Distributed system structuring using multi-functions

Jean-Pierre Banâtre, Michel Banâtre, Florimond Ployette

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France

Tél. (1) 39 63 55 11

Rapports de Recherche

N° 694

DISTRIBUTED SYSTEM STRUCTURING USING MULTI-FUNCTIONS

Jean-Pierre BANATRE
Michel BANATRE
Florimond PLOYETTE

JUIN 1987

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

PUBLICATION INTERNE N° 363 - MAI 1987
28 PAGES

Distributed System Structuring Using Multi-functions.

Les Multifonctions: un Outil pour la Structuration de Systèmes Distribués

Jean-Pierre BANATRE
IRISA/INSA and INRIA-Rennes,
Michel BANATRE, Florimond PLOYETTE
IRISA/INRIA-Rennes,
Campus de Beaulieu, 35042, RENNES-cedex (FRANCE)

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Abstract:

This paper introduces the new concept of multi-function that allows the description of parallel tasks which may be nested in a very general fashion. After the presentation of the concept of multi-function, some programming examples are given. Implementation aspects are discussed considering several target architectures (multi-processor, network of machines). Finally the concept is compared with other recent proposals.

Key Words: Distributed Systems, Multi-functions, nesting, parallel programming.

Résumé:

Cet article introduit le concept de multi-fonction qui permet la description de tâches parallèles qui peuvent être imbriquées de manière générale. Après la définition du concept de multi-fonction, nous en présentons quelques exemples d'utilisation. Les problèmes d'implémentation sont abordés en considérant plusieurs architectures cibles possibles: multi-processeurs , réseau de machines,... Nous terminons cette article par une comparaison du concept de multi-fonction avec les recentes propositions faites dans le domaine de la programmation d'applications parallèles.

Mots clés: Systèmes Distribués, Multi-fonctions, programmation parallèle.

Contents.

1-Introduction.

Recent advances in parallel architectures have spurred the development of new distributed systems [BLAC-85], [CHER-85]. One can view a distributed system as a set of interacting objects. Any object is composed of a concrete representation and of operational facilities (functions, processes,...). In the GOTHIC distributed system [BANA-86b], the concrete representation of an object may be located on one node (centralized object), split up into several parts located on different nodes (fragmented object) or replicated on several nodes (replicated object). Operational facilities should make it possible to describe parallel computations dealing with objects possessing such concrete structures. For example, if one considers a virtual storage made out of all the segments situated on all k nodes of a distributed architecture, the segment table will be split up into k sub-tables (one per node). This segment table is a fragmented object. Searching for information in such a table would naturally involve k parallel computations (one per sub-table).

Our research focuses on providing appropriate language support for distributed programming. Actually, even if distributed systems are now fairly common place, the programming of distributed applications is still a very complex task. This is mainly due to the lack of adequate distributed software structuring concepts.

Two research directions are possible: the message passing model or the parameter passing/result return model (procedural model). The message passing model has been used in several systems projects [CHER-85], [RASH-81]. Its main advantages stand in the great flexibility it offers for description of process interactions. However, this freedom may turn out to be dangerous for many reasons: (i) the semantics of message passing models are still being debated and those which exist are quite difficult to understand, (ii) the programming experience with this model is quite recent and there is no recognized programming method. It is our belief that programming with a message passing model is more error-prone than with a procedural model. In particular, new types of programming error have to be considered such as deadlocks, starvation.

The procedural model is conceptually simple and its semantics are now well-defined, in particular if we restrict ourselves to call by value. The run-time structures induced by the procedural model are well-understood in the case of local calls and also (though some work remains to be done)

in case of remote calls [BIRR-84], [SHRI-82]. Furthermore recursion is an important and powerful tool for describing simply quite complex computations which could only described in a very awkward way using conventional tools. For these reasons the procedural model has been recognized as fundamental for system structuring. However, the main defect of this model stands in its inadequacy to describe parallel computations and as such it cannot satisfy our goals.

We have designed a computational model which provides the ability to describe atomic nested concurrent computations. Our idea consists of generalizing in an orthogonal fashion the procedure concept. Actually the body of an extended procedure is composed of several computations which may run in parallel. Furthermore, we extend the notion of nesting introduced by recursion to a general form of nesting allowing a set of parallel computations $p_1 \ldots p_k$ to synchronize and suspend their execution in order to give control to another set of parallel computations $q_1, \ldots q_n$ which when terminated give the control back to $p_1 \ldots p_k$. This nesting appears as a generalization of the nesting inherent in recursion, where one computation (caller) stops, gives control to another computation (callee) which, when terminated, gives control back to the caller. This generalization of the procedural model has been encapsulated under the term **multi-function** in GOTHIC.

Section 2 of this paper describes in more detail the concept of multi-function. Section 3 presents two programming examples: one dealing with a virtual storage facility and the other with document management. Implementation issues are discussed in section 4 where a prototype run-time system is described. The conclusion summarizes the significance of multi-function concept, compares it with other proposals and discusses future plans.

2. The concept of multi-function.

A well-known structuring concept for classical operating systems and even for distributed operating systems is the procedure or function. The procedure is an abstraction of the notion of block with strict rules for communication with the environment (parameter and result passing mechanisms). Furthermore, procedures offer the possibility of nested computation through "recursive" calls.

Our purpose was to discover a somewhat similar concept allowing for:

-simultaneous processing of different components,

-parameter/result communication scheme,

-general nesting facilities.

2.1.Block and parallel clauses.

A block may be represented as (D;F) where D stands for declarations and F a sequence of instructions and blocks.

Given two blocks, B_1: (D_1; F_{11}; F_{12}) and B_2: (D_2; F_2),

B_2 "nested within" B1 may be represented as:

(1) (D_1; F_{11}; (D_2; F_2); F_{12}), with the following properties:

D_1 and F_{11} are the first executed, then block B_1 is interrupted. D_2 and F_2 are executed and B_1 is resumed thus allowing the execution of F_{12}. Visibility rules are such that F_2 "sees" D_2 and D_1, F_{11} and F_{12} "see" D_1.

A parallel clause may be described as:

cobegin (D_1; F_1)//(D_2; F_2)// ... //(D_n; F_n) **coend**

Components (D_i; F_i) may be run in parallel. Several languages allow the following structure:

(2)
```
begin
    D0;
        cobegin
            (D1; F1) // ... // (Dn; Fn)
        coend;
    F0
end
```

where D_0 is first executed, then the (D_i; F_i)'s ($i \geq 1$) are executed and finally F_0 is executed. Visibility rules are such that F_0 "sees" D_0, but cannot "see" the D_i's ($i \geq 1$).

Procedural nesting (1) is referred to as 1-1 nesting (one caller, one callee), and parallel clause nesting is referred to as 1-p nesting (one caller, p callees). This last form of nesting is the one generally found in distributed systems where such concepts as nested actions or nested activities are implemented [LISK-84], [MUEL-83], [MOSS-81]. Let us now describe a more general form of

nesting as introduced in GOTHIC.

2.2.A general form of nesting.

The nesting of a parallel clause within another parallel clause may be visualized as follows:

cobegin

$(A_{11}/$	.	$/A_{12})$
$(A_{21}/$	(B_1)	$/A_{22})$
.	(B_2)	.
.	.	.
.	(B_P)	.
$(A_{n1}/$		$/A_{n2})$

coend

Where the parallel clause **cobegin** $(B_1)// \ldots //(B_p)$ **coend** is nested within the parallel clause **cobegin** $(A_{11}/A_{12})//\ldots//(A_{n1}/A_{n2})$ **coend**. The execution of this structure can be described as follows:

A_{i1}'s sequences of instructions are initiated, and when they have **all** reached their "/", B_j's are executed. Upon termination of **all** B_j's, A_{i2}'s are resumed with the property that A_{i2} may access the context defined in A_{i1}.

This form of nesting is the most general (n callers, p callees) and we can see that 1-1 and 1-p nesting are particular cases of this n-p nesting [BANA-80]. After this informal description of the nested parallel clause, let us introduce the concept of multi-function.

2.3. General form of multi-functions

In the same way as procedures are abstractions of blocks, we can define a computational model, the **multi-function**, which may be seen as the abstraction of the parallel clause. It is possible to call a multi-function from a procedure but also from another multi-function, thus providing a general form of nesting.

The description of multi-functions can be seen as a generalization of PASCAL functions.

For example, here is the definition of a multi-function called "mf":

```
multi-function mf;
      (x, y, z:integer): (u, v, w:integer);

      var
       <declarations>

      cobegin
            (x,y)u: begin     ... return u end //        (1)
            (z)v: begin       ...  return v end //       (2)
            (y,z)w: begin     ...   return w end        (3)
      coend;
```

This multi-function has three components. Component (1) deals with input parameters (x,y) and delivers the output u, component (2) deals with input parameter z and delivers v and finally (3) deals with input parameters (y,z) and delivers w. Let us describe the simplest multi-function call (or 1-p call).

a) 1-p multi-function call.

The statement (l,m,n) := mf(a,b,c) describes a call to mf where input parameters are (a,b,c) and the final result of the call will be assigned to variables l, m, n. The execution of this multi-function call may be depicted as follows:

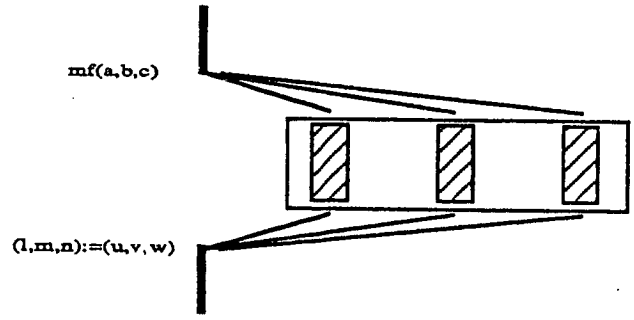


Figure 1: 1-3 call.

Where the call mf(a,b,c) results in:

-distribution of input parameters to components of the newly created instance of the multi-function mf,

-parallel execution of the components,

-synchronization for result construction and transmission,

-resumption of the caller.

This execution scheme can be seen as a generalization of the usual procedure call (1-1 to 1-3).

b) Coordinated multi-function call (or n-p call).

Let us now introduce the most general multi-function call, the coordinated call (or n-p call). Assuming the multi-function mf previously defined, consider the following program skeleton:

```
            cobegin
(1)             (integer a, k, l; ... ; (k,l):=mf (x<-a).(u,v); ... ) //
(2)             (integer b, c, m; ... ; m:=mf(y<-b,z<-c).v; ...  ) //
(3)             (integer n;  ... ; n:=mf().w;  ... )
            coend
```

The call to mf is distributed among the three components of the parallel clause. For example the expression m:=mf(y<-b,z<-c).v means that component (2) of the parallel clause contributes to the mf call by providing value b for the parameter y and value c for the parameter z . This call is referred to as a "partial" call. The actual call to mf is effective only after all partial calls have taken place. After completion of the execution of mf , the u and v parts of the result are transmitted to component (1) and assigned to variables k and l, the v part of the result is transmitted to component (2) and assigned to variable m and the w part of the result is transmitted to component (3) and assigned to variable n. This kind of multi-function call is referred to as "coordinated call". A partial call to mf such that n:=mf().w appearing in component (3) means that this component is involved in the coordinated call without providing any input parameter, but, after the processing of the call, receives the w part of the result .

The execution of the coordinated call to mf can be pictured as follows:

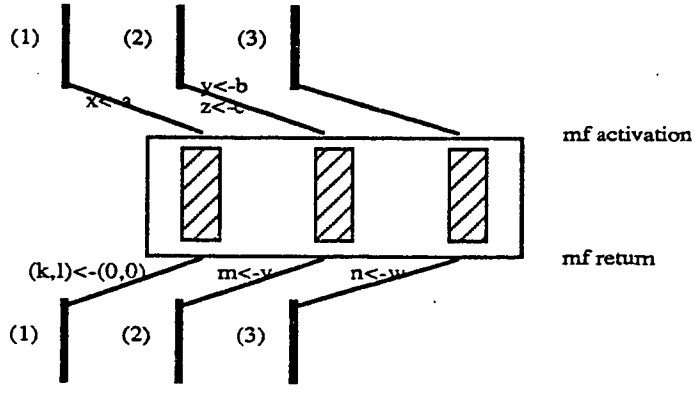


Figure 2: 3-3 call.

The execution of this call can be described as follows:

-coordination of components (1), (2) and (3) of the calling parallel clause for input parameter transmission. Notice that component (3) gets coordinated without providing any input parameter,

-distribution of the input parameters to the components,

-parallel execution of the components,

-synchronization of the components for result construction and transmission,

-distribution of the result to the components of the calling parallel clause.

2.4. Multi-functions and atomicity.

Our experience with ENCHERE [BANA-86a] convinced us that the atomicity property is fundamental in the construction of reliable distributed systems. So, in the same way as ENCHERE activities (which are quite similar to cooperating process groups) had to be atomic, we require multi-function call and execution to be atomic. This atomicity ensures that:

i)-either appropriate parameters are transmitted to every component of the multi-function or a failure is reported to the caller,

ii)-in case of failure of a component of the callee, a failure is reported to the caller and the effects of this tentative call are undone,

iii)-all the results, elaborated independently by the components of the callee should be transmitted to the caller. This means that, in case of failure during result transmission, this operation is retried till it is successful.

2.5. Communications between components.

Components of a multi-function can communicate only by using coordinated calls: actually all components have to synchronize in order to perform a coordinated call to a multi-function which implements information exchange. However, systematic use of coordinated calls for communication may turn out to be very inefficient due to over-synchronisation.

In order to overcome this problem, we have extended the notion of "locality of context". Actually, in the same way as it is possible to declare a set of objects local to a procedure, we allow the

declaration of objects local to a multi-function. These objects do not belong to any component in particular, but can be accessed by all components: they constitute a context shared by the components.Of course, appropriate synchronization tools (such as monitors) have to be provided in order to manage these shared objects. We do not detail this aspect here, we only give a short illustrative example:

```
begin
    multi-function exchange;
        (val1:integer):(val2:integer);

        var ex:record(x:integer, b:bool):=(0,false);

        cobegin
            (val1):():begin ex.x:=val1;  ex.b:=true  end//

            ():val2: begin
                        while not ex.b do end;
                        val2:=ex.x; ex.b:=false; return val2
                    end
        coend;

    cobegin
        (var y:integer; y:=3;...;exchange(val1<-y);...)//
        (var z:integer; z:=exchange().val2;...)
    coend
end.
```

The multi-function exchange uses a shared variable ex to implement the transfer from val1 to val2. One can notice that a synchronization tool is necessary to ensure that ex.b is accessed in a mutual exclusion mode.

3. Applications of multi-functions.

This section describes some examples of application of multi-functions. These examples are intended to exhibit the use of synchronization facilities offered by multi-functions in order to model agreement problems.

3.1.Generalized rendez vous.

The use of nested multi-functions provides a possible means of implementing

communications between components of a multi-function, as illustrated by the following example:

```
begin

multi-function rdv;
        (input: integer): (output: integer);
        cobegin
                (input): output:
                        begin
                        output:= input
                        end
        coend; #rdv#

multi-function com;
        cobegin
(1)             begin
                ...
                rdv(input<-9);
                ...
                end //

(2)             begin
                y:integer;
                ...
                y:=rdv().output;
                ...
                end //
        coend; #com#

...
#com invocation#
com
...

end.
```

Component (1) of the multi-function com sends the value 9 to the component (2) by calling the multi-function rdv. This nested call implements a generalized form of rendez vous [HOAR-78].

3.2.Cooperation between groups of processes.

As an example, imagine the situation where a common agreement must be reached by two groups of people, G1 and G2. The following communication scheme may be used:

Group G1 has a meeting and makes a proposal, which is sent to group G2 for approval or modification, eventually G1 takes the final decision. Notice that this is a group to group communication and that a member of a group cannot be distinguished.

If we model the behaviour of a group by the execution of a multi-function, where a component represents a member of the group, control flow governing the above decision taking algorithm may be pictured as follows (fig. 3).

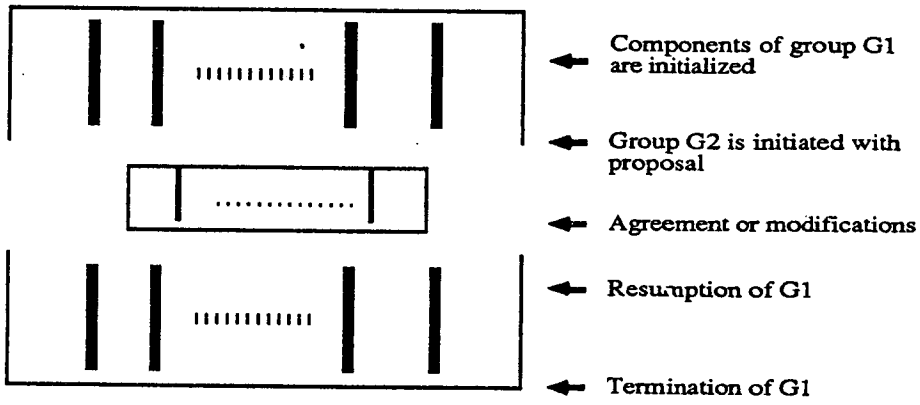


Figure 3: Cooperation between G1 and G2.

3.3. Global virtual memory.

Let us consider the problem of implementing a global virtual memory over a network quite similar to the one described in [LEAC-83]. In the most general case, an object is made out of a number of pages which may be located on different nodes.

Imagine that we want to allocate a number of pages ("pages") for a new object O. This will be achieved in three steps:

(i) A multi-function ("allocate-pages") is called. A component of allocate-pages runs on each node S_i of the network and determines for each S_i the number (ap_Si) of pages potentially available on node S_i.

(ii) The different numbers ap_Si are determined by a coordinated call to another multi-function "available-pages". Available-pages receives as input the number of pages request for the object, the number (fp_Si) of free pages of each Si and determines for each site the number of pages which can be allocated to the object.

(iii) The execution of the multi-function allocate-pages is resumed and actual allocation takes place on each node by updating local data structures (page lists).

Figure 4 visualizes control and information transfers occurring during this distributed page allocation algorithm:

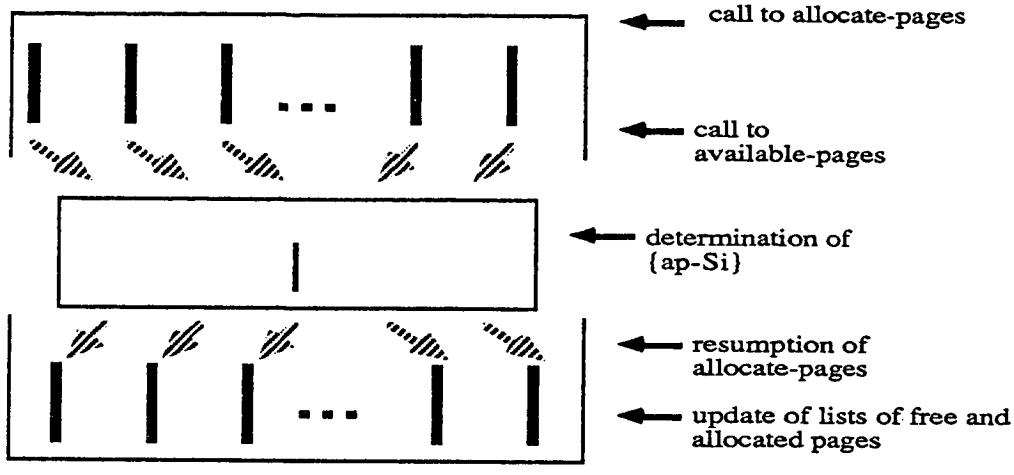


Figure 4: Page allocation.

Now we give a detailed program of the global virtual memory example.

We assume the availability of the type **list** with three operations:

add:**list**⊕**list**→**list**, which concatenates two lists.

remove:N⊕**list**→**list**, which removes the n first elements of a list and creates a new list with these elements.

card:**list**→ N, which delivers the cardinality of a list.

Let us define the two following types:

type site_memory: record(allocated-pages, free-pages:**list**)

type obj_repr: record(S1_pages, S2_pages: **list**)

Actually, we restrict our problem where only two sites are available. Dealing with the problem in its full generality would lead us to manage list of sites or multisets of sites, thus making the solution less clear. So pages allocated for the representation of an object belong either to S1 or to S2.

The following program represents a possible solution:

```
begin
...
S1, S2 : site_memory;
Obj : obj_repr; #lists of pages#
...
multi-function available_pages;
        (nb_pages, fp_S1, fp_S2 : integer): (ap_S1, ap_S2: integer);

        var ap_S1, ap_S2:integer;

        cobegin
        (nb_pages,fp_S1,fp_S2):(ap_S1,ap_S2):

                var np: integer;

                begin
                        if fp_S1≥nb_pages
                        then
                        ap_S1:=nb_pages;
                        ap_S2:=0
                        else
                        np:=nb_pages-fp_S1;
                        if np>fp_S2
                                then
                                #error#
                                ap_S1:=ap_S2:=0;
                                else
                                ap_S1:=fp_S1;
                                ap_S2:=np
                        fi
                fi;
                return (ap_S1,ap_S2)
                end

        coend #available_pages#


multi-function allocate_pages;
        (pages: integer): O: obj_repr;

        var O:obj_repr;

        cobegin
        (pages):(O.S1_pages):
                var v_ap_S1: integer;
                begin
                v_ap_S1:=available_pages(nb_pages<-pages,
                                        fp_S1<-card(S1.free_pages)).ap_S1;

                if v_ap_S1≠0
                        then
                        O.S1_pages=remove(v_ap_S1,S1.free_pages);
                        S1.allocated_pages:=
                        append(S1.allocated_pages,O.S1_pages)
                        else
```

```
                O.S1_pages:= nil;
        fi;
        return O.S1_pages
        end  //

(pages):(O.S2_pages):
        var v_ap_S2: integer;
        begin
        v_ap_S2:=available_pages(nb_pages<-pages,
                                 fp_S2<-card(S2.free_pages)).ap_S2;

        if v_ap_S2≠0
                then
                O.S2_pages=remove(v_ap_S2,S2.free_pages);
                S2.allocated_pages:=
                append(S2.allocated_pages,O.S2_pages)
                else
                O.S2_pages:= nil;
        fi;
        return O.S2_pages
        end

coend #allocate-pages#

...
Obj:=allocate_pages(pages<-10).O;
...

end.
```

3.4.Document management system.

In a document management system, a document may be represented in two parts: (i) the unformatted text and the list of editing commands and (ii) the formatted text itself.

Imagine that we want to create the representation of a new document. This can be achieve in three steps:

(i) Search in the directory for a place where the attributes of the document to be created will eventually be stored. In our example, we consider that the directory is duplicated,

(ii) Actual creation of the document,

(iii) Actual update of both copies of the directory.

Steps (i) and (iii) are implemented by a multi-function "directory_update" which deals with the two copies of the directory. Step (ii) is implemented by a multi-function "create_document" with two components dealing respectively with part (i) and (ii) of the representation of the document.

Figure 5 visualizes control and information transfer during document creation:

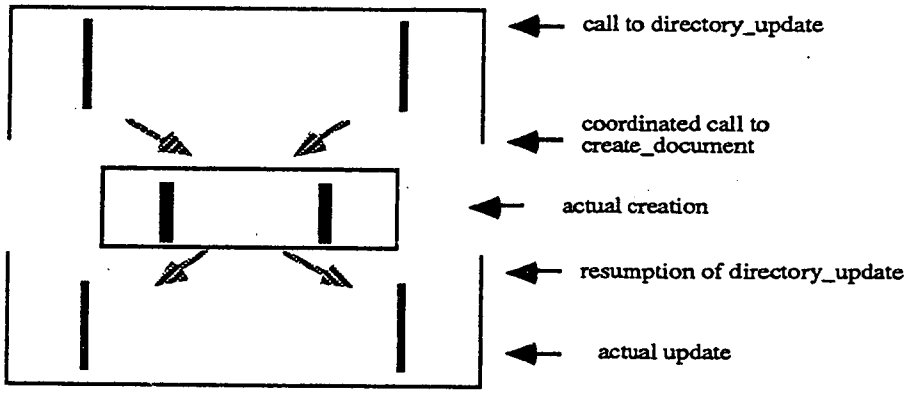


Figure 5: Control transfer between directory_update and create_document

The following program is a solution to this problem:

Data structures are described by three types: **document** which represent the document itself, a **list_of_documents** and **directory** which is a replicated list_of_documents. We assume the availability of an operation new_file to create files.

```
begin
...
type document: record(text:file, form_text:file);
type list_of_document: record([...] ↑article, index:integer);
type directory: record(replic_dir1, replic_dir2: list_of_document);
...
multi-function create_document:():(doc:↑document);

        var doc:document;

        cobegin
        ():doc.text:
                begin
                doc.text:=new_file;
                return doc.text
                end//

        ():doc.form_text:
                begin
                doc.form_text:=new_file;
                return doc.form_text
                end
        coend;

multi-function directory_update: (dir:directory):();

        var doc: document;
```

```
        cobegin
        (dir.replic_dir1):
                begin
                ...
                dir.replic_dir1[dir.replic_dir1.index]:=create_document().doc;
                ...
                end//

        (dir.replic_dir2):
                begin
                ...
                dir.replic_dir2[dir.replic_dir2.index]:=create_document().doc;
                ...
                end
        coend;
...
dir: directory;
...
directory_update(dir);
...
end.
```

This program exhibits a 2-2 coordinated call. This program does not take into account the possibility that directory_update could not find any place in the directory in order to store the attributes of the document to be created. This problem is solved, as usual, by an exception handling mechanism which can be considered as an extension of a CLU-like one to multi-functions [LISK-79].

From these examples, it is clear that multi-functions are well suited to program distributed problems. Generally, the treatment of such problems involves three steps:

i)-invocation of a multi-function each component of which is dedicated to the processing of a particular data fragment. In the virtual memory example, the set of pages can be considered as fragmented data distributed over the set of sites. Each search for free pages is performed by a component of the multi-function,

ii)-common agreement between components. Of course, this phase can be expressed with a nested multi-function,

iii)-termination of the multi-function execution and delivery of the result.

4. Implementation issues.

A top-down approach has been adopted for design of multi-function implementation. In

this section, we present the general problem and its solution in terms of context management. Different implementations are derived from this general solution depending on the underlying architecture, uni-processor machine, multi-processor machine or network of machines.

The conventional implementation of procedure is based on a stack with static and dynamic chains. Run-time supporting parallel expression has been studied and is based on so called "cactus stacks" [CLEA-69]. Every parallel statement in the expression is run with its local stack, and these local stacks are rooted on the calling environment stack. Upon termination of all the parallel statements, the local evaluation stacks are discarded and the calling environment becomes the current environment.

4.1. The problem of implementing multi-function calls.

In order to implement multi-function calls, one should be prepared to manage "fragmented" contexts. Actually the context of an instance of a multi-function is the set of the contexts of its components. As these components may be executed on a set of different nodes, one has to consider that the global context of a multi-function is a fragmented object.

Two problems have to be solved:

(i) management of nested fragmented contexts,

(ii) management of synchronization involved in a n-p multi-function call.

4.1.1. An extended stack machine for managing fragmented contexts.

In the conventional stack model for procedure implementation, a stack is composed of several layers. Each layer is the representation of a single context (a block or a procedure body). A layer is itself made out of a static part which corresponds to the local declarations of the block and a dynamic part which is used for expression evaluation. The different layers are linked together by two chains:

-the static chain which binds a context (i.e. the environment defined by a procedure) to its defining environment,

-the dynamic chain which binds a context to its caller environment.

When a procedure is called, the value representing this procedure is pushed on the current stack and a new context is created which is statically bound to the defining environment of the procedure. After completion of the procedure body, the caller context is restored using the dynamic chain.

This model is extended in order to deal with multi-functions.Actually one can see the extended representation as a meta-stack where every meta-layer is in fact a fragmented object composed of as many stacks as there are components in the multi-function.

When a multi-function mf is called, its representation is sought in its defining environment and a new meta-layer is constructed on the top of the current meta-stack. This meta-layer is a fragmented stack, each component of which contains a part of the multi-function value. Figure 6 shows the defining environment, the caller environment and the multi-function context.

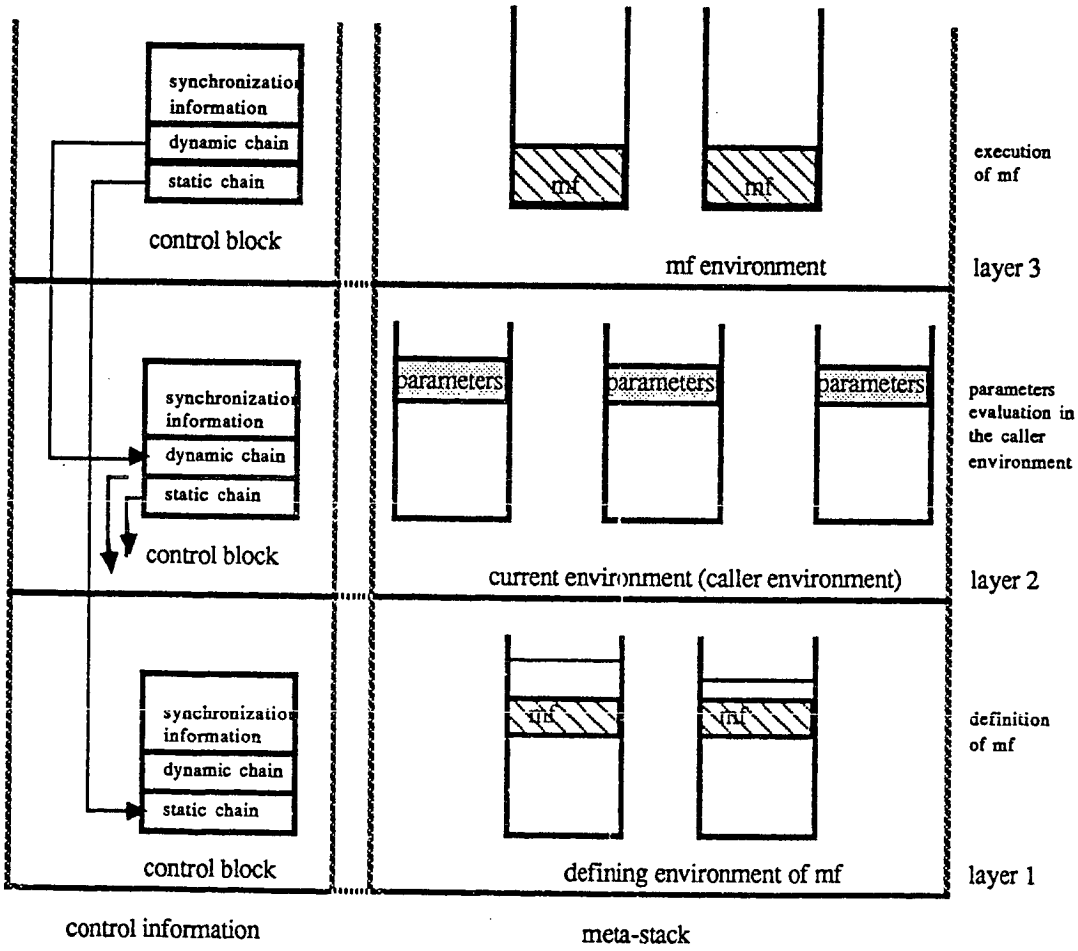


Figure 6: The extended stack machine.

Each meta-layer (representing a context) contains one stack per parallel computation and a control block that holds binding and synchronization informations. Once the context of the callee is created, the parameters are transferred from the caller environment to the callee. Then, the current environment becomes the multi-function one. When the multi-function is running, it can access, via the static chain, the values of its defining environment. The dynamic chain is used to restore the environment of the caller after completion of the callee.

4.1.2. Synchronization problems.

A coordinated call is effective only when all the components of the caller multi-function have performed their partial call. In order to check this property, a boolean synchronization vector (start[1:n]) is attached to every instance of multi-function(with n components) , where start[i] is true if the i^{th} partial call has been performed. The actual multi-function call will be effective when for all i in [1 ..n], start[i]=true. A second vector terminate[1:p] is also necessary, terminate[i] is set to true when the i^{th} component of the multi-function is terminated. The callee multi-function itself will terminate when terminate[i]=true for all i in [1..p]. These two vectors are stored in the synchronization part of the control block associated with each multi-function.

4.2. Mapping the implementation model to particular architectures.

In the following, we consider the problem of mapping this general model to various architectures, from a uni-processor machine to a network of multi-processor machine.

In a uni-processor machine implementation, all the data structures are located in the same memory. The CPU is shared by threads of control which correspond to multi-function components.

In a multiprocessor implementation, each processor has a local memory and shares a global memory with other processors. The control blocks are located in global memory. They are linked together and form a kind of centralized stack of control. The stacks necessary for the evaluation of the components are located in the local memories.

In a distributed implementation there is no shared memory. In such an architecture, control blocks have to be distributed over the different nodes and to be managed as fragmented objects.

As is pictured in figure 7, the static and dynamic chains are lists of fragmented objects. The synchronization information also is also a fragmented object.

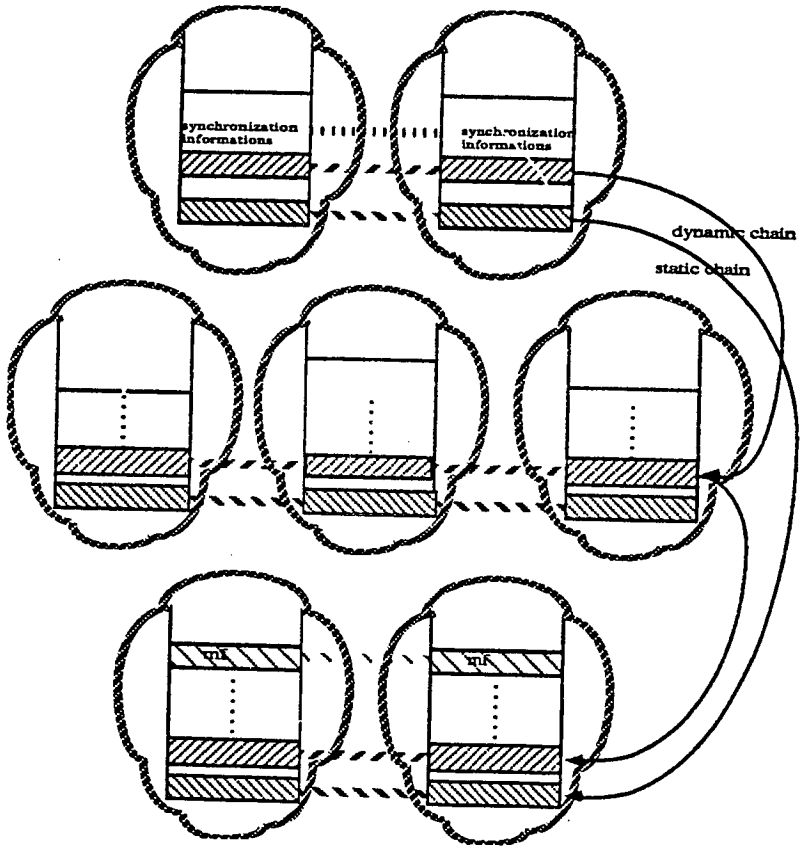
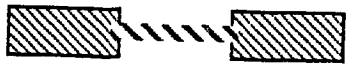


Figure 7: The decentralized implementation.

On this figure,  represents a fragmented object.

The main implementation problem is the management of these fragmented objects which are located on distinct addressing spaces and even on distinct machines over a network.

Fragmented objects are managed by the kernel of the system. A fragmented object has to be created and updated atomically. A new protocol related to remote multi-function call is under

investigation.

A first experiment in implementing multi-function has been made on top of UNIX running on SUN machines. Each multi-function component is run as a UNIX process and the parameter passing mechanism uses the Remote Procedure Call of UNIX heavily. The aim of this first attempt to implement multi-functions was to prove the effectiveness of the concept.

A multi-processor implementation has been undertaken and is near to completion. It is based on the elements given in the above section. The main difficulties are related to load balancing among processors and shared memory management.

5. Discussion.

This paper has presented a new computation model derived from the procedural model. Its main properties lie in its ability to describe atomic nested concurrent computations. Examples of uses have been given and implementation issues have been discussed.

Concerning related work, we can mention two attempts in generalizing the procedural concept, the replicated calls [COOP-85] and the parallel-RPC [MART-86].

The notion of replicated call has been introduced as a means of achieving high fault-tolerance. Actually, the fundamental concept is the procedure, however a user may specify that a procedure call should be replicated, i.e. the procedure is executed n times with the same arguments on n different nodes and possibly in parallel. Appropriate techniques such as voting, allow a decision as to whether the replicated call was successful or not. One can realize that a replicated call on a unique processor is meaningless...the idea being to exploit distribution in order to enhance fault-tolerance.

A parallel remote procedure call initiates a procedure that executes in parallel at n different sites. The caller remains blocked while the n procedures execute. The termination of a partial result (from one procedure) implies the resumption of the caller which processes this result and can decide whether or not to wait for further partial results. This proposal is quite similar to the preceeding one, except that it provides a programmed control on the partial results, here too, the basic concept is still the procedure. They have been designed with specific problems in mind, such as replicated data management and in general voting based algorithms.

So both attempts are founded on the procedural model which is adapted in order to cope with problems specific to several distributed algorithms.

Multi-functions proceed differently from the two approaches above. Actually, they constitute a new language concept whose purpose is the description of concurrent computations independently from any implementation concern. It is as meaningful to execute a multi-function on a single processor as on a set of processors. In the first case, the components of the multi-function will be executed pseudo-concurently and in the second case, they will be executed concurrently, but the end result will be the same.

The concepts described in this paper have been integrated in an object-oriented language for the GOTHIC project. This language supports the notion of abstract data type. A GOTHIC abstract data type is,

i) a description of its signature (list of types describing the operational facilities of this abstract data type),

ii) a description of the concrete data structure representing the object. This data structure may be declared as fragmented or replicated. Only certain data structures may be fragmented, e.g., records, array, files,

(iii) the operational part is given by a set of multi-functions. Exception handling mechanisms have been designed.

GOTHIC objects are instances of abstract data types which are created dynamically. These features have been added to the MODULA language [WIRT-85] and a compiler is under construction for a target machine based on the Mc68020 processor.

Concerning future work, several topics are currently addressed, for example:

(i) the implementation of a reliable Remote Multi-Function Call (RMFC) protocol. This work is currently underway as a major aspect of the GOTHIC project. The general problem may be summarized as follows: Find the protocols which allow a multi-function whose components are located on a set of nodes to call a multi-function whose components are situated on a disjoint set of nodes. A generalization of RPC protocols [BIRR-84] has to be devised.

(ii) the programming of distributed or parallel applications. Only more pratice will allow

a full evaluation of multifunctions as far as parallel processing is concerned. In particular, recent work shows how recursive multi-functions may be used in order to describe parallel breadthfirst tree processing [TRIL-87]. In this application, multi-functions allow a natural programming of a solution, which does not assume the introduction of extra-data structures to capture the control constrainst. Moreover initial ideas towards a programming method for the systematic construction of multi-functions has been put foward and needs further investigation.

To conclude, the concept of multi-function appears to provide a satisfactory answer to problem crucial to distributed applications. However, only long term experience can support a final verdict.

References:

[BANA-80] BANATRE J.P.
Contribution à l'étude de méthodes et d'outils de construction de programmes parallèles et fiables.
Thèse d'Etat, Université de Rennes 1, déc. 1980.

[BANA-86a] BANATRE J.P., BANATRE M., LAPALME G., PLOYETTE Fl.
The Design and Building of ENCHERE, a Distributed Electronic Marketing System.
Com. of the ACM, Vol 29, n°1, January 1986. pp.19-29.

[BANA-86b] BANATRE J.P., BANATRE M., PLOYETTE Fl.
An Overview of the GOTHIC Distributed Operating System.
INRIA Research Report, n° 504, March 1986.

[BLAC-85] BLACK A.C.
Supporting Distributed Applications: Experience wirh Eden.
Proc. 10th ACM Symp. Operat. Syst. Principles, Dec. 1-4, 1985, pp. 181-193.

[BIRR-84] BIRRELL A., NELSON B.
Implementing Remote Procedure Calls.
ACM TOCS, Vol 2, n°1, Feb. 1984, pp. 39-59.

[CHER-85] CHERITON D.P., ZWAENEPOEL W.
Distributed Process Group in the V Kernel.
ACM TOCS, Vol. 3, N°2, May 1985, pp. 77-107.

[CLEA-69] CLEARY J.G.
Process Handling on Burroughs B6500
Proc. of 4th Australian Computer Conference, Adelaïde, South Australia, 1969.

[COOP-85] COOPER E.
Replicated Distributed Programs.
Ph.D. Thesis. Technical Report UCB/CSD 85/231. Univ. of Calif. Berkeley, May, 1985.

[HOAR-78] HOARE C.A.R.
Communicating Sequential Processes.
Com. ACM 21,8, Aug.1978, pp.666-677.

[LEAC-83] LEACH P.J., LEVINE P.H. DOUROS B.P., HAMILTON J.A., NELSON D.L., STUMPF B.L.
The architecture of an Integrated Local Network.
IEEE Journal on selected areas in comm.,Nov. 1983, pp.842-856.

[LISK-79] LISKOV B., SNYDER A.
Exception Handling in CLU.
IEEE Trans. on Soft. Eng., vol. SE-5 6, pp. 546-558 (Nov. 1979).

[LISK-84] LISKOV B.
The Argus Language and System.
LNCS 190,1984, pp. 343-430.

[MART-86] MARTIN B.
Parallel Remote Procedure Call and the Portable C Stub Compiler.
Proc. of the workshop on Design Principles for Experimental Distributed Systems,
Purdue University, West-Lafayette, Indiana, Oct. 16-17, 1986.

[MOSS-81] MOSS J.E.B.
Nested Transactions: an Approach to Reliable Distributed Computing.
MIT/LCS/TR-260, M.I.T. LCS, Cambridge, Ma., 1981.

[MUEL-83] MUELLER E., MOORE J., POPEK G.
A Nested Transaction System for LOCUS.
Proc of 9th SOSP, Bretton Woods, N.H., Oct. 10-13.

[RASH-81] RASHID R., ROBERTSON G.
Accent: A Communication Oriented Network Operating System Kernel.
Proc. of 8th Symp. on Operating Systems Principles. Pacific Grove, Calif. Dec. 1981.

[SHRI-82] SHRIVASTAVA S., PANZIERI F.
The Design of a Reliable Remote Procedure Call Mechanism.
IEEE Trans. on Computer, vol C-31, n° 37, July 1982, pp. 692-697.

[TRIL-37] TRILLING L.
New Tools for Recursive Parallel Programming and their application to a Search Problem.
INRIA Research Report, (to appear).

[WIRT-85] WIRTH N.
Programming in Modula-2, third ed.
Springer-Verlag, New-York, 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique