Logic Channels: a Coordination Approach to Distributed Programming*

Manuel Díaz, Bartolomé Rubio, José María Troya
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga
Campus de Teatinos - 29071
Málaga - SPAIN

Abstract

In this paper, we present a new coordination model and a small set of programming notations for distributed programming that can be integrated in very different programming languages (imperative, declarative or object oriented). Both together, allow the development of distributed programs in a compositional way, by assembling different independent pieces of (possibly preexisting and heterogeneous) code. This approach is in the spirit of many other similar proposals as Linda, PCN, CC++, etc., but it allows multiparadigm and multilingual integration and provides a powerful set of concurrent programming techniques, inherited from Concurrent Logic Languages (CLLs), that can be efficiently implemented in distributed systems. The coordination model is based on Logic Channels; these are an evolution of the concept of shared logic variable used in CLLs that, with the same expressive power, can be more efficiently implemented in distributed systems.

1. Introduction

The development of distributed programs in a compositional way is an important challenge. The idea of program development by the composition of different independent pieces is a central point in software engineering for any kind of system, but in the case of distributed and parallel systems it is specially relevant [1]. However, in this kind of system, software composition is still far from being solved. Parallel and distributed systems normally present reactive behaviours, that have to be taken into account in the composition of software. The composition of reactive programs makes the definition of coordination models necessary in order to allow the plugging together of different pieces of code with possibly very different communication and synchronization requirements [2]. These coordination models must be expressive enough to cope with all the possible behaviours that a reactive system can show.

Much work has been achieved in the field of coordination and software composition for parallel and distributed systems in the last few years. One of the approaches to this problem has been to design new coordination languages, as PCN [3] and Linda [4]; other proposals provide extensions that can be added to existing languages, as CC++ [5] and Fortran M [6].

The Fortran M and CC++ approach consists of defining extensions to the Fortran and C++ programming language to allow process creation, communication and synchronization. In Fortran M, the communication model is based on typed one-to-one communication channels, and the main goal of the extension is to provide a simple mechanism to implement parallel programs. The goal of CC++ designers was not to define a coordination language, but to define a language extension to allow the integration of concurrence mechanisms into an object oriented language as C++, trying to take advantage of the combination of both paradigms (objects and concurrency). However, from a compositional point of view, the object abstraction, that can be very powerful to model some kind of systems, does not fulfill all the requirements of a coordination abstraction for reactive systems [1][2]. In addition to this, the CC++ approach does not allow the integration of other paradigms or languages.

In the PCN approach, a new language is defined; in this language both the coordination and computation models are integrated. In addition, the coordination model only allows the composition of transformational components, using CLL constructions (mainly streams and logic variables) to express concurrency and

* This work has been carried out in the framework of the Spanish CICYT project TIC94-0930-C02-01.

communication [7]. PCN allows the composition of software written in other programming languages, but this software has to show transformational behavior in order to allow composition.

The Linda approach allows the coordination of both transformational and reactive pieces of code, possibly written in different languages. Linda is a coordination language that defines a coordination model that can be applied to any programming language (that defines the computational model, according to the point of view of Linda designers [8]). Other authors [9], considered Linda just as an extension that can be added to different languages and that allows process creation, communication and synchronization. By means of a shared virtual memory (Tuple Space), Linda provides simple and powerful communication and synchronization mechanisms. The Linda model has been integrated in a wide variety of languages, including logic languages [10].

In this paper we present a new coordination model and a set of programming notations that can be integrated in different languages. Our proposal follows the Linda approach, but it defines a different coordination model, inspired in the CLL paradigm. Moreover, our main goal is perhaps different: our coordination model has been thought of as a tool for the compositional development of distributed applications. In this sense, we also try to overcome some of the drawbacks of Linda in the development of open systems, such as the lack of safety in the communication model and the inefficiency of distributed implementations [11].

In order to achieve our goal we have established some basic requirements for our coordination model:

- The model should be easy to integrate in languages based on different paradigms (declarative, imperative or object oriented).
- The language extensions should be simple and easy to understand.
- The model should permit the cooperation between software components implemented in different implementation languages and executing on different types of computers.
- The model should be expressive enough to allow the composition of different reactive behaviors under the same framework.
- The implementation of the coordination model should be efficient.

Our proposal is based on the utilization of shared logic variables and logic channels to model the communication and synchronization of different activities. Logic channels were defined in the context of the DRL language [12] with the objective of overcoming some efficiency problems found in the distributed implementations of CLLs [13]. Logic channels preserve all the advantages of shared logic variables (expressiveness, programming techniques, etc.), but allow a more efficient implementation of process communication. Both active and reactive control are important for simplifying distributed applications [14]. Logic channels and logic variables provide both kind of control through explicit and implicit communication. Besides the communication based on logic channels, our coordination model presents other characteristics that are not present in other coordination/composition approaches. For example, our model takes into account fault tolerance as a main requirement in the development of a distributed system. Other coordination models, such as Linda, have had to be extended to support fault tolerance [15].

The rest of the paper is structured as follows. In section 2, we present the coordination model. In section 3, we show the expressiveness of the model by means of two programming examples, using C as the base language. We introduce some implementation details in section 4 and, finally, in section 5 some conclusions are sketched.

2. The coordination model

Our aim is to define a coordination model to build applications by the composition of active pieces of computation. These active pieces are processes that are running asynchronously, in the same or in a different processor. Our model unifies all these kinds of computations under the framework of a single distributed virtual machine, with multiple logical processors (figure 1).

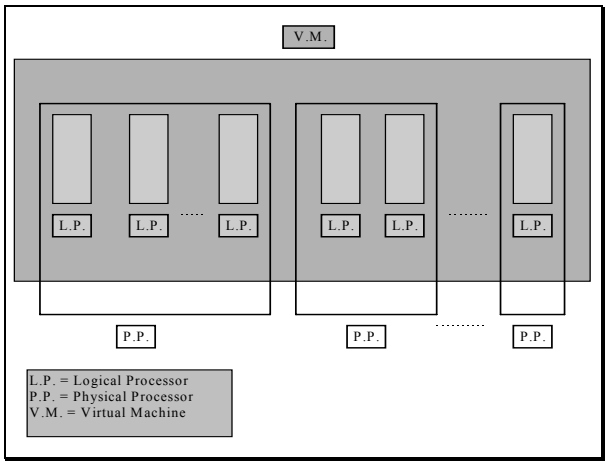

Figure 1. A virtual machine.

Each process will completely run on a logical processor, and different logical processors can be assigned to the same physical one. We will also see that within the model it is possible to interconnect different distributed virtual machines dynamically (subsection 2.4).

2.1. Process creation

A process is created dynamically by using an explicit process creation primitive: *ccall*. This primitive can be invoked from any of the pieces of code that are being composed. Process execution is asynchronous with respect to the computation of its parent process. In order to create a process, it is necessary to give four arguments:

ccall(Processor, Procedure, Status, Control)

- *Processor* is the identification of the logical processor where the process will be executed. We can alternatively use the terms *local* (the process is executed in the current processor) and *auto(x)* (the system will decide where to allocate the process; the logic variable *x* will be instantiated to the selected logical processor identifier).
- *Procedure* is the invocation of an executable piece of code; its parameters must contain the logic channels used to carry out the communication with other processes, as described in the next subsection. An unbound logic variable cannot be a parameter of a *Procedure*.
- *Status* and *Control* are two logic channels that will be used to monitor and to control externally process execution. Any of them can be omitted if no control or monitoring is needed. Their use will be described in subsection 2.3.

2.2. Communication model

The model is based on the utilization of logic channels to carry out communication and synchronization between the processes that form the computation. A logic channel can be seen as an ordered sequence of logic terms that can be shared by different producer and consumer processes, allowing many to many communication. A logic term can be a:

- Logic variable. A logic variable in our model is a special kind of single assignment variable that can be instantiated to any data type of the base or extended language, and also to logic structured terms and logic channels.
- Logic structured term. A logic structured term has the form *s(t1,t2,...,tn) / n>=0*, where *s* is a string (any character sequence) and *ti* is a logic term.

Synchronization is achieved by waiting on empty logic channels and non-instantiated logic variables. When a process needs to know the value of a logic variable to evaluate any expression, and the variable has still not been instantiated, the process automatically suspends until that variable becomes instantiated.

Processes access channels by mean of two primitives:

- *Put(Logic_Channel,Logic_Term*): Producers will use the put primitive to send information through the channel. Each time a put operation is executed, a new logic term *Logic_Term* is added to the end of the channel *Logic_Channel*. The put primitive is asynchronous, i.e. the process performing the put primitive does not have to block until the corresponding term is actually sent.
- *Get(Logic_Channel,Logic_Variable*): Consumers will use the get primitive to receive information from the channel. Each time a get operation is executed by a process, the logic variable *Logic_Variable* is instantiated to a new term obtained from the logic channel *Logic_Channel*. The get primitive is blocking, i.e. the process executing it will suspend if the channel is empty. The get primitive does not withdraw any data from the channel, from the point of view of the other consumers, and so, the same information can be received by them. Only when all the possible consumers of a term have consumed it, that term will be eliminated from the channel by the automatic garbage collection mechanisms.

When a process creation primitive is invoked, the logic channels that will be used by the process must be specified as arguments of this process. We call these logic channels statics to distinguish them of those created dynamically, when they are necessary, for example after receiving a message. In our model, a logic channel shared by different processes will be replicated in every logical processor where processes are running. The replication mechanism allows efficient local access and fault tolerance.

Besides this basic communication and synchronization, our coordination model provides other useful mechanisms: incremental communication, backward communication and logic channel interconnection.

2.2.1. Incremental communication. Complex data structures can be sent incrementally through logic channels in a nearly transparent way. When any part of a term is still unknown, a logic variable can be used to represent it. When this variable is instantiated later, the consumers will receive the rest of the data structure, in an explicit way. The new term can contain again logic variables.

2.2.2. Backward communication. A programming technique widely used in the field of CLLs is that of Incomplete Messages [7]. In outline, an incomplete message is a term, with unbound variables, that is communicated between two processes. The receiving process can instantiate these variables, sending the values

automatically to the producer of the term. Incomplete messages have different applications in CLLs, as for example:

- two-way communication using a single shared logic variable.
- expressing lazy communication.

Our model allows this elegant and useful way of backward communication by means of unbound logic variables sent through a logic channel. It avoids a consumer having to use another logic channel, or carry out a send operation to the same channel (from which it consumes), to answer some request. Both two-way and lazy communication are achieved within the model.

2.2.3. Channel interconnection. The model allows for the interconnection of logic channels. This interconnection is achieved by means of three basic primitives; these primitives allow the dynamic interconnection of logic channels during program execution, giving a great flexibility for the implementation of complex interaction protocols. We introduce these primitives below:

- *LC1 = LC2*. This primitive makes both channels a single one. Any data sent through any of the channels is also sent through the other one. If any of the channels are not empty when the channels are interconnected, the data of each channel is automatically sent to the other one.
- *LC1 <= LC2*. This primitive sends the data contained in channel *LC2* to *LC1*. After the interconnection, the *LC2* channel becomes empty.
- *connect(LC)*.This primitive allows a process to access a preexisting stable logic channel of a virtual machine. This is explained in subsection 2.4.

2.3. Exception handling

As noted in subsection 2.1, when a process uses the *ccall* primitive to dynamically create another one, it can specify as third and fourth arguments two logic channels that will be used to monitor and control the new created process. The parent process or another one that shares the *Status* channel can receive any information about the process internal status from it. On the other hand, any process sharing the *Control* channel can send any message through it to control the process execution. When the explicit process creation primitive is executed, not only the specified process is created, but an exception handling process is also created (in the same logical processor). This exception handling process is in charge of handling the *Status* and *Control* channels of the process. Inside a process, a message can also be sent to its *Control* channel by mean of the *raise(Exception)* primitive. This primitive raises an exception by sending it through the *Control* channel. Exceptions raised by the system are also sent through this channel, for example when a process fails the predefined exception *failed(process_name)* is sent through it.

2.4. Distributed virtual machines.

As we have introduced, the coordination model is based on the definition of a distributed virtual machine, consisting of a set of logical processors. Several virtual machines can also be interconnected; the interconnection of virtual machines has two main purposes:

- Providing a higher order composition mechanism, to allow the composition of coarse grain applications (these applications can be distributed on themselves). In this way we can distinguish two levels of interconnection: inside the virtual machine for tighter coupled processes, and between virtual machines for loosely coupled applications. These applications can be implemented in different implementation languages and executed on different types of computers.
- Providing a way of sharing resources between different applications running on the same physical processors.

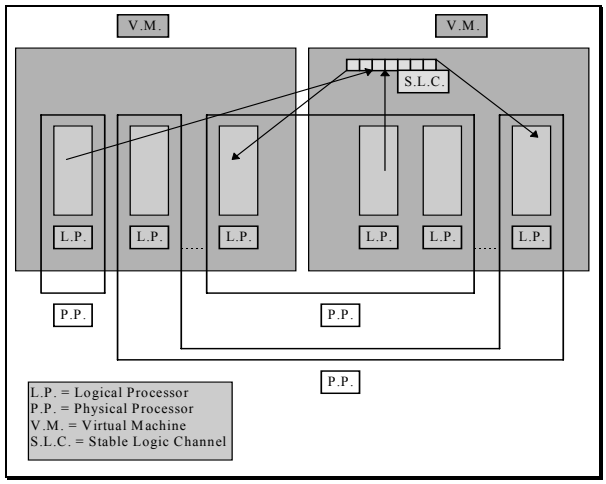

Figure 2. Interconnection of virtual machines.

Each virtual machine can be started independently sharing physical processors with other virtual machines. The way of interconnecting virtual machines is through stable logic channels. A stable logic channel is a global logic channel that is created when the virtual machine is initialized. Any process inside a virtual machine can access this logic channel by executing the *connect(LC)* primitive. After executing this primitive, the *LC* channel can be accessed as a normal one. In addition, by means of

this primitive, any process in another virtual machine can also use this primitive for accessing a stable logic channel. In figure 2 we can see how two virtual machines are interconnected through a stable logic channel, and how virtual machines can share physical processors.

The term stable is used in the sense that a logic channel persists until the virtual machine finishes its execution, independently of the termination of the processes that access the channel (from inside or outside the owner virtual machine). Stable logic channels provide our model with temporal uncoupling, in the sense that a process can communicate with processes that are not running at the same time.

3. Programming examples

In this section we show, by means of two examples, the suitability and expressiveness of our model. In order to code the examples, we have chosen C as the base language where to integrate our coordination model. Firstly, we introduce the necessary notation to integration.

- Each virtual machine is a program with a main function.
- Two new data types are introduced to declare logic channels and logic variables in the form:
  LC channel_name; (*logic channel declaration*)
  LV variable_name; (*logic variable declaration*)
  The accesses to obtain information from logic channels (*get*) and logic variables (==, *switch*) could suspend, as it is required in our model. Stable logic channels are declared as global variables. Anonymous logic variables and logic channels are represented as _.
- A logic term is represented as it was expressed in subsection 2.2. The identifier representing the name of a logic term must not be declared.
- The *ccall* primitive of the model is a predefined function that will be invoked in the form:
  ccall(Processor, Procedure, Status, Control);
  where *Procedure* is a function invocation.
- Primitives put and get are also predefined functions that will be invoked in the form:
  put(Logic_Channel,Logic_Term);
  get(Logic_Channel,Logic_Variable);
  with the meaning explained in subsection 2.2.
- By means of = we can instantiate a logic variable.
- By means of == (or *switch* constructor) we can know the value of a logic variable.
- The three channel interconnection primitives of the model are permitted: =, <= and *connect(LC)*.
- A function that will be executed by means of a *ccall* primitive, can contain at the end of its code a special piece of code representing its exception handling process. This process is expressed by means of a block of code preceded by the term *except_process*. *Ct* and *St* will be the predefined identifiers to refer to *Control* and *Status* channels of the process.

```
main()
{
 int i;
 LC Ch;
 ccall(auto(_),res_alloc(Ch),_,_);
 for (i=1;i<=CLIENTS;i++)
  ccall(auto(_),client(Ch),_,_);
}
void client(LC Ch)
{
 int n;
 while (1)
 {
  LV OK;
  n = necessary_resources();
  put(Ch,p(n,OK));
  if (OK == true) computation();
  put(Ch,r(n));
 }
}
void res_alloc(LC Ch)
{
 LC Lch;
 int n,resources=RESOURCES;
 while (1)
 {
  LV T,OK;
  get(Ch,T);
  switch T
  {
   case p(n,OK) :
    if (n <= resources)
    {
     resources = resources - n;
     OK = true;
    }
    else
     put(Lch,p(n,OK));
    break;
   case r(n) :
    resources = resources + n;
    Ch <= Lch;
  }
 }
```

Figure 3. Resource allocation example.

Figure 3 shows the utilization of this notation. The program implements an example of the Resource Allocation problem, where there is a process *res_alloc* that schedules the system resources between several *client* processes. Clients send a resource request message by means of the logic term *p(n,OK)*, and they send a resource release message through the logic term *r(n)*, where *n* is the number of resources required or released respectively, and *OK* is a logic variable for backward communication, by means of which the scheduler process, in a two-way communication form, grants requested resources. After a

client has sent a request message, it suspends until *OK* is instantiated. If the scheduler has not enough resources to serve a request message, it sends this message through a local channel *Lch*. Through the channel interconnection primitive <=, messages inside channel *Lch* are sent through the shared channel *Ch* when some resources are released.

```
main()
{ int i;
  LC Ch;
  ccall(auto(_),bag(Ch),_,_);
  for (i=1;i<=WORKERS;i++)
    ccall(auto(_),worker(Ch),_,_);
}
void worker(LC Ch)
{ int sum, i, cont=1,
       a[COL_A_ROWS_B],b[COL_A_ROWS_B];
  while (cont)
  {
    LV Task,R;
    put(Ch,more(Task));
    switch (Task)
    {
      case m(a,b,R):
        sum = 0;
        for (i=0;i< COL_A_ROWS_B;i++)
          sum = sum + a[i] * b[i];
        R = sum;
        break;
      case end:
        cont = 0;
    }
  }
}
void bag(LC Ch)
{ int i,j,a[ROWS_A] [COL_A_ROWS_B],
       b[COL_A_ROWS_B] [COLUMNS_B];
  LV c[ROWS_A] [COLUMNS_B];
  read(a,b); trans(b);
  for(i=0;i< ROWS_A;i++)
    for (j=0;j< COLUMNS_B;j++)
    {
      LV T,Task;
      get(Ch,T);
      if (T == more(Task)) Task = m(a[i],b[j],c[i][j]);
    }
  for (i=0;i<WORKERS;i++)
  {
    LV T,Task;
    get(Ch,T);
    if (T == more(Task)) Task = end;
  }
  print(c);
}
```

Figure 4. Matrix multiplication.

In figure 4, as an example of the Bag of Tasks problem, we outline a distributed solution for Matrix Multiplication. In the program, the task of multiplicating two matrices is divided into several tasks, each of them is in charge of the inner product of a row of the first matrix and a column of the second one. In concurrent programming, a good solution to that kind of problem is to program a *bag* process and several *worker* processes. The *bag* process distributes the tasks among *worker* processes that do the actual work. The *bag* process distributes the different tasks in a lazy way, i.e. when a *worker* process requests a job, the *bag* process gives it a task. *Worker* processes use the *Task* logic variable as backward communication to achieve that. On the other hand, results are also obtained by means of backward communication through *c[i][j]* variables. The *end* message is also used in a lazy way to terminate *worker* processes.

4. Implementation approach

The implementation is still under development, however the communication mechanism used in our coordination model has been previously evaluated on the implementation of the DRL language [12].

As in other proposals [4][5], our implementation strategy is based on source to source transformations together with the necessary libraries/classes and a suitable run-time system. In the transformation, we add to the source program the necessary code for:

- Suspension on non-instantiated logic variables. Before each reference to a logic variable it is necessary to test if the variable has been previously instantiated.
- Calling to the appropriate library functions to operate on logic terms. Logic variables are polymorphic, i.e. they can receive different types of data that have to be correctly managed.
- The format conversion between the representation of logic terms in the notation and its representation in the extended language.
- The creation and initialization of logic channels.
- The creation of local or remote processes.

The run-time system will be normally implemented as a daemon process and will be in charge of:

- Garbage collection. Our model uses a garbage collection algorithm inherited from the DRL implementation that allows for asynchronous independent garbage collection in each processor.
- The management of replicated logic channels. In order to maintain the coherence of replicas, if several producers for a logic channel exist in different physical processors, a leader is chosen to control the sending through the channel.
- System exception handling and process control.
- The management of stable logic channels.
- External monitoring of the state of the processes.
- Termination and deadlock detection.

A prototype is currently being achieved on a network of heterogeneous workstations. We are using C as the

base language and PVM [16] for supporting remote process creation and message passing.

5. Conclusions

We have presented the logic channel coordination model. This model provides the means for developing distributed applications by interconnecting heterogeneous pieces of code, developed with languages based on different paradigms. Our model has been specially designed to be used in the development of applications for loosely coupled distributed systems, providing a powerful communication mechanism that can be efficiently implemented in this kind of system. Both, the implementation techniques and the communication mechanism have been inherited from DRL, a Concurrent Logic Language for distributed systems.

We have shown how the coordination model primitives can be easily integrated into the C language. Some examples have been presented showing the suitability and expressiveness of the coordination model.

Some basic implementation issues have been outlined. The implementation is currently under development, but the model communication mechanism has been widely evaluated during the development of the DRL language. As future work, we are currently working on two main aspects: the definition of an operational semantics for the coordination model (including its integration in languages with different paradigms) and the implementation and evaluation of the prototypes.

References

[1] Nierstrasz, O. and Meijler, T. D. Requirements for a Composition Language. *Workshop on Model and Languages for Coordination of Parallelism and Distribution ECOOP94*. LNCS 1995. Springer Verlag.

[2] Bergmans, L. Composing Concurrent Objects. *PhD. Thesis*. *University of Twente*. June 1994.

[3] Foster, I. and Taylor, S. A compiler approach to scalable concurrent program design. *ACM Transactions on Programming Languages and Systems*. Vol. 16, N. 3. May 1994.

[4] Carriero, N. and Gelernter, D. *How to write parallel programs: a first course*. The MIT Press. 1990.

[5] Chandy, K. M. and Kesselman, C. CC++: *A Declarative Concurrent Object Oriented Programming Notation. Research Directions in Concurrent Object Oriented Programming*. The MIT Press. 1993.

[6] Foster, I. *Designing and Building Parallel Programs*. Addisson Wesley. 1995.

[7] Huntbach. M. M. and Ringwood, G. A. Programming in Concurrent Logic Languages. *IEEE Software*. November 1995.

[8] Carriero, N. and Gelernter, D. Coordination Languages and their significance. *Communications of the ACM*. Vol. 35, N. 2. February 1992.

[9] Khan, K. M. and Miller, M. S. Language Design and Open Systems. *The Ecology of Computation*. North Holland. 1988.

[10] Ciancarini, P. Distributed Programming with Logic Tuple Spaces. *New Generation Computing*. Vol. 12. N. 3. p. 251-284. Springer-Verlag. May 1994.

[11] Minsky, N. H. and Leichter, J. Law-Governed Linda as a Coordination Model. *Workshop on Model and Languages for Coordination of Parallelism and Distribution ECOOP94*. LNCS 1995. Springer Verlag.

[12] Díaz, M., Rubio, B. and Troya, J. M. DRL: a Distributed Real-time Logic Language. To appear in a *Special Issue of the Journal of Computer Languages on Extensions of Logic Programming*.

[13] Tick, E. The Deevolution of Concurrent Logic Programming Languages. *Journal of Logic Programming*, Tenth Anniversary Special Issue, Vol. 22. 1995.

[14] Goldman, K. J., Swaminathan, B., McCartney, T. P., Anderson, M. D. and Sethuraman, R. The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications.. *IEEE Transactions on Software Engineering*. Vol. 21. N. 9. September 1995.

[15] Bakken, D. E. and Schlichting, R. D. Supporting Fault Tolerant Parallel Programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*. Vol. 6. N. 3. March 1995.

[16] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press. Cambridge, Massachusetts. London, England. 1994.