arguments offered by the service-user. When `SERVE` executes, a serve-form is built from the arguments, the service-space is scanned for a service-request form whose service-type matches the offered type, and the code of the requested operation and the provided arguments are copied to the serve-form and bound to the program variables according to the binding list. After performing the requested service, the service-provider uses `RESULT` to deliver a result-form to the service-space. Finally, another agent with interface `small-agency` which wishes to invoke the service `getflightticket` executes `SERVICE`; program variables are then bound accordingly in order for `small-agency` to provide `large-agency` with needed information (such as `cc` and `dest`) and also let `large-agency` pass back to `small-agency` the results of the service (such as `ack`).

3.1.7 Ariadne/HOPLa

Ariadne and its modelling language HOPLa ([36]) are an attempt to use Linda-like coordination to manage hybrid collaborative processes. As in all the Linda-like models, Ariadne uses a shared workspace, which however holds tree-shaped data and is self-descriptive, in the sense that in addition to the actual data it also holds constraints (i.e., type definitions) that govern its structure. Both highly structured data (e.g., forms consisting of typed fields) and semi-structured data (e.g., email messages) can be handled. The processes comprising an Ariadne system are defined in the Hybrid Office Process Language (HOPLa) and use the concept of *flexible records* enhanced with constructors such as `Set` for collections (aggregate operations) and constraints. Tasks are defined by means of `Action` terms and use the following coordination operators: `Serie` for sequential execution, `Parl` for parallel execution, and `Unord` for execution in random order. The following example models an electronic discussion between a group of people:

```
Discussion<Process(
    group -> Set+Action(type -> Actor; value -> PS: set));
    discuss -> Thread<Data+Serie(
        message -> String+Action(actor -> {p | p in PS));
        replies -> Set+Parl(type -> Thread)))
```

First, the set of actors participating in the discussion is defined by setting the feature group. After the group has been established, a string for the message feature must be provided by one of the actors in the group. After that, replies can be added in parallel by other members of the group, each one spawning a different thread of execution.

3.1.8 Sonia

Sonia ([10]) is another approach to using Linda-like coordination languages to model activities in Information Systems. In fact, Sonia is not so much an extension of Linda with extra functionality but rather an adaptation of the latter for coordinating human and other activities in organisations. The basic Linda functionality is expressed by higher level metaphors which should be understandable by everyone, including non-computer specialists.

Thus, in Sonia, there exists an *agora* (the equivalent of a tuple space) and a number of *actors* communicating by means of posting to and extracting messages from it. An agora ([50]) is a first class citizen and many (nested) agoras can be defined. Messages are written as tuples of named values, e.g. `Tuple(:shape “square” :color “red”)`. Nested and typed tuples are also supported. The traditional Linda primitives `out`, `in` and `rd` are replaced by the more intuitively named primitives `post`, `pick` and `peek`, respectively. A new primitive, `cancel`, is also introduced intended to abort an outstanding `pick` or `peek` request. Templates can be enhanced with timeout functionality and rules: the template `Template(:shape any :color Rule(“value=‘red’ or :[value=‘blue’]”))` would match a tuple with any shape as long as and colour is either red or blue.

3.1.9 Linda and the World Wide Web

Recently, the concept of coordination has been introduced into the development of middleware web-based environments. The Shared Dataspace paradigm of Linda is particularly attractive for orchestrating distributed web-based applications and a number of extensions have been proposed to the basic Linda suitable for developing interactive WWW environments. The basic advantage of introducing a coordination formalism into a web application is that it then becomes easier to separate I/O from processing concerns. CGI scripts deal only with I/O (e.g., getting data by means of electronic forms and/or displaying them) whereas the coordination formalism becomes responsible for the sending/receiving of data via the tuple space. The system is also enhanced with the two major features of a coordination formalism: heterogeneous execution of applications and multilinguality ([31]).

Jada ([30]) is a combination of Java with Linda, able to express mobile object coordination and multithreading and is suited for open systems. Suitable classes such as `TupleServer` and TupleClient have been defined for providing remote access to a tuple space and communication is done by means of sockets. A TupleClient needs to know the host and port id of TupleServer and the language provides appropriate constructs for specifying this information. Jada can be used both as a coordination language *per se* and as a kernel language for designing more complex coordination languages for the WWW. The following Jada program implements a symmetric ping-pong.

```
//--PING--
import jada.tuple.*;
import java.client.*;

public class Ping
{
 static final String ts_host=“foo.bar”;

 public void run()
 {
  // a tuple client interacts with remote server
  TupleClient ts = new TupleClient(ts_host);
  // do ping-pong
  while (true)
   {
    ts.out(new Tuple(“ping”));
    Tuple tuple = ts.in(new Tuple(“pong”));
   }
 }

 public static void main(String args[])
 {
  Ping ping = new Ping();
  ping.run();
 }
}

//--PONG--
import jada.tuple.*;
import java.client.*;

public class Pong
{
 static final String ts_host=“foo.bar”;

 public void run()
 {
  // a tuple client interacts with a remote server
  TupleClient ts = new TupleClient(ts_host);
  // do ping-pong
  while (true)
   {
    ts.out(new Tuple(“pong”));
    Tuple tuple = ts.in(new Tuple(“ping”));
   }
 }

 public static void main(String args[])
 {
  Pong pong = new Pong();
  pong.run();
  }
}
```

SHADE ([24]) is a higher level object-oriented coordination language for the Web. In SHADE the coordinated entities are Java objects. However, whereas Jada is based on singleton tuple transactions, SHADE is based on multiset rewriting. Each object in SHADE has a name, a class and a state. The name is the pattern used to deliver messages. The type defines the object behaviour. The state is the contents of the multiset associated with the object. The above ping-pong program can be defined in SHADE as follows.

```
class ping_class =                class pong_class =
{                                 {
 in do_ping;                       in do_pong;
 send pong, do_pong                send ping, do_ping
 #                                 #
 in done;                          in done;
 terminate                         terminate
}                                 }
```

Each class comprises two methods (separated by '#'). The first method is activated when one of the items `ping` or `pong` appear in the proper object's multiset. When the method is activated in the object `ping` (say), a message (`do_pong`) is sent to the object `pong` (and vice versa). When the message is delivered to an object, it is put in the object's multiset and triggers the activation of the first method of that object, and so on. The second method is triggered by the item `done` and causes the termination of the object.

3.1.10 GAMMA

The GAMMA (General Abstract Model for Multiset mAnipulation) model ([9]) is a coordination framework based on *multiset rewriting*. The basic data structure in GAMMA is a *multiset* (or *bag*), which can be seen as a chemical solution and, unlike an ordinary set, can contain multiple occurrences of the same element. A simple program is a pair (Reaction Condition, Action) and its execution involves replacing those elements in a multiset satisfying the reaction condition by the products of the action. The result is obtained when no more such reactions can take place and thus the system becomes stable.

There is a unique control structure associated with multisets, namely the Γ operator, whose definition is as follows.

```
Γ((R1,A1),…,(Rm,Am)) (M) =
   if   ∀ i ∈ [1,m] ∀ x1,…,xn ∈ M, ¬Ri(x1,…,xn)
   then M
   else let x1,…,xn ∈ M, let i ∈ [1,m] such that ¬Ri(x1,…,xn) in
            Γ((R1,A1),…,(Rm,Am)) ((M-{x1,…,xn})+Ai(x1,…,xn))
```

where `{…}` represents multisets and `(Ri,Ai)` are pairs of closed functions specifying reactions. The effect of `(Ri,Ai)` on a multiset `M` is to replace in `M` a subset of elements `{x1,…,xn}` such that `Ri(x1,…,xn)` is true for the elements of `Ai(x1,…,xn)`. GAMMA enjoys a powerful *locality property* in that `Ri` and `Ai` are pure functions operating on their arguments. Thus, if the reaction condition holds for several disjoint subsets, the reactions can be carried out in parallel. The following code implements a prime number generator.

```
prime_numbers(N)  =  Γ((R,A))
                     ({2,…,N}) where
                     R(x,y) = multiple(x,y)
                     A(x,y) = {y}
```

The solution consists of removing multiple elements from the multiset `{2,…,N}`. The remaining multiset contains exactly the prime numbers less than `N`.

Although the operational behaviour of the model is strictly implicit (the programmer does not specify any order of execution and the latter is by default completely parallel), practical use of it reveals that a number of program schemes can be identified which are the ones most often used by programs. These schemes, referred to as *tropes*, are: `Transmuter(C,f)`, which applies the same operation `f` to all the elements of the multiset until no element satisfies the condition `C`; `Reducer(C,f)`, which reduces the size of the multiset by applying the operation `f` to pairs of elements satisfying `C`; `Optimiser(<,f1,f2,S)`, which optimises the multiset according to some criterion expressed through the ordering `<` between the functions `f1` and `f2`, while preserving the structure `S` of the multiset; `Expander(C,f1,f2)`, which decomposes the elements of a multiset into a collection of basic values according to the condition `C` and by applying `f1` and `f2` to each element; `S(C)`, which removes from the multiset all those elements satisfying `C`. Tropes can be combined together to form complex programs; the following combination of tropes (as opposed to using the fundamental Γ operator) implements the functionality of a Fibonacci function:

```
fib(n) =  add(zero(dec({n})))
dec    =  E(C,f1,f2) where C(x)=x>1, f1(x)=x-1, f2(x)=x-2
zero   =  T(C,f) where C(x)=(x=0), f(x)=1
add    =  R(C,f) where C(x,y)=true, f(x,y)=x+y
```

The vanilla GAMMA model has been enriched with sequential and parallel operators, higher-order functionality and types. Furthermore, in order to express structure, a variant of the basic model has been proposed, namely Structured GAMMA, featuring *structured multisets*. These can be seen as a set of addresses satisfying specific relations and associated with a value. A type is defined in terms of rewrite rules and a structured multiset belongs to a type T if its underlying set of addresses satisfies the invariant expressed by the rewrite system defining T. In Structured GAMMA reactions test and/or modify the relations on addresses and/or the values associated with those addresses. Structured GAMMA is particularly suited to the concept of coordination since addresses can be interpreted as individual entities to be coordinated. Their associated value defines their behaviour (in a given programming language which is independent of the coordination one) and the relations correspond to communications links. Also, a structuring type provides a description of the shape of the overall configuration. Structured GAMMA has been used for modelling software architectures, a particular type of coordination usually modelled by the coordination languages of the control-driven family. For instance, the following Structured GAMMA code models a client-server architecture:

```
CS  =  N n
N n =  cr c n, ca n c, N n
N n =  sr n s, sa s n, N n
N n =  m n
```

where `cr c n` and `ca n c` denote respectively a communication link from a client `c` to a manager `n` and the dual link from `n` to `c`. A new client can now be added by means of the rewrite rule

```
m n => m n, cr c n, ca n c
```

3.1.11 LO and COOLL

Linear Objects (LO,[3,5,15]) is an object-oriented language based on the Interaction Abstract Machines computational model. The concept of “interaction” is closely related to the principles underpinning the theory of Linear Logic and in fact LO amalgamates linearity with multiset rewriting (as in GAMMA) and asynchronous, actor-like communication by means of broadcasting. LO views the computation as a system of communicating agents whose state is represented as a multiset. Agents evolve in terms of transitions which are transformations from one state to another; in addition, agents can be created or terminate. Inter-agent communication is achieved by means of broadcasting.

An LO program is a set of (effectively multi-headed) rewrite rules taking the general form

```
<multiset> <broadcast> <built-ins> ⊸ <goal>

multiset = a1 @ … @ an
broadcast = ^a | ^a @ broadcast
goal = a1 @ … @ an | goal1 & … & goaln | #t | #b
```

where the symbols '@' ("par"), '&' ("with"), '⊸' (implication), `#t` ("top") and `#b` ("bottom") are taken from Linear Logic. The following example implements Multiminds, a multiplayer version of MasterMind.

```
coder(S) @ current(I) @ ^go(I) ⊸ coder(S)
/* coder calls the player ("go(I)") */

coder(S) @ try(I,G) @ players(N) @ ^result(I,G,B,C) @
    { answer(S,G,B,C), C=\=0, next_player(N,I,I1) } ⊸
        coder(S) @ current(I1) @ players(N).
/* coder sends to the player I the answer (bulls B and cows C) to the guess
G. */

coder(S) @ try(I,G) @ ^victory(I,G) @
    { answer(S,G,B,C), C=:=0 } ⊸ #t.
/* player I has guessed the secret code with G – coder informs players
before ending */

decoder @ alp_l([A|List]) @ n_decod(N) @ { nextplayer(N,N1) } ⊸
    decoder @ alp_l(List) @ n_decod(N1) &
    decoder(N) @ db([]) @ alph(A).
/* creation of players */

decoder(I) @ go(I) @ db(L) @ alph(A) @ ^try(I,G) @ { compute(A,L,G) } ⊸
    decoder(I) @ db(L) @ alph(A).
/* after receiving the message "go(I)", player I computes a guess G, sends
it to the coder ("try(…)"), and waits for an answer */

decoder(I) @ result(I,G,B,C) @ db(L) ⊸
    decoder(I) @ db([tried(G-[B,C])|L]).
/* player I stores the answer to the guess G ("result(…)") */
```

Note that the purely computational procedures `answer` and `compute` would normally be implemented in some ordinary programming language, with Prolog or one of its variants being the most natural choice. A initial query to the program above could be the following:

```
coder([h,a,l,l,o]) @ players(2) @ current(0) &
decoder @ n_decod(0) @ alp_l([[l,o,l,h,a],[h,l,a,o,l]])
```

COOLL ([23]) extends the basic LO model with modularity and a new form of broadcast-like communication, namely *group* (multicast) communication. A COOLL program is a set of theories, each theory having the following structure:

```
theory  theory_name ⊸
            method1
            #
```

```
                .
                .
                .
                #
                methodN
```

Communication can be either broadcast or group communication, the latter directed to specific theories:

```
Communications = ^A | !(dest,msg) | Communications @ Communications
```

where `dest` is the name of a theory which will receive `msg`.

Methods have the general form:

```
Conditions => Communications => Body
```

where `Conditions` specify when methods will be triggered, `Communications` specifies broadcast and/or group communication, and `Body` defines a transition to a new configuration. The previous LO program can be written in COOLL as follows.

```
theory coder ⊸
    current(I) => !(decoder,go(I)) => #b
  #
    try(I,G) @  { code(S) @ players(N) } @
                { { answer(S,G,B,C),C=\=0, next_player(N,I,I1) } }
                => !(decoder,result(I,G,B,C)) => current(I1)
  #
    try(I,G) @  { code(S) } @
                { { answer(S,G,B,C), C=:=0 } }
                => ^victory(I,G) => #t.

theory decoder ⊸
    alpl([L|List]) @ n_decod(N) @ { next_player(N,N1) }
        => => alp_l(List) @ n_decod(N1) & id(N) @ db([]) @ alph(L)
  #
    go(I) @  { id(I) @ alph(A) @ db(L) } @
             { { compute(A,L,G) } } => !(coder,try(I,G)) => #b
  #
    result(G,B,C) @ db(L) => => db([tried(G-[B,C])|L])
  #
    victory(X,G) => => #t
```

The program is structured naturally into two theories; furthermore, through employing group communication, the decoder refrains from receiving guesses from other decoders. The previous query now takes the following form:

```
*coder @ code([h,a,l,l,o]) @ players(2) @ current(0) &
*decoder @ n_decod(0) @ alp_l([[l,o,l,h,a],[h,l,a,o,l]])
```

where ‘`*`’ denotes the name of a theory.

3.1.12 MESSENGERS

MESSENGERS ([40]) is a coordination paradigm for distributed systems, particularly suited to mobile computing. MESSENGERS is based on the concept of *Messengers* which are autonomous messages. A Messenger, instead of carrying just data (as is the case for ordinary passive messages), contains a process, i.e. a program together with its current status information (program counter, local variables, etc.). Each node visited by the Messenger resumes the interpretation of the Messenger’s program until a navigational command is encountered that causes it to leave the current node. A distributed application is thus viewed as a collection of functions whose coordination is managed by a group of Messengers navigating freely and autonomously through the network. In addition to navigational autonomy, MESSENGERS supports both inter- and intra-object coordination.

As is the case for the Linda-like paradigms, MESSENGERS also supports the concept of a structured global state space. However, MESSENGERS explicitly partitions it by means of the navigational features it supports. The following example models a manager-worker scenario.

```
manager_worker()
{
 create(ALL);
 hop(ll = $last);
 while (task = next_task()) != NULL
   {
    hop(ll = $last);
    res = compute(task);
    hop(ll = $last);
    deposit(res);
   }
}
```

The above Messenger script is injected into the *init* node of some daemon. It first creates logical nodes connected to the current node on every neighboring daemon. It also causes a replica of the Messenger to be created on each node and start executing. Each of the Messengers hops back to the original node by following the most recently traversed logical link, which is obtained by accessing the system variable `$last`. It then attempts to get a new task to work on. If successful, it hops back to its logical node, computes the result, and carries it back to the central node to deposit it there. This activity is repeated until no further work is left to do in which case it ceases to exist. The primary role in the Messenger functionality is played by the `hop` statement which is defined as follows:

```
hop(ln=n;ll=l;ldir=d)
```

where `ln` represents a logical node, `ll` represents a logical link and `ldir` represents the link's direction. The triple `(n,l,d)` is a destination specification in the network; `n` can be an address, a variable, or a constant (including the special node `init`); `l` can be a variable, a constant, or a virtual link (corresponding to a direct jump to the designated node); finally, `d` can be one of the symbols '`+`', '`-`' or '`*`' denoting "forward", "backward" or "either" respectively, with the latter playing also the role of being the default value. The wild card '`*`' applies also to `n` and `l` with obvious meanings.

As another example consider the matrix multiplication.

```
distribute_A(s)
{
 M_sched_time_abs((j-i) mod s);
 msgr_A = copy_block(resid_A);
 hop(ll = "row");
 new_resid_A = copy_block(msgr_A);
}

rotate_B(m)
{
 msgr_B = copy_block(resid_B);
 for (k=0; k<m; k++)
   {
    M_sched_time_dlt(.5);                                /* synchronisation */
    resid_C = block_multiply(msgr_B,resid_A,resid_C);    /* Cij=Aij*Bij */
    hol(ll = “column”; ldir = -);            /* rotate B to column i-1 */
```

The code comprises two Messengers. `distribute_A` implements the movement of the array `A` using *temporal coordination* which is also supported by the model (each of the `distribute_A` Messengers schedules itself to wake up at the time corresponding to its position in the logical network). `rotate_B` is the embodiment of one of the blocks of the matrix `B` which it copies from the node `resid_B` to its private area `msgr_B`. It then enters a loop during which it keeps moving the block it is responsible for along its respective column. Temporal coordination is also used here and in fact the two Messengers always alternate between their respective executions. Each time `rotate_B` wakes up, it performs a block multiplication using its own block of `B` and the currently resident block of `A`, adds it to the resident block of `C` and hops to its northern neighbor.

3.1.13 Synchronisers

Synchronisers ([38,39,62]) are based on the Actor model of computation and are a set of tools able to express coordination patterns within a multi-object language framework based on specifying and enforcing constraints that restrict invocation of a set of objects. Constraints are defined in terms of the interface of objects being invoked rather than their internal representation. Constraints can enforce access restrictions either temporarily or permanently. These constraints are typically used to express certain properties fundamental to concurrent object-oriented languages such as temporal ordering or atomicity of method invocation.

Synchronisers are expressed in an abstract format which is independent of both the syntax of the particular host languages as well as the underlying protocols used to enforce the required object properties. Thus, the programmer specifies multi-object constraints in an abstract and high-level manner which is independent of the details involved in explicit message passing. Furthermore, the implementation of synchronisers can involve direct communication between the constrained objects or indirect communication with a central coordinator process. Synchronisers are not accessed directly by message passing but indirectly through pattern matching. The following code defines a synchroniser that enforces collective bound on allocated resources.

```
AllocationPolicy(adm1,adm2,max)
{
 init prev:=0;

 prev >= max disables (adm1.request or adm2.request),
 (adm1.request or adm2.request) updates prev:=prev+1,
 (adm1.release or adm2.release) updates prev:=prev-1
}
```

The above synchroniser has a local constraint (`prev >= max`) that prevents (by means of using the keyword `disable`) allocation of more resources than the system provides (by disabling the invocation of the `request` method). Furthermore, upon encountering an invocation pattern, the local variable `prev` is updated accordingly. The next example shows how the coordination code for the five philosophers can be encapsulated into a synchroniser.

```
PickUpConstraint(c1,c2,phil)
{
 atomic( (c1.pick(sender) where sender=phil),
         (c2.pick(sender) where sender=phil)  ),
 (c1.pick where sender=phil) stops
}
```

The synchroniser is parameterised with the two chopsticks (`c1` and `c2`) and the philosopher who accesses the chopsticks (`phil`). Furthermore, the synchroniser applies only to `pick` messages sent by `phil`. We assume the existence of an `eat` method which invokes concurrently `pick` on each of the needed chopsticks. The synchroniser enforces atomic access to the two chopsticks; when `phil` has successfully acquired both chopsticks, the constraint is terminated.

3.1.14 Compositional Programming

The concept of *compositionality* ([25]), an important design principle for task parallel programs, shares the same goals with coordination, namely reusability of sequential code, generality, heterogeneity, and portability; as such it can be seen as a coordination model. A compositional programming system is one in which properties of program components are preserved when those components are composed in parallel with other program components. Thus, it is possible to define in a compositional way recurring patterns of parallel computation, whether configuration ones (such as mapping techniques) or communication ones (such as streamers and mergers), as building blocks and combine them together to form bigger programs. If desired, the compositional assembly preserves the deterministic behaviour of its constituent parts, thus simplifying program development by allowing program components to be constructed and tested in isolation from the rest of their environment.

There are basically two approaches to deriving compositional programs. The first approach is based on *concurrent logic programming* and is exemplified by languages such as Strand, the Program Composition Notation (PCN), Fortran-M and Compositional C++ ([37]). Concurrent logic programming offers a powerful computational model for parallel computing and over the years a number of techniques have been developed for expressing useful coordination patterns. In the case of Strand, the language is used to express the coordination aspects of some parallel program, whereas the actual computation code is written in some other more suitable language, typically C or Fortran. The following code implements a genetic sequence alignment algorithm.

```
align_chunk(Sequences,Alignment) :-
   pins(Chunks,BestPin),
   divide(Sequences,BestPin,Alignment).

pins(Chunk,CpList) :-
   cps(Chunk,CpList),
   c_form_pins(CpList,PinList),
   best_pin(Chunk,PinList,BestPin).

cps([Seq|Sequences],CpList) :-
   CpList := [CPs|CpList1],
   c_critical_points(Seq,CPs),
   cps(Sequences,CpList1).
cps([],CpList) :- CpList := [].

divide(Seqs,Pin,Alignment) :-
   Pin =\= []  |  split(Seqs,Pin,Left,Right,Rest),
                  align_chunk(Left,LAlign) @ random,
                  align_chunk(Right,RAlign) @ random,
                  align_chunk(Rest,RestAlign) @ random,
                  combine(LAlign,RAlign,RestAlign,Alignment).
divide(Seqs,[],Alignment) :-
   c_basic_align(Seqs,Alignment).
```

In the above program, the coordination/communication component is quite separate from the computational one. The first component is expressed in Strand itself; in fact, the actual details are taken care of by the underlying concurrent logic model using standard techniques such as shared single assignment variables, dependent AND-parallelism, list composition and, where appropriate, guarded clauses. The second component, the three procedures with the `c_` prefix, is written in C. Note that different mapping techniques can be explored using the `@` notation (in this case `random` specifies that the indicated procedure calls will be executed on randomly selected processors).

Whereas Strand is a concrete language (and thus it needs a dedicated WAM-based implementation to run), the Program Composition Notation (PCN) is more like a set of notations adhering to the concurrent logic paradigm (and thus PCN can be implemented as an extension of the host language(s) to be used). The above program can be written in PCN as follows.

```
align_chunk(sequences,alignment)
{  ||
    pins(chunks,bestpin),
    divide(sequences,bestpin,Alignment)
}

pins(chunk,cplist)
{  ||
    cps(chunk,cplist),
    c_form_pins(cplist,pinlist),
    best_pin(chunk,pinlist,bestpin)
}

cps(sequences,cplist)
{  ?  sequences ?= [seq|sequences1] ->
      {  ||
          cplist = [cps|cplist1],
          c_critical_points(seq,cps),
          cps(sequences1,cplist1)
      },
      sequences ?= [] -> cplist = []
}

divide(seqs,pin,alignment)
{  ?  pin != [] ->
      {  ||
          split(seqs,pin,left,right,rest),
          align_chunk(left,lalign) @ node(random),
          align_chunk(right,ralign) @ node(random),
          align_chunk(rest,restalign) @ node(random),
          combine(lalign,ralign,restalign,alignment)
      },
      pin == [] -> c_basic_align(seqs,alignment)
}
```

The reader may recall the use of logic programming in expressing coordination laws in Law Governed Linda (section 3.1.4). There, logic programming is interfaced to some other coordination formalism (namely the tuple space). Here, however, (concurrent) logic programming is itself the coordination formalism used.

The second approach to deriving compositional programs originates from *functional programming* and is expressed in the form of *skeletons*. Skeletons ([32,67]) are higher order functional forms with built-in parallel behaviour. They can be used to abstract away from all aspects of a program's behaviour such as data partitioning, placement and communication. Skeletons are naturally data parallel and inherit all the desirable properties of the functional paradigm such as abstraction, modularity, and transformation. The latter capability allows a skeletons-based program to be transformed to another, more efficient one, while at the same time preserving the properties of the original version. Thus, all analysis and optimisation can be confined to the functional coordination level which is more suitable for this purpose. Furthermore, one is able to reason about the correctness of the programs produced or derived after transformations. Skeletons can be both *configuration* and *computational* ones and, being independent of the host computational language, they can be combined with C, Fortran, etc. The following is a configuration skeleton.

```
distribution (f,p) (g,q) A B = align (p ° partition f A)
                                     (q ° partition g B)
```

distribution takes two function pairs; f and g specify the required partitioning strategy of A and B, respectively, and p and q specify any initial data rearrangement that may be required. partition divides a sequential array into a parallel array composed of sequential subarrays. align pairs corresponding subarrays in two distributed arrays together, to form a new configuration which is an array of tuples. A specialised partition for a 1 x m two dimensional array using row_block can be defined as follows.

```
partition (row_block p) A = << ii := B | ii <- [1..p] >>
   where B  =  SeqArray (1:1/p,1:n)
               [ (i,j) := A (i+(ii-1)*l/p,j) | i <- [1..l/p], j <- [1..n] ]
```

A computational skeleton for a matrix addition performed in parallel using the configuration skeleton above, can be defined as follows:

```
matrixAdd A B = (gather ° map ° SEQ_ADD) (distribution fl dl)
   where C = SeqArray ((1..SIZE(A,1)), (1:SIZE(A,2)))
         fl = [((row_block p),id), ((row_block p),id), ((row_block p),id)]
         dl = [A,B,C]
```

Note that SEQ_ADD is defined in some other computational language. There is no unique set of skeletons and a number of them have been designed with some special purpose in mind ([14,18,33,34]).

3.1.15 CoLa

CoLa ([44]) is particularly suited for Distributed Artificial Intelligence applications implemented using massive parallelism. It is effectively a set of primitives (quite independent of the host programming language) that introduce and enforce a number of desired properties such as high-level communication abstraction (*correspondents*), virtual communication topologies, and a *local view* of computation for each process. In particular, associated with each process is a *Range of Vision* which defines the set of correspondents the process can locally communicate with, plus a *Point of View* which indicates a specific communication topology the process is involved in. The following program models bi-directional communication in a tree topology.

```
with csTopoVision                                    -- CoLa base topology class
class csTreeVision is                                   -- Define Point of View
   father(csCor, const csCor);                  -- father node in Point of View
   son(csCor, const csCor);                        -- son node in Point of View
end class;

implementation csTreeVision is    -- Implementation of the Point of Views
   son is rule son(X,Y) :- csTopoVision.isLinked(X,Y).
   father is rule father(X,Y) :- son(Y,X).           -- Prolog like clauses
end implementation;

procedure p(T: in csTreeVision) is
   F: csSet := {X in T | father(X,self)};          -- Compute correspondence
   S: csSet := {X in T | son(X,self)};
   myMsgDep := csMsgSendAssDep(highest_prio(S),T,csREAD);
   myMsgId  := csMsgAss(myMsgBody,myMsgDep,csFIFO);
   csMsgSend(myMsgId);                             -- Send in the tree topology
   M := {};                                                      -- Enter loop
   csLoop do                                      -- Read all messages
      myMsgDep    := csMsgRecvAssDep(C in S, T);
      myMsgRecvId := csMsgAss(myMsgBody,myMsgDep,csIMMEDIATE);
      csMsgRecv(myMsgRecvId);
      M := union(M,{C});
      exit if csIsSmallSubset(M,S);
   end csLoop
   myMsgDep := csMsgSendAssDep(F,T,csREAD);           -- Build depiction
   myMsgId  := csMsgAss(myMsgBody,myMsgDep,csCAUSAL);
   csMsgSend(myMsgId);              -- Send results upward to father node
end procedure
```

A process sends information to a set of other processes (computed by the relation `son` and the filter `highest_prio`) and expects a reply from some, but not necessarily all of its addressed processes. It then sends some computed results upwards to its father. The program uses a predefined topology (`csTopoVision`) whose specification and implementation are also given. On setup of the procedure `p`, it first computes its father and children correspondents, using the appropriate Points of View supplied as parameters. It then constructs the message `Depiction` for the destination processes and finally sends a message to all its children selected by the filter `highest_prio`. To receive an answer, the program enters a loop and specifies from whom in the topology it is ready to process a message, and then forwards the results to the father node. Note the declarative Prolog-like style for implementing Points of View. Note also the use of predefined communication view primitives (such as `csMsgSendAssDep`) throughout the code.

3.1.16 Opus

Opus ([19]) is effectively a coordination superlanguage on top of High Performance Fortran (HPF) for which it was designed, for the purpose of coordinating concurrent execution of several data-parallel components. Interaction between concurrently executing tasks is achieved via the *ShareD Abstraction (SDA)*, a Linda-like common forum. An SDA is in fact an Abstract Data Type containing a set of data structures that define its state and a set of methods for manipulating this state. SDAs can be used either as traditional ADTs that act as data servers between concurrently executing tasks or as computation servers driven by a main, controlling task. In that respect, Opus combines data- and task-parallelism.

Execution of an Opus program begins with a single coordinating task that establishes all the participating computation and data servers. The coordinating task drives the computation by invoking the proper methods within the computation SDAs. Communication and synchronisation between the concurrently executing tasks is managed by the data SDAs. The following code implements a data server for a FIFO bounded buffer.

```
SDA TYPE buffer_type(size)
   INTEGER              ::size
   REAL, PRIVATE        ::fifo(0:size-1) ! FIFO buffer
   INTEGER, READ ONLY   ::count=0        ! number of full elements in fifo
   INTEGER, PRIVATE     ::px=0           ! producer index
   INTEGER, PRIVATE     ::cx=0           ! consumer index
     …
   CONTAINS
      SUBROUTINE put(x) WHEN (count .LT. size)
         REAL, INTENT(IN) ::x
         fifo(px)=x                  ! put x into first empty buffer element
         px=MOD(px+1,size)
         count=count+1
      END
      SUBROUTINE get(x) WHEN (count .GT. 0)
         REAL, INTENT(OUT) ::x
         x=fifo(cx)               ! get next element from full buffer
         cx=MOD(cx+1,size)
         count=count-1
      END
        …
END buffer_type
```

SDAs of the above type are created and activated as follows:

```
PROCESSORS R(128)

SDA(buffer_type)::buffer1, buffer2
…
CALL buffer1%CREATE(256) on PROCESSORS R(1)
CALL buffer2%CREATE(000000,STAT=create_status) ON PROCESSORS R
```

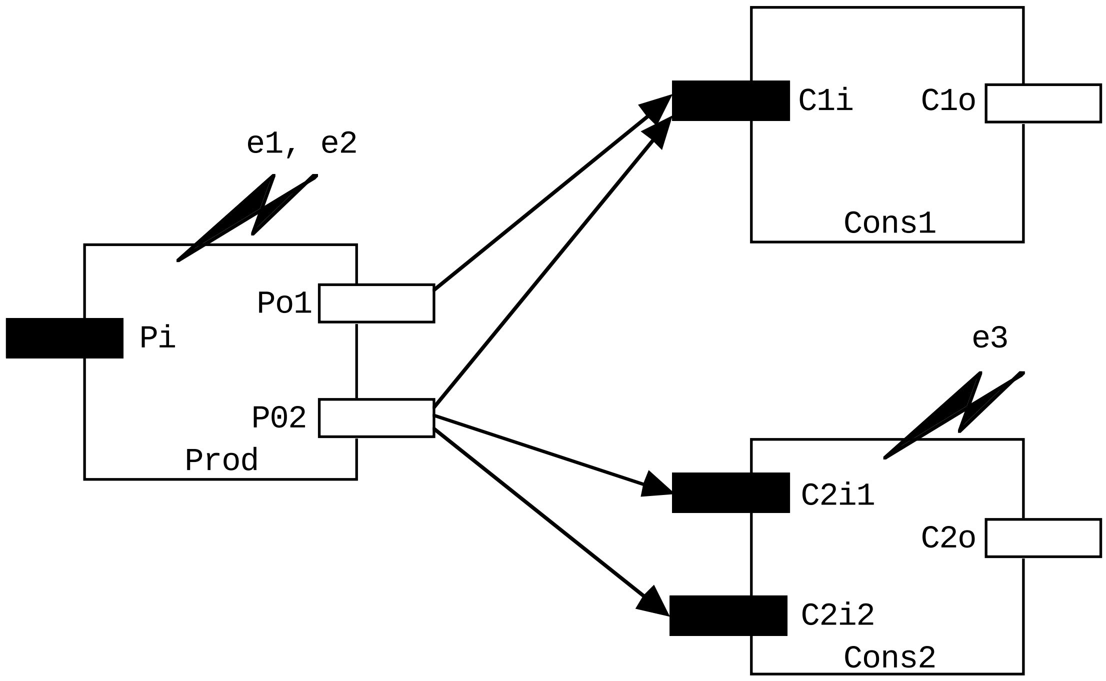
The first `CREATE` statement generates an SDA with buffer size 256 and allocates it on processor `R(1)` with the variable `buffer1` playing the role of a handle. The second `CREATE` statement allocates a much bigger buffer size across the rest of the processors with `buffer2` as the handle.

Opus is in fact one of the few languages in the data-driven category which separates quite clearly the coordination component from the HPF computational component. However, we choose to include the model in this category rather than the one on control-driven coordination languages because of the SDA mechanism it employs which is quite remoniscent to the Shared Dataspace one.

3.2 Control-Driven (Process-Oriented) Coordination Models

In control-driven or process-oriented coordination languages, the coordinated framework evolves by means of observing state changes in processes and, possibly, broadcast of events. Contrary to the case of the data-driven family where coordinators directly handle and examine data values, here processes (whether coordination or computational ones) 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 previous family. In addition to using ports, processes often send out to their environment *control messages* or *events* with the purpose of letting other interested processes know in which *state* they are or informing them of any *state changes*. The diagram below depicts these concepts.

In particular, the figure shows a configuration involving one producer with one input and two output ports and two consumers, one with a single input port and a single output port and the other with two input ports and one output port. Stream connections have been established between the output ports of the producer and the input ports of the consumers, sometimes with more than one stream entering an input port or leaving an output port. Furthermore, the producer and one of the consumers either raises and/or observes the presence of some events. Most of the coordination languages to be described in this section realise in one way or another the above CSP- or Occam-like formalism. However, they differentiate in the exact functionality of the involved concepts. For instance, in some languages events can be parametric with types and data values (effectively another mechanism for interprocess communication) whereas in other languages events are strictly simple units signifying state changes. Furthermore, in some languages events are broadcast by means of mechanisms different from stream connections whereas in other languages events actually travel through streams. Stream connections themselves can be realised in a number of ways; for instance, they may or may not support synchronous communication. In some cases streams are interfaced to some common medium (such as a “data bus”) rather than being point-to-point connections between ports; even in this latter case however, the medium is not used for unrestricted broadcasting. Also, some languages support dynamic creation of ports and exporting of their id for use by other processes whereas others limit such functionality.



The ability to visualise the evolution of computation in this family of coordination models using metaphors similar to the ones shown in the above figure is not irrelevant to the fact that for many of these coordination languages graphical programming environments exist ([16]).

3.2.1 PCL (Proteus Configuration Language)

PCL ([68]) is a language designed to model architectures of multiple versions of computer-based systems. Furthermore, it has been used to model static as well as dynamic configurations. Coordination in PCL is understood as a *configuration*; the unit of configuration is a family entity, representing one or more versions of a logical component or system. A family entity may be related to other family entities through inheritance, composition, and relationship participation. A family entity has various kinds of associated information, namely a composition structure, a classification (specifying its type), a list of attributes, a parts section specifying the composition of the entity in terms of other entities, and a number of version descriptors.

In the configuration paradigm, an application is presented as a co-operating set of components. A component *encapsulates* state and may provide and require services, to and from other components. There exist *single* and *composite* components, the latter merely providing abstraction mechanisms since at run-time the system unfolds to one comprising only simple components realised as processes. Simple components may be classified as *active* (that provide services to other components, but are also able to execute in the absence of external stimuli) and *passive* (acting only when some external stimulus requests a service from them). Another major element of the configuration paradigm is the *ports* which are used to represent either provided or required service. A component may have a number of required and/or provided ports. Inter-component communication is facilitated indirectly by transmitting messages through bindings, where a *binding* is used to connect two ports. Communication can either be synchronous or asynchronous.

The following example models a scenario involving a `log` component, a `controller` component and three `sensor` components. `log` provides two services: an `add` service allowing a `sensor` component to register a value and a `readings` service allowing the `controller` component to read the last n `sensor` readings registered. Only the code for `log` is shown below.

```
family log inherits component
 class
  type => active
 end
 attributes
  persistentState=true
 end
 interface
  provides => (add,readings)
 end
 structure
  portBindLeastOnce => add, (sensor)
  portBindExactlyOnce => readings, (controller)
 end
 behaviour
  intraAtomicOperation => (writeToDisk)
 end
end
```

`log` is defined as an active component which periodically writes data to the disk; meanwhile, this component may not be substituted with another one. Furthermore, `log` supports persistent state in the sense that during dynamic reconfiguration, persistent information should survive.

The above mentioned framework inherently supports a clear distinction between the configuration component (namely PCL) and what is being configured (i.e., computational components written in any conventional programming language). Furthermore, components are context independent since inter-component interaction and communication is achieved only by means of indirect interfaces comprising ports connected by means of bindings. Thus, a separation is achieved between the functional description of individual component behaviours and a global view of the formed system as a set of processes with interconnections. In addition, port connections are effectively unlimited buffers. If component replacement is to take place, any outstanding messages not yet delivered to a to-be-replaced component are retained by the run-time system and eventually forwarded to the component's replacement.

Finally, note that PCL is object-oriented and polymorphic. Thus, inheritance can be exploited to build hierarchies of family entities (in the example above, for instance, `log` inherits the basic functionality of a component). In addition, it is easy to create reusable descriptions of configuration scenarios.

3.2.2 Conic

Conic ([47]) is another language where coordination is viewed as configuration. In fact, Conic is two languages: a programming language which is a variant of Pascal enhanced with message passing primitives, plus a configuration language, similar in nature to PCL, featuring logical nodes that are configured together by means of links established among their input/output ports. A *logical node* is a system configuration unit comprising sets of tasks which execute concurrently within a shared address space. Configured systems are constructed as sets of interconnected logical nodes; these sets are referred to as *groups*.

The programming subcomponent of Conic is based on the notion of *task module types*, which are self-contained, sequential tasks; these are used at run-time by the Conic system to generate respective module instances, which exchange messages and perform various activities. The modules' interface is defined in terms of strongly typed ports. An *exitport* denotes the interface at which message transactions can be initiated and provide a local name and type holder in place of the source name and type. An *entryport* denotes the interface at which message transactions can be received and provides a local name and typeholder in place of the source name and type. A link between an exitport and an entryport is realised by means of invoking the message passing facilities of the programming subcomponent. The system supports both unidirectional asynchronous and bi-directional synchronous communication. Since all references are to local objects, there is no direct naming of other modules or communication entities. Thus each programming module is oblivious to its environment, which renders it highly reusable, simplifies reconfiguration, and clearly separates the activities related to the latter from purely programming concerns.

The following example illustrates the syntax of the configuration subcomponent of Conic; it is part of a typical (regarding configuration languages) example describing a patient monitoring system comprising nurses and patients.

```
group module patient;
 use monmsg: bedtype, alarmstype;
 exitport alarm: alarmstype;
 entryport bed: signaltype reply bedtype;
 << code >>
end.

group module nurse;
 use monmsg: bedtype, alarmstype;
 entryport alarm[1..maxbed]: alarmstype;
 exitport bed[1..maxbed]: signaltype reply bedtype;
 << code >>
end.
```

The first module models the monitoring device of a patient. The device periodically reads sensors attached to the patient. If any readings are detected which are outside their established ranges, suitable alarm messages are sent to the exitport. Also, a request message received in the entryport, returns the current readings. The second module displays alarms received at the entryports and can request the display of the readings associated with a patient from its exitports. Note that a nurse may be monitoring more than one patient (hence the use of arrays of ports).

The following configuration code creates instances of the above modules (at specified machine locations in the system) and establishes the required communication links.

```
system ward;
 create
  bed1: patient at machine1;
  nurse: nurse at machine2;
 link
  bed1:alarm to nurse.alarm[1];
  nurse.bed[1] to bed[1].bed;
end.
```

Conic supports a limited form of *dynamic reconfiguration*. First of all, the set of task and group types from which a logical node type is constructed is fixed at node compile time. Furthermore, the number of task and group instances within a node is fixed at the time a node is created. Dynamic changes to link set-ups can be achieved through the `unlink` command. The following example shows how the above system can evolve at run-time to one where the nurse module instance changes behaviour and starts monitoring the readings of another patient (such a scenario would make sense if the nurse already monitors the maximum number of patients it can handle).

```
manage ward;
 create
  bed2: patient at machine1;
  unlink
    bed1:alarm from nurse.alarm[1];
    nurse.bed[1] from bed[1].bed;
  link
    bed2:alarm to nurse.alarm[1];
    nurse.bed[1] to bed[2].bed;
end.
```

Another limitation of the dynamic reconfiguration functionality of Conic is related to the very nature of the links that are being established between entryports and exitports. In particular, these links are not viewed as (unbounded) buffer areas. Thus, when some link is severed between a pair or set of ports, the module instances involved in communication must have stopped exchanging messages, otherwise information may be lost and inconsistent states may result. The Conic developers have designed a system reconfiguration model whereby links may be severed only between nodes which enjoy a *quiescence* property. More to the point, a node is quiescent if: (i) it is not currently involved in a transaction that it initiated, (ii) it will not initiate new transactions, (iii) it is not currently engaged in servicing a transaction, and (iv) no transactions have been or will be initiated by other nodes which require service from this node. Finally, one cannot underestimate the fact that in Conic a user is constrained by using a single programming language (the Pascal like Conic programming subcomponent).

3.2.3 Darwin/Regis

The Regis system and the associated configuration language Darwin ([48]) are effectively an evolution of the above mentioned Conic model. Darwin generalises Conic by being largely independent of the language used to program processes (although the Regis system actually uses a specific language, namely C++). Furthermore, Darwin realises a stronger notion of dynamic reconfiguration: it supports (lazy) component instantiation and (direct) dynamic component instantiation (in contrast, Conic supports only static configuration patterns which can be changed at run-time only by explicitly invoking a configuration manager). Furthermore, it allows components to interact through user defined communication primitives (whereas Conic offers only a predefined set of such primitives). Finally, there is a clear separation of communication from computation (in Conic, the computation code is intermixed with the communication code).

As in Conic, a Darwin configuration comprises a set of *components* with clearly defined interfaces which are realised as *ports*, the latter being queues of typed messages. Ports are of two types: those used by a process to receive data (effectively input ports) are understood as being *provided* to the environment of the process for the benefit of other processes to post the messages, and those used to send data (effectively output ports) are understood as *requiring* a port reference to some remote port in order to post there the involved data. In fact, ports in Darwin/Regis are viewed as the more general concept of *services* (either provided or required); combined with port references, this allows the realisation of rather liberal notions of port connections. In addition to the provided/required ports included in a process definition, processes may at run-time realise various communication and configuration patterns by exchanging port references and then using them to send/receive messages.

The following example calculates in a distributed fashion the number π. It consists of three groups of processes: a set of worker processes dividing among themselves the computational work to be done, a supervisor process which combines the results, and a top-level coordinator process which sets up the whole apparatus. First we show the configuration component of the example written in Darwin.

```
component supervisor (int w)
{
  provide
    result <port,double>;
  require
    labour <component,int,int,int>;
}

component worker (int id, int nw, int intervals)
{
 require
   result <port,double>;
}

component calcpi2(int nw)
{
 inst
   S:supervisor(nw);
 bind
   worker.result -- S.result;
   S.labour -- dyn worker;
}
```

The `supervisor` process is responsible for dynamically spawning new `worker` processes. Note that the **`require`** part of `supervisor` specifies as the required type of service a component rather than merely a port. The coordinator process `calcpi2` generates an instance of `supervisor`; furthermore, it dynamically generates instances of `labour` (by means of the **`dyn`** primitive which in this case creates instances of `worker` when invoked) and sets up the port connections accordingly.

The criteria that dictate precisely when new `worker` processes are spawned, as well as the actual computation code form the Regis C++ based computation subcomponent are shown below.

```
worker::worker(int id, int nw, int intervals)
{
 double area=0.0;
 double width=1.0/intervals;
 for (int i=id; i<intervals; i+=nw)
   {
    double x=(i+0.5)*width;
    area+=width*(4.0/(1.0+x*x));
   }
 result.send(area);
 exit();
}

supervisor::supervisor(int nw)
{
 const int intervals=400000;
 double area=0.0;
 for (int i=0; i<nw; i++)
   {
    labour.at(i);
    labour.inst(i,nw,intervals);
   }
 for (int i=0; i<nw; i++)
   {
    double tmp;
    result.in(tmp);
    area+=tmp;
   }
 printf(“Approx pi %20.15lf\n”,area);
}
```

Note the use of the communication primitives `send` and `in` which are used to post and retrieve, respectively, a message to/from a port. Furthermore, note the expression `labour.inst(i,nw,intervals)` which actually invokes a new `worker` process (the `at` primitive in the previous command line is used to specify on which processor the new process should run).

3.2.4 Durra

Durra ([11,12]) is yet another architecture configuration language. A Durra application consists of a set of *components* and a set of *configurations* specifying how the components are interrelated. Components consist of application *tasks*, which feature *input/output ports*, and communication *channels*. At run-time, tasks create *processes* and channels create *links*; composite process configurations are achieved by using links to connect input/output ports between different processes.

Durra's main concern is how to coordinate resources, i.e., load and execute programs at different locations (thus supporting heterogeneous processing), route data, reconfigure the application, etc. As all the other members in this family of coordination languages, it makes a clear distinction between application structure and behaviour. Tasks implement the functionality of the application whereas channels implement communication facilities. Thus, it is possible to support different kinds of communication; furthermore, reusability of components is enhanced.

The following example shows how one can realise a producer-consumer scenario in Durra. In particular, it presents the definitions for a producer task, a consumer task and a FIFO channel.

```
task producer                           task consumer
 ports                                   ports
  output: out message;                    input: in message;
 attributes                              attributes
  processor="sun4";                       processor="sun4";
  procedure_name="producer";              procedure_name="consumer";
  library="/usr/durra/srclib";            library="/usr/durra/srclib";
end producer;                           end consumer;
```

```
channel fifo(msg_type:identifier, buffer_size:integer)
 ports
  input: in msg_type;
  output: out msg_type;
 attributes
  processor="sun4";
  bound=buffer_size;
  package_name="fifo_channel";
  library="/usr/durra/channels";
end fifo;
```

The above piece of code defines a `producer` task with an output port of type message. Furthermore, the Durra code specifies that task instances of `producer` should run on the indicated machine, the task's implementation code can be found in the procedure `producer` in the directory `/usr/durra/srclib`. Similar things can be said about the task `consumer` and the channel `fifo` (the description of the latter also generically specifies through parameters, the types of messages its two ports can receive and other relevant information, such as the size of the channel). The actual implementation of the above tasks and channel are not shown here; they can be written in any conventional programming language.

The following Durra code uses the above defined entities to generate a compound task description featuring dynamic reconfiguration.

```
task dynamic_producer_consumer
 components
```

```
    p: task producer;
    c[1..2]: task consumer;
    buffer: channel fifo(message,10);
  structures
    L1: begin
          baseline p, c[1], buffer;
          buffer: p.output >> c[1].input;
        end L1;
    L2: begin
          baseline p, c[2], buffer;
          buffer: p.output >> c[2].input;
        end L2;
  reconfigurations
    enter => L1;
    L1 => L2 when signal(c[1],1);
  clusters
    cl1: p, buffer;
    cl2: c[1], c[2];
end dynamic_producer_consumer;
```

The scenario involves one producer, two consumers and a FIFO channel of buffer size 10. Two different configuration scenarios are possible: `L1` involving the producer, the first consumer and the channel, and `L2`, of similar in nature to `L1`, where the second consumer is used instead. In either case, the producer sends data via its output port to the input port of the first or the second consumer through the channel. Initially, the configuration `L1` is active; transition to `L2` is done when some particular signal is raised by the first consumer.

Durra is tailored more to support rapid prototyping of distributed heterogeneous applications and test different configuration strategies, rather than as a means to actually implement these applications. Its task emulator supports a number of useful features including timing constraints (thus, rendering the language suitable for real-time applications) but its implementation is centralised. Furthermore, although in principle any implementation language can be used, the Durra system is tailored towards the use of Ada. Finally, unrestricted dynamic creation of task instances is not possible; for instance, the above code restricts the involved entities at run-time to be one producer, two consumers and a channel.

3.2.5 CSDL

CSDL (Cooperative Systems Design Language, [58,59]) is a specification and design language that supports the definition of the coordination aspects, and the definition of the logical architecture of a cooperative system. A CSDL configuration comprises users, applications and *coordinators* where the latter define the cooperation policies and control the data flowing between users and shared applications. A coordinator is composed of three parts: a *specification* that defines *groups* and cooperation policies in terms of *requests* exported selectively to members of different groups; a *body* that defines the *access rights* associated with the groups in terms of communication system control; and a *context* that defines coordinator dependencies in terms of groups mapping. The following CSDL code defines the specification and body of an X-Windows coordinator.

```
coordinator XWindow
{
  group ConnectedUsers;
  group Output
    nestedIn ConnectedUsers;
  group Input
    nestedIn Output;
  invariant #Input <= 1;
  requests
  {
   exportedTo extern
   {
    join Output other
    {
     actions: insert ConnectedUsers other;
              insert Output other;
    }
    join Input other
    {
     requires: other in Output and #Input = 0;
     actions: insert Input other;
    }
   }
    leave Output other
    {
     actions: extract Output other;
              extract ConnectedUsers other;
    }
    leave Input other
    {
     actions: extract Input other;
    }
  }

}

coordinator body XWindow
{
 S: switcher inOut XSwitcher;
 group ConnectedUsers
 { connected; inOff; outOff; }
 group Output
 { outOn; }
 group Input
 { inOn; }
}
```

The specification includes declaration of group identifiers that may involve the definition of a *type* and of a *nesting*. It also includes an invariant stating some constraints on group cardinality and membership through logical expressions, and a set of requests (such as **`join`** or **`leave`**) exported selectively to members of groups according to a desired policy. For instance, **`exportedTo extern`** refers to any sender not belonging to the group of the coordinator. Exchange of information between components is done by means of *virtual switches*, defined within the body of a coordinator, that model multiplexing and demultiplexing of data streams. Declaration of switches is accompanied by kinds and modes of access; in the example above, members of the `ConnectedUsers` group can be connected but cannot send and/or receive data since they have both their input and their output channels disabled.

3.2.6 POLYLITH

POLYLITH ([26,60]) is a software interconnection system, effectively a MIL enhanced with functionality (such as input/output ports and, more recently, events) found usually in coordination languages. POLYLITH clearly separates functional requirements from interfacing requirements, thus enhancing decoupling and reuse of software components. A component is treated as a *module*; modules have interfaces for each communication channel upon which the running instances of a module (i.e., processes) will send or receive messages. An abstract decoupling agent, called the *software bus*, is used as a means for process communication;

message passing routines provided by the system allows processes to get “plugged” to or “unplugged” from the bus.

The program code for a module is written separately from the rest of the code describing how it interfaces with the rest of the system and, in fact, the language supports the mixed language approach. Furthermore, the software bus actually encapsulates separately the interfacing decisions of the involved modules. Thus, it is possible to use the same set of modules with different buses, say, one for distributed systems based on the TCP/IP paradigm, another one tailored to use shared memory, etc.

The following code shows the outline of the implementation and the specification of two modules.

```
main(argc,argv)
/* a.c (exec in a.out) */
{
 char str[80];
 …
 mh_write(“out”,…,”msg1”);
 …
 mh_read(“in”,…,str);
 …
}
```

```
main(argc,argv)
/* b.c (exec in b.out) */
{
 char str[80];
 …
 mh_read(“in”,…,str);
 …
 mh_write(“out”,…,”msg2”);
 …
}
```

```
service “A”:
{
 implementation: {binary: “a.out”}
 source “out”: {string}
 sink “in”: {string}
}

service “B”:
{
 implementation: {binary: “b.out”}
 source “out”: {string}
 sink “in”: {string}
}
```

```
orchestrate “example”:
{
 tool “foo”: “A”
 tool “bar”: “B”
 tool “bartoo”: “B”
 bind “foo out” “bar in”
 bind “bar out” “bartoo in”
 bind “bartoo out” “foo in”
}
```

In the implementation part and using the primitives `mh_read` and `mh_write`, each of the two modules sends to its output channel `out` the messages `msg1` and `msg2`, respectively, and receives a message of type `string` from its own input channel `in` into its local variable `msg`. In the specification part, two services `A` and `B` are defined of type `a.out` and `b.out`, respectively, and furthermore, it is specified that each of them has an outgoing interface `out` and an incoming interface `in`, both of type `string`. The application definition `example` (effectively the “coordinator” process) actually creates a specific scenario involving one instance of `a.out` and two instances of `b.out` and properly connects their respective input/output channels.

Recently, the model has been enhanced with *events* allowing event-based interaction: modules register their interest in observing the raising of an event, at which point they invoke a procedure associated with the event. Furthermore, “event coordinators” are used to match together events of the same functionality but with different names among a number of modules realising a composite interaction. The following code illustrates some of these points.

```
module “A”:
{
 declare Sig1 {integer,string};
 generate Sig1;
 when Sig2 => Proc1;
```

```
module “B”:
{
 declare Sig2 {integer};
 generate Sig2;
 when Sig1 => Proc2;
}

main()
{
 char *event_type, *event;

 /* initialisation */
 Init(argc,argv,NULL,NULL);

 /* events declaration */
 DeclareEvent(“IS”,”Sig1”);

 /* register interest */
 RegisterEvent(“Sig2”);

 while (true)
 {
  /* get next event */
  GetNextEvent(event_type,event);

  /* invoke corresp proc */
  if (strcmp(“Sig2”,event_type)==0)
     Proc1(event);
 }
}
```

```
}

main()
{
 char *event_type, *event;

 /* initialisation */
 Init(argc,argv,NULL,NULL);

 /* events declaration */
 DeclareEvent(“I”,”Sig2”);

 /* register interest */
 RegisterEvent(“Sig1”);

 while (true)
 {
  /* get next event */
  GetNextEvent(event_type,event);

  /* invoke corresp proc */
  if (strcmp(“Sig1”,event_type)==0)
     Proc2(event);
 }
}
```

Each module declares an event and registers its interest in observing the raising of some other event. Upon detecting the presence of the specified event, the module calls some procedure. The first part of the code specifies the intended interaction, while the second part presents the outline of the implementation using C. Note that events are actually parameterised with data, so in fact they substitute the use of input/output channels in the previous version of POLYLITH.

3.2.7 The Programmer’s Playground

The Programmer’s Playground ([42]) shares many of the aims of languages such as Conic, Darwin and Durra, in that it is a software library and run-time system supporting dynamic reconfiguration of distributed components. Furthermore, the model supports a uniform treatment of discrete and continuous data types and, as in the case of the other models in this family, a clear separation of communication from computation concerns.

The Programmer’s Playground is based on the notion of *I/O abstraction*. I/O abstraction is a model of interprocess communication in which each *module* in the system has a presentation that consists of data structures that may be externally observed and/or manipulated. An *application* consists of a collection of independent modules and a *configuration of logical connections* among the data structures in the module presentations. Whenever published data structures are updated, communication occurs implicitly according to the logical connections.

I/O abstraction uses *declarative* communication (as opposed to *imperative* communication) in the sense that the user declares high-level logical connections among state components of modules (rather than expressing direct communication within the control flow of the program). Declarative communication enhances the separation of communication from computation, is less error-prone and facilitates the automatic updating of the modules’ states in cases where changes in the state of one module should be reflected in some other module. This last functionality is achieved by means of *connections*: if an item `x` in a module `A` is connected to an item `y` in another module `B`, then any change in `x`’s value will cause an appropriate update in the value of `y`. Such connections can be *simple* (point-to-point) or *element-to-aggregate* (one-to-many); furthermore, the former can be *unidirectional* or *bi-directional*.

The following producer-consumer apparatus illustrates some of the above points.

```
#include “PG.hh”

PGint next=0;
PGstring mess;

send_next(PGstring mess, static int i)
{
 if (strcmp(mess,”ok”)==0)
    next=i++;
}

main()
{
 PGinitialise(“producer”);
 PGpublish(next,”next_int”,READ_WORLD);
 PGpublish(mess,”ok”,WRITE_WORLD);

 while (1)
 {
  PGreact(mess,send_next);
 }
 PGterminate();
}


#include “PG.hh”

PGint next=0;
PGstring mess;

void consume_int(PGint i)
{ /* consumes list of integers */ }

main()
{
 PGinitialise(“consumer”);
 PGpublish(mess,”ok”,READ_WORLD);
 PGpublish(next,”next_int”,WRITE_WORLD);

 while (1)
 {
  PGreact(next,consume_int);
  mess=“ok”;
 }
 PGterminate();
}
```

The above program consists of two modules forming a producer-consumer pair communicating synchronously to exchange an infinite list of integers. Both modules use an integer variable used to send/receive the integers and a string variable used by the consumer to declare that it is ready to receive the next integer. These variables are published by the two modules with an external name and a protection flag (read/write). The procedure `PGreact` is used to suspend execution of the module until the variable indicated in its first argument has been updated. The logical connection between the two variables in the respective modules is assumed to have already been established (in the Programmer’s Playground this is achieved graphically: modules are shown as boxes, published variables as input/output “ports” and logical connections as lines drawn between their respective ports).

The Programmer’s Playground is based on the C/C++ language formalism and has been implemented on Sun Solaris. Although it claims to clearly separate communication from computation concerns, at least stylistically the two different types of code are intermixed within a module. The model was developed primarily for developing distributed multimedia applications and in fact it places emphasis on supporting uniform treatment of discrete and continuous data, where differences in communication requirements are handled implicitly by the run-time system. Since the nature of the data being handled is here of more importance than in most of the other coordination models comprising the control-driven category, the Programmer’s Playground could also be included in the data-driven category. We chose to place it here though, because of the well-defined input/output connections that each process possesses as well as for sharing the same application domains with configuration languages.

3.2.8 RAPIDE

RAPIDE ([66]) is an architecture definition language and in that respect shares many of the aims of languages such as Conic and Durra. It supports both component and communication abstractions as well as a separation between these two formalisms. An *architecture* in RAPIDE is an executable specification of a class of systems. It consists of *interfaces*, *connections*, and *constraints*. The interfaces specify the behaviour of components of the system, the connections define the communication between the components using only those features specified by the components’ *interfaces*, and the constraints restrict the behaviour of the interfaces and connections. RAPIDE is *event-driven*; components generate (independently from one another) and observe events. Events can be parameterised with data and types. Asynchronous communication is modelled by connections that react to events generated by components and then generate events at other components. Synchronous communication can be modelled by connections between function calls. The result of executing a RAPIDE architecture (i.e., a set of interfaces and connections) is a *poset* showing the dependencies and independencies between events.

The following producer-consumer example illustrates some of the above points.

```
type Producer(Max: Positive) is interface
   action out Send(N: Integer);
   action in Reply(N: Integer);
behavior
   Start => Send(0);
   (?X in Integer) Reply(?X) where ?X < Max => Send(?X+1);
end Producer;

type Consumer is interface
   action in Receive(N: Integer);
   action out Ack(N: Integer);
behavior
   (?X in Integer) Receive(?X) => Ack(?X);
end Consumer;

architecture ProdCons() return SomeType is
   Prod: Producer(100);
   Cons: Consumer;
connect
   (?n in Integer)
   Prod.Send(?n) => Cons.Receive(?n);
   Cons.Ack(?n) => Prod.Reply(?n);
end architecture ProdCons;
```

The above code initially declares two types of components. `Producer` is designed to accept events of type `Reply` and broadcast events of type `Send`, both parameterised with an integer value. Upon commencing execution, `Producer` broadcasts the event `Send(0)` and upon receiving an event `Reply(X)` it will reply with the event `Send(X+1)` provided that `X` is less than a certain value. `Consumer` has similar functionality. The "coordinator" `ProdCons()` creates two process instances for `Producer` and `Consumer` and furthermore it associates the output event of the former with the input event of the latter and vice versa. Note that the above code specifies how the two components interact with each other but the actual details of their implementation are left unspecified.

3.2.9 ConCoord

ConCoord ([43]) is a typical member of this family of control-driven coordination languages. A ConCoord program is a dynamic collection of computation processes and coordination processes. A computation process executes a sequential algorithm and can be written in any conventional programming language augmented with some communication primitives. Coordination processes are written in CCL, ConCoord's Coordination Language. Communication between processes is done in the usual way of sending data to output ports they own and receiving data from input ports they also own, thus effectively, achieving complete decoupling of producers from consumers. Processes raise or broadcast their state which is, in fact, an event or signal parameterised with data. States are communicated by message passing. The following example shows a dynamically evolving pipeline of generic processes.

```
coordinator <t_node, t_data> gen_dyn_pipeline
{
 inport <t_data> in;
 outport <t_data> out;
 states error(), done();

 create t_node n bind in -- n.left, n.out -- out;
 loop
 {
  choose
  {
   sel(t_node n | n.new and not n.right--)
     =>  create t_node new_n
         bind n.right -- new_n.left, new_n.out -- out;
   sel(t_node n | n.new and n.right--)
     =>  error();
  }
 }
}
```

`gen_dyn_pipeline` is a coordinator parameterised with the types of both the computation processes forming the pipeline (`t_node`) and the data being communicated (`t_data`). The pipeline of nodes communicates with the outside world by means of the `in` and `out` ports of `gen_dyn_pipeline`; namely, the first process will get data via `in` and all processes will output data via `out`. Initially, one process is created and its own ports `left` and `out` are bound to `gen_dyn_pipeline`'s `in` and `out` respectively. Then, each time a process at the end of the pipeline raises the state `new`, a new process is created and inserted in the pipeline. Whether a process raising the state `new` is actually the last one in the pipeline is determined by means of examining as to whether its port `right` is linked to some other port (in this particular configuration to the port `left` of some other process). If that is indeed the case, then instead of creating a new process `gen_dyn_pipeline` raises the state `error`.

The language features nested hierarchies of coordination domains and synchronous or asynchronous communication between components. In particular, a computational process raising a state blocks until this is treated by the coordinator process in charge of it; thus, communication appears to be synchronous from the process's point of view and asynchronous from the point of view of the coordinator process. Coordinator processes and their groups of coordinated processes are configured in a hierarchical manner with the top level of the configuration being a coordinator. Furthermore, the language enforces and encourages the building of structured programs by treating each pair of coordinator-coordinated processes as a separate domain. The coordinator of some domain is unaware of any nested subdomains and treats homogeneously computational and coordination processes within the latter. Furthermore, state change notification by some process is only visible in the domain of its supervisor coordinator process.

3.2.10 TOOLBUS

The TOOLBUS coordination architecture ([13]) is reminiscent of models such as POLYLITH, featuring a component interconnection metaphor (the toolbus) on which tools can be "plugged in". The toolbus itself consists of a number of processes which manage the tools forming the system. Although the number of tools comprising a system is static, the number of processes changes dynamically according to the intended functionality of the system. Thus, in addition to the straightforward one-to-one correspondence between tools and processes, it is also possible to have a tool controlled by a number of processes or groups of tools controlled by one process. Tools communicate implicitly via the toolbus; no direct tool-to-tool communication is allowed. A number of primitives are offered by the system realising both synchronous and asynchronous communication among the processes and between processes and tools. The TOOLBUS architecture recognises a common format for the interchanged data; thus, each tool must use an *adapter* which changes data formats accordingly. Furthermore, the intended behaviour of the system is specified by means of *T-scripts* which contain a number of definitions for processes and tools followed by a configuration statement. The following example shows how a compiler-editor cooperation can be modelled in TOOLBUS.

```
define COMPILER =
   (  rec-msg(compile,Name) . snd-eval(compiler,Name) .
      (  rec-value(compiler,error(Err),loc(Loc)) .
           snd-note(compile-error,Name,error(Err),loc(Loc))
      ) *  rec-value(compiler,Name,Res) . snd-msg(compile,Name,Res)
   ) *  delta

define EDITOR =
   subscribe(compile-error) .
   (  rec-note(compile-error,Name,error(Err),loc(Loc)) .
        snd-do(editor,store-error(Name,error(Err),loc(Loc)))
   +  rec-event(editor,next-error(Name)) .
        snd-do(editor,visit-next(Name)) . snd-ack-event(editor)
   ) *  delta

define UI =
   (  rec-event(ui,button,(compile,Name)) .
        snd-msg(compile,Name)  . rec-msg(compile,Name,Res)
                               . snd-ack-event(ui)
   ) *  delta
```

`COMPILER` receives a compilation request (from `UI`), starts the compilation, broadcasts any errors it encounters and finally sends the result back to the process that invoked it. `EDITOR` either receives a message with a compilation error and stores the compiled program and the error location for future reference or receives a next-error event from the editor and goes to a previously stored error location. Finally, `UI` is a user-interface with a compile button which when pushed causes a compile message to be sent and waits for a reply.

The following TOOLBUS primitives are used in the above program: `snd-msg` is used by a process to synchronously send a message to another process which the latter will receive by invoking `rec-msg`; `snd-note` is used by a process to asynchronously broadcast messages to other processes which the latter receive by invoking `rec-note`, and `subscribe` is used by a process to declare its interest in receiving certain asynchronous broadcasts from other processes. Furthermore, `rec-event` and `rec-value` are used by a process to receive, respectively, an event and the evaluation result from some tool, and `snd-do` is used by a process to request the evaluation of some term by a tool. Finally, '`+`' and '`.`' are the selection and sequential operators, respectively, and `delta` signifies process termination.

Using the above definitions, a number of configurations are possible to set up, namely:

```
toolbus(COMPILER,EDITOR,UI)        toolbus(COMPILER,UI)
```

where in the latter case a simpler system is configured without the ability to refer back to error locations.

The TOOLBUS enjoys formal semantics (the T-scripts can be formally analysed in terms of process algebras) and recently it has been extended with the notion of discrete time. A prototype interpreter C-based implementation exists and has been used to test the model's effectiveness in a number of applications.

3.2.11 MANIFOLD

MANIFOLD ([8]) is one of the latest developments in the evolution of control-driven or process-oriented coordination languages. As is the case in most of the other members of this family, MANIFOLD coordinators are clearly distinguished from computational processes which can be written in any conventional programming language augmented with some communication primitives. Manifolds (as MANIFOLD coordinators are called) communicate by means of *input/output ports*, connected between themselves by means of *streams*. Evolution of a MANIFOLD coordination topology is *event-driven* based on *state transitions*. More to the point, a MANIFOLD coordinator process is at any moment in time in a certain state where typically it has set up a network of coordinated processes communicating by sending and/or receiving data via stream connections established between respective input/output ports. Upon observing the *raising* of some event, the process in question breaks off the stream connections and evolves to some other predefined state where a different network of coordinated processes is set up. Note that, unlike the case with other coordination languages featuring events, MANIFOLD events are not parameterised and cannot be used to carry data — they are used purely for triggering state changes and causing the evolution of the coordinated apparatus. The following example is a bucket sorter written in MANIFOLD.

```
export manifold Sorter()
{
 event filled, flushed, finished.
 process atomsort is AtomicSorter(filled).
 stream reconnect KB input -> *.
 priority filled < finished.

 begin:
    (  activate(atomsort), input -> atomsort,
          guard(input,a_everdisconnected!empty,finished)
    ).

 finished:
    {  ignore filled.
       begin:  atomsort -> output
    }.

 filled:
    {  process merge<a,b | output> is AtomicIntMerger.
       stream KK * -> (merge.a, merge.b).
```

```
        stream KK merge -> output.

        begin:
           (  activate(merge),
              input -> Sorter -> merge.a,
              atomsort -> merge.b,
              merge -> output
           ).

        end | finished: .
     }.

  end:
     {  begin:
           (  guard(output,a_disconnected,flushed),
              terminated(void)
           ).

        flushed: halt.
     }.
}
```

The apparatus created by the above program functions more or less as follows: `Sorter` initially activates a computation process performing the actual sorting (`AtomicSorter`). This latter process which is capable of performing very fast sorting of a bucket of numbers of size *k* will raise the event `filled` once it receives the maximum number *k* of numbers to sort. Upon detecting the raising of `filled`, `Sorter` will activate a new sorting computation process as well as a merger process which is responsible for merging the output of both sorting processes into one stream. Depending on the bucket size *k* and the number of units to be sorted, an arbitrary number of sorting and merger processes may be created and linked together at run-time. Note that every process has by default an `input` and an `output` port; additionally, it may have other named ports too. The triggering process which, in fact, is also responsible for passing the units to be sorted to `Sorter` and printing out the output is shown below.

```
manifold Main
{
 auto process read is ReadFile("unsorted").
 auto process sort is Sorter.
 auto process print is printunits.

 begin:  read -> sort -> print.
}
```

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 in most 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) 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. For example, in the above program the stream connected to the input port of `Sorter` has been declared of type `KB` (keep-break) meaning that even if it is disconnected from its *arrival side* (the part actually connected to `Sorter`) it will still remain connected at the *departure side* (the part connected to the process which sends data down the stream — in our case `read`). Hence, when `read` must break connection with a filled sorter and forward the rest of the data to be sorted to a new sorting process, the data already in the stream will not be lost. Furthermore, *guards* are installed in the input and output ports of `Sorter` to make sure that all units to be sorted have either been received by `Sorter` or got printed successfully. These primitives, e.g., *guards*, 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. In contrast to this extensive repertoire of coordination constructs, MANIFOLD does not support ordinary computational entities such as data structures, variables, conditional or loop statements, etc. although syntactically sugared versions of them do exist for a programmer's convenience.

Although not shown here, manifolds can actually be parameterised; these highly reusable generic manifolds are called *manners*. MANIFOLD has been successfully ported to a number of platforms including IBM SP1/SP2, Solaris 5.2, SGI 5.3/6.3 and Linux. Furthermore, it has been used with many conventional programming languages including C, C++ and Fortran ([7]). Recently it has been extended with real-time capabilities ([54]). Its underlying coordination model IWIM ([6]), which is in fact independent of the actual language, has been shown to be applicable to other coordination models and frameworks ([55,56]).

4. Comparison

In the previous section we described in some detail the most important members of the two major families of coordination models and languages, namely the data-driven and the control-driven ones. In this section we present in a tabular form a comparison between these formalisms along some major dimensions that characterise a coordination formalism.

These dimensions are the following: (i) the entities being coordinated, (ii) the mechanism of coordination, (iii) the coordination medium or architecture, (iv) the semantics, rules of protocols of coordination employed, (v) whether a model supports a different (from the computational component) coordination language or involves the use of "add-on" primitives, (vi) whether a model supports and encourages the use of many computational languages, (vii) what is the most relevant application domain for each model, and (viii) what is the implementation status of the proposed framework.

These are by no means the only issues that differentiate one model from another. For instance, regarding the category of control-driven coordination models, an issue worth comparing is the exact nature of the port-to-port connections via streams each model employees (where this is indeed the case) and whether and how any dynamic reconfigurations are realised. For example, some models only support static configurations or restrict access to ports to their owners whereas other models support dynamic (re)configurations and exporting of ports identifiers. Although such a rather low-level comparison is useful, we felt that it would run the danger of obscuring the main differences between all the models involved across both main categories. In any case, we have outlined differences of this nature in the respective sections for each model.

5. Conclusions

The purpose of this article was to provide a comprehensive survey of those models and languages forming the family of coordination formalisms. In the process, we have classified the members of the coordination family into two broad categories, namely the data-driven and the control-driven ones. Furthermore, we have described in some depth the most prominent members in each family, highlighting their features and presenting typical examples of their use.

Most members of the first family have evolved around the notion of a Shared Dataspace which plays the dual role of being both a global data repository and an interprocess communication medium. Processes forming a computation either post and/or retrieve data from this medium. The most prominent member of this family (and, indeed, historically the first genuine coordination model) is Linda where the common medium is a tuple space and processes use it to send to or retrieve from it tuples. Linda has been used extensively and over the years a number of other similar models evolved, their main purpose being to address some of the deficiencies, weaknesses and inefficiencies of the basic vanilla model, such as issues of security, locality of reference, hierarchy of global data and optimisation in tuple access. A number of other Shared Dataspace based coordination models have been proposed, where the emphasis is on providing even more implicit (than Linda’s associative pattern matching) semantics of tuple handling such as multiset rewriting. However, not all members of this family are adhering to the Shared Dataspace concept; there are a few which use the message -passing metaphor or a limited form of Shared Dataspace in the form of common buffer areas or global synchronisation variables being manipulated concurrently by a number of processes.

Whereas the first family has been influenced by the concept of a shared medium, the second family has evolved around the Occam notion of distinct entities communicating with the outside world by means of clearly marked interfaces, namely input/output ports, connected together in some appropriate fashion by means of streams or channels. In fact, the coordination paradigm offered by this second family is sometimes characterised as being *channel-based* as opposed to the *medium-based* notion of coordination supported by the first family. Traditionally, languages of this family where initially proposed for configuring systems and modelling software architectures. However, recently a number of proposals has been put forward where control-driven languages are designed with more conventional coordination application areas in mind. Most members of this family share the concept of a separate coordination language (as opposed to the case of the data-driven models where a set of coordination primitives are used in conjunction with a host language) which is used to define pure coordination modules featuring ports, streams or channels and possibly event broadcasting. They differentiate though in issues such as whether the id of ports can become public or not, whether the communication is asynchronous or synchronous (or both), or whether events carry values.

An interesting and quite fruitful “confrontation” between the data- and control-driven coordination approaches is with respect to whether and to what extent a program need be structured and locality of communication be supported. The Shared Dataspace vanilla models, such as Linda and GAMMA, encourage a flat unstructured communication medium employing global broadcasting. However, some of their variants, such as Bauhaus Linda and Structured GAMMA, provide hierarchical levels of their communication medium which are able to express locality of communication and support structure. On the other hand, control-driven coordination languages such as MANIFOLD, support multiple port-to-port stream connections which employ limited forms of broadcasting. Furthermore, these streams are first class citizens, able to hold data within themselves while stream connections break off and get reconnected between different coordinated processes, thus providing to a certain extent the functionality of a shared communication medium. It is the authors’ belief that a number of novel coordination models and languages will be proposed which will further converge these two approaches towards the formation of communication media which will provide the desired (ideal?) degree of shared or point-to-point communication as well as support naturally the structuring of programs.

The issue of coordination is rather broad ([46,49]) and in this article we have only concentrated on the “programming languages” aspect and furthermore we have advocated a practical flavour. Thus, aspects of coordination related to, say, workflow management, cooperative work and software composition (to name but a few) have not been addressed. Neither did we dwell into theoretical issues such as semantics, formal specification and reasoning. Coordination models and languages have evolved rapidly over the past few years and the concept of coordination is now being introduced in many aspects of contemporary Computer Science including middleware domains such as the Web and CORBA-like platforms, modelling activities in Information Systems, and "coordination-in-the-large" application areas such as software engineering and open distributed systems. Thus, we expect a proliferation of many more models and languages over the years to come, addressing these issues and possibly also offering unified solutions for a number of different application domains.

ACKNOWLEDGMENTS

A number of people have provided useful feedback in improving the presentation of the survey and its contents. Special thanks are due to Anne-Alexandra Holzbacher Jensen and Andrea Omicini for detailed comments on earlier drafts of this chapter.

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.

REFERENCES

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

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

[3] J.-M. Andreoli, H. Gallaire and R. Pareschi, "Rule-Based Object Coordination", in [28], pp. 1-13.

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

[5] J.-M. Andreoli and R. Pareschi, "Linear Objects: Logical Processes with Built-In Inheritance", *New Generation Computing* **9 (3-4)**, 1991, pp. 445-473.

[6] F. Arbab, "The IWIM Model for Coordination of Concurent Activities", in [28], pp. 34-56.

[7] F. Arbab, C. L. Blom, F. J. Burger and C. T. H. Everaars, "Reusable Coordinator Modules for Massively Concurrent Applications", *Europar'96*, Lyon, France, 27-29 Aug., 1996, LNCS 1123, Springer Verlag, pp. 664-677.

[8] F. Arbab, I. Herman and P. Spilling, "An Overview of Manifold and its Implementation", *Concurrency: Practice and Experience* **5 (1)**, 1993, pp. 23-70.

[9] J.-P. Banâtre and D. Le Métayer, "GAMMA and the Chemical Reaction Model: Ten Years After", in [4], pp. 1-39.

[10] M. Banville, "Sonia: an Adaptation of Linda for Coordination of Activities in Organizations", in [28], pp. 57-74.

[11] M. R. Barbacci, C. B. Weinstock, D. L. Doubleday, M. J. Gardner and R. W. Lichota, "Durra: A Structure Description Language for Developing Distributed Applications", *Software Engineering Journal,* IEE, March 1996, pp. 83-94.

[12] M. R. Barbacci and J. M. Wing, "A Language for Distributed Applications", *International Conference on Computer Languages (ICCL'90),* New Orleans, Lui., USA, 12-15 March, 1990, IEEE Press, pp. 59-68.

[13] J. A. Bergstra and P. Klint, "The TOOLBUS Coordination Architecture", in [28], pp. 75-88.

[14] G. H. Botorog and H. Kuchen, “Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming”, *Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC-5)*, New York, USA, 6-9 Aug., 1996, IEEE Press, pp. 243-252.

[15] M. Bourgois, J-M. Andreoli and R. Parechi, “Concurrency and Communication: Choices in Implementing the Coordination Language LO”, *Object-Based Distributed Programming ECOOP’93 Workshop*, Kaiserslautern, Germany, 26-27 July, 1993, LNCS 791, Springer Verlag, pp. 73-92.

[16] P. Bouvry and F. Arbab, “Visifold: A Visual Environment for a Coordination Language”, in [28], pp. 403-406.

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

[18] H. Burkhart, R. Frank and G. Hächler, “ALWAN: A Skeleton Programming Language”, in [28], pp. 407-410.

[19] B. Chapman, M. Haines, P. Mehrotra, J. V. Rosendale and H. Zima, “Opus: A Coordination Language for Multidisciplinary Applications”, *Scientific Programming* **6 (2)**, 1997.

[20] N. Carriero and D. Gelernter, “Linda in Context”, *Communications of the ACM* **32 (4)**, 1989, pp. 444-458.

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

[22] N. Carriero, D. Gelernter and L. Zuck, “Bauhaus Linda”, in [29], pp. 66-76.

[23] S. Castellani and P. Ciancarini, “Enhancing Coordination and Modularity Mechanisms for a Language with Objects-as-Multisets”, in [28], pp. 89-106.

[24] S. Castellani, P. Ciancarini and D. Rossi, “The ShaPE of ShaDE: a Coordination System”, Technical Report UBLCS 96-5, Dipartimento di Scienze dell’Informazione, Università di Bologna, Italy, March, 1996.

[25] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison Wesley Publs., 1989.

[26] C. Chen and J. M. Purtilo, “Configuration-Level Programming of Distributed Applications Using Implicit Invocation”, *IEEE TENCON’94*, Singapore, 22-26 Aug., 1994, IEEE Press.

[27] P. Ciancarini, “Coordination Models, Languages, Architectures and Applications: a Personal Perspective”, University of Leuven, Feb., 1997, `http://www.cs.unibo.it/~cianca/coord_ToC.html`.

[28] P. Ciancarini and C. Hankin (eds), *First International Conference on Coordination Models, Languages and Applications (Coordination’96)*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag.

[29] P. Ciancarini, O. Nierstrasz and A. Yonezawa (eds), *Object-Based Models and Languages for Concurrent Systems*, Bologna, Italy, 5 July, 1994, LNCS 924, Springer Verlag.

[30] P. Ciancarini and D. Rossi, “Jada: Coordination and Communication for Java Agents”, *Second International Workshop on Mobile Object Systems: Towards the Programmable Internet (MOS’96)*, Linz, Austria, July, 1996, LNCS 1222, Springer Verlag, pp. 213-228.

[31] P. Ciancarini, R. Tolksdorf and F. Vitali, “Weaving the Web Using Coordination”, in [28], pp. 411-415.

[32] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, Pitman/MIT Press, 1989.

[33] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti and M. Vanneschi, "A Methodology for the Development and the Support of Massively Parallel Programs", *Programming Languages for Parallel Processing*, IEEE Press, pp. 205-220.

[34] J. Darlington, Y. Guo, H. W. To and J. Yang, "Functional Skeletons for Parallel Coordination", *EUROPAR'95*, Stockholm, Sweden, 23-31 Aug., 1995, LNCS 966, Springer Verlag, pp. 55-68.

[35] M. D. Feng, Y. Q. Gao and C. K. Yuen, "Distributed Linda Tuplespace Algorithms and Implementations", *Parallel Processing: CONPAR'94-VAPP VI*, Linz, Austria, 6-8 Sept., 1994, LNCS 854, Springer Verlag, pp. 581-592.

[36] G. Florijn, T. Bessamusca and D. Greefhorst, "Ariadne and HOPLa: Flexible Coordination of Collaborative Processes", in [28], pp. 197-214.

[37] I. Foster, "Compositional Parallel Programming Languages", *ACM Transactions on Programming Languages and Systems* **18 (4)**, 1996, pp. 454-476.

[38] S. Frølund and G. Agha, "A Language Framework for Multi-Object Coordination", in [29], pp. 107-125.

[39] S. Frølund and G. Agha, "Abstracting Interactions Based on Message Sets", *Seventh European Conference on Object-Oriented Programming (ECOOP'93)*, Kaiserslautern, Germany, 26-30 July, 1993, LNCS 707, Springer Verlag, pp. 346-360.

[40] M. Fukuda, L. F. Bic, M. B. Dillencourt and F. Merchant, "Intra- and Inter-Object Coordination with MESSENGERS", in [28], pp. 179-196.

[41] D. Gelernter and D. Kaminsky, "Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha", *Sixth ACM International Conference on Supercomputing,* Washington D. C., USA, 19-23 July, 1992, ACM Press, pp. 417-427.

[42] K. J. Goldman, B. Swaminathan, T. P. McCartney, M. D. Anderson and R. Sethuraman, "The Programmer's Playground: I/O Abstractions for User-Configurable Distributed Applications", *IEEE Transactions on Software Engineering* **21 (9)**, 1995, pp. 735-746.

[43] A. A. Holzbacher, "A Software Environment for Concurrent Coordinated Programming", in [28], pp. 249-266.

[44] B. Hirsbrunner, M. Aguilar and O. Krone, "CoLa: A Coordination Language for Massive Parallelism", *ACM Symposium on Principles of Distributed Computing (PODC'94)*, Los Angeles, Ca, USA, 14-17 Aug., 1994, pp. 384.

[45] T. Kielmann, "Designing a Coordination Model for Open Systems", in [28], pp. 267-284.

[46] M. Klein, "Challenges and Directions for Coordination Science", *Second International Conference on the Design of Cooperative Systems*, Juan-les-Pins, France, 12-14 June, 1996, pp. 705-722.

[47] J. Kramer, J. Magee and 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.

[48] J. Magee, N. Dulay and J. Kramer, "Structured Parallel and Distributed Programs", *Software Engineering Journal*, IEE, March 1996, pp. 73-82.

[49] T. W. Malone and K. Crowston, "The Interdisciplinary Study of Coordination", *ACM Computing Surveys* **26**, 1994, pp. 87-119.

[50] M. Marchini and M. Melgarejo, "Agora: Groupware Metaphors in OO Concurrent Programming", in [29].

[51] N. H. Minsky and J. Leichter, "Law-Governed Linda as a Coordination Model", in [29], pp. 125-145.

[52] R. Motschnig-Pitrik and R. T. Mittermeid, “Language Features for the Interconnection of Software Components”, *Advances in Computers* **43**, 1996, pp. 51-139.

[53] H. P. Nii, “Blackboard Systems”, *The Handbook of Artificial Intelligence* **4**, Addison Wesley, 1989, pp. 1-82.

[54] G. A. Papadopoulos and 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.

[55] G. A. Papadopoulos and F. Arbab, “Control-Based Coordination of Human and Other Activities in Cooperative Information Systems”, *Second International Conference on Coordination Models, Languages and Applications (Coordination’97)*, Berlin, Germany, 1-4 Sept., 1997, LNCS 1282, Springer Verlag, pp. 422-425.

[56] G. A. Papadopoulos and F. Arbab, “Control-Driven Coordination Programming in Shared Dataspace”, *Fourth International Conference on Parallel Computing Technologies (PaCT-97)*, Yaroslavl, Russia, 8-12 Sept., 1997, LNCS 1277, Springer Verlag, pp. 247-261.

[57] M. Papathomas. G. S. Blair and G. Coulson, “A Model for Active Object Coordination and its Use for Distributed Multimedia Applications”, in [29], pp. 162-175.

[58] F. DePaoli and F. Tisato, “Development of a Collaborative Application in CSDL”, *Thirteenth International Conference on Distributed Computing Systems*, 25-28 May, 1993, Pittsburgh, USA, IEEE Press, pp. 210-217.

[59] F. DePaoli and F. Tisato, “Cooperative Systems Configuration in CSDL”, *Fourteenth International Conference on Distributed Computing Systems*, 21-24 June, 1994, Poznan, Poland, IEEE Press, pp. 304-311.

[60] J. M. Purtilo, “The POLYLITH Software Bus”, *ACM Transactions on Programming Languages and Systems* **16 (1)**, 1994, pp. 151-174.

[61] M. Rem, “Associons: A Program Notation with Tuples instead of Variables”, *ACM Transactions on Programming Languages and Systems* **3 (3)**, 1981, pp. 251-262.

[62] S. Ren and G. A. Agha, “RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems”, *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, California, June 21-22, 1995.

[63] M. D. Rice and S. B. Seidman, “A Formal Model for Module Interconnection Languages”, *IEEE Transactions on Software Engineering* **20**, 1994, pp. 88-101.

[64] G.-C. Roman and H. C. Cunningham, “Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency”, *IEEE Transactions on Software Engineering* **16 (12)**, 1990, pp. 1361-1373.

[65] A. Rowstron and A. Wood, “BONITA: A Set of Tuple Space Primitives for Distributed Coordination”, *30th Hawaii International Conference on Systems Sciences (HICSS-30)*, Maui, Hawaii, 7-10 Jan., 1997, IEEE Press, Vol. 1, pp. 379-388.

[66] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik, “Abstractions for Software Architecture and Tools to Support Them”, *IEEE Transactions on Software Engineering* **21 (4)**, 1995, pp. 314-335.

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

[68] I. Sommerville and G. Dean, “PCL: A Language for Modelling Evolving System Architectures”, *Software Engineering Journal*, IEE, March 1996, pp. 111-121.

[69] R. Tolksdorf, “Coordinating Services in Open Distributed Systems With LAURA”, in [28], pp. 386-402.

[70] C. C. Weems, G. E. Weaver and S. G. Dropsho, “Linguistic Support for Heterogeneous Parallel Processing: A Survey and an Approach”, *Third Heterogeneous Computing Workshop*, Canceen, Mexico, 26 Apr., 1994, pp. 81-88.

[71] P. Wegner, “Coordination as Constrained Interaction”, in [28], pp. 28-33.

[72] G. Wilson (ed.), “Linda-Like Systems and Their Implementation”, Edinburgh Parallel Computing Centre, TR-91-13, 1991.

[73] P. Zave, “A Compositional Approach to Multiparadigm Programming”, *IEEE Software*, Sept. 1989, pp. 15-25.