DISTRIBUTED ACTIVE OBJECTS: A METHODOLOGICAL PROPOSAL AND TOOL FOR DISTRIBUTED PROGRAMMING WITH TRANSPUTER SYSTEMS

M.Capel*, J.M.Troya*, A.Palma*

*Dpt. Lenguajes y Sistemas Informáticos. Facultad de Ciencias. Universidad de Granada. Avda. Fuentenueva, s/n. 18071 Granada. SPAIN. E-mail: mcapel@ugr.es

*Dpt. Lenguajes y Ciencias de la Computación. Universidad de Málaga. El Ejido, s/n. 29013 Málaga. SPAIN.

This work mainly addresses the development of a methodology for programming distributed algorithms and applications in multicomputers, based on the specification and implementation of global shared ADTs separate from the sequential algorithmic code. To represent instances of the global ADTs and global control operations, a class of distributed objects is proposed. The specifications of high-level user-defined topologies along with the implementation of ADT operations are carried out for each distributed object in a proposed language, precompiled on top of Occam.

1. Introduction.

Within the framework of the parallelization scheme known as SPMD (Single Program Multiple Data) [1], we propose a programming methodology based on a class of distributed objects, which allows us to encapsulate the global data in distributed programs and also the global control operations defined on these data. Unlike other schemes [2], [3], [4], [5], in our case the specification of each distributed object includes the definition of a high-level user-defined topology (virtual topology) which coincides with the logical communication structure of a data domain [6].

The data are distributed assuming a domain decomposition scheme [2]. This scheme presents a series of drawbacks which make its direct use in distributed applications difficult because: (a) the decomposition of the domain does not match the number of nodes; (b) the logical communication structure between the data partitions does not match the physical topology of the network; (c) even if matching between the domain partition and the number of nodes exists, there are global computations which need to operate on data mapped onto different nodes.

The case (a) implies defining operations necessary to update the values of global shared data by the processes, which are very difficult to implement. Moreover, the problem stressed in (b) leads us to the inclusion of a complex routing code in the user distributed programs.

The last case (c) reflects the existence of global control operations defined between different partitions of one data domain.

To derive distributed applications in an easy and flexible way, the programmer is helped by an object based specification scheme and a programming language implemented on top of Occam. The proposed language can be implemented in other languages and systems.

The ODA (Spanish acronym for Distributed Active Object) based methodology has been applied to develop software for Transputer systems [6] and to derive schemes for the distribution of sequential algorithm design techniques such as Divide-and-Conquer, Backtracking and Branch-and-Bound [7]. With respect to previous publications, this paper is now based on the greater degree of virtualization obtained for the ODA description language by the implementation of a kernel with routing and deadlock prevention capabilities.

2. Distributed Active Objects.

In the SPMD scheme, global data could appear and should be distributed. The encapsulation of these data and their operations is achieved by defining global ADTs in the distributed applications.
Each global ADT of an object is implemented in the proposed methodology by an Active Distributed Object (ODA) module. The specification of an ODA mainly addresses two aspects:

- Specification of the represented object, which will be finally distributed by replication in the nodes of the network (ODA-replicas). The global data represented by each ODA is interpreted as a domain to be partitioned.
- One logical topology, defined by the user, is included in the specification of each ODA module. This software topology simplifies the implementation communication code in the whole application.

2.1. Communication Schemes in ODAs.

Two communication schemes appear in the application programming based on ODAs.

- Communication relationships between application-processes and ODA-replicas. When an application process calls on an operation defined on the ODA interface, it sends a message to the ODA-replica co-resident in the same processor. This interaction defines a client-server relationship between each application-process and its ODA-replica. The methods invoked in this way by the application process are named server methods.

- Communication relationships between ODA-replicas. The ODA defines some operations which involve communication between different ODA-replicas. These operations are performed following a message diffusion communication scheme, according to the logical topology defined by the user. In this case, the methods invoked are called diffusion methods. Examples of these operations are those which maintain the consistency between the local data in the ODA-replicas, or those which define a data movement between the ODA-replicas.

3. ODA description language.

A description language has been defined to specify ODAs. This language is intended to address the methodological aspects of ODAs. We distinguish two main parts in the description of an ODA module: definition of the ODA, in which the logical communication structure of the ODA is specified, and implementation of the ODA, which deals with the procedural aspects of the implementation of the ODA.

```
ODA.description ::= ODA.definition ODA.implementation
ODA.definition ::= ODA.DEFINITION name (REF name, NODES name) [(formal)]
                     [use.clause:]
                     [constants.definition]
                     topology.definition
                     [protocol.declaration]
                     remote.communication.channels
                     local.application.channels
                     constants.definition ::= CONST (defined.constant)
                     topology.definition ::= TOPOLOGY (link.definition)
link.definition ::= link.item
                     | DO (link.definition)
                     | DO replicator link.definition
                     | IF (link.alternative)
                     | IF replicator link.alternative
                     | IF replicator link.alternative
link.item ::= CONNECT ODA.name [integer.expr] (LNK) link.subscript
                     TO ODA.name [integer.expr] (LNK) link.subscript
link.alternative ::= boolean link.definition
protocol.definition ::= PROTOCOL (definition.item)
definition.item ::= name IS simple.protocol:
                     | name IS sequential.protocol:
                     | name CASE tagged.protocol:
remote.communication.channels ::= DIFFUSION.MESSAGES (channel.assignment)
channel.assignment ::= channel.list ONTO link.list
link.list ::= (LNK) [integer.expr]
channel.list ::= (channel.declaration)
channel.declaration ::= [integer.expr]
                     CHANOF protocol (,, name)
link.list ::= (,, [LNK] [integer.expr])
                     | replicator [LNK] [integer.expr]
local.application.channels ::= SHARED.OBJECTS (channel.declaration)
ODA.implementation ::= ODA.IMPLEMENTATION (use.clause:)
                     (channel.declaration)
                     IN.SECTION section.body
                     SHARED.OBJECTS section.body
                     OUT.SECTION section.body
use.clause ::= USE (,, name)
section.body ::= (specification)
                     [INIT (assignment:)]
                     PORT.construction
                     PORT.construction ::= PORT alternative
                     alternative ::= (# guarded.command
                     | replicator guarded.command
                     guarded.command ::= variant.guarded.command | guard body
                     guard ::= [boolean] simple.input -> [boolean ->
                     variant.guarded.command ::= [boolean] CASE (name)
                     variant.input -> body)
body ::= sentence
simple.input ::= RECEIVE(name, (,, variable))
variant.input ::= RECEIVE(tag (,, variable))
sentence ::= SKIP | STOP | TERMINATE | output
| assignment | sequence | conditional | selection
| name (,, actual) | specification sentence
output ::= SEND(name, (,, expression))
| SEND(name, tag (,, expression))
```

Fig. 1. EBNF for the ODA Description Language.
ODA description language in EBNF notation are shown in fig.1; these rules have to be added to the Occam syntax rules.

An example of a distributed program is presented to illustrate the ODAs description language, which consists in a distributed sum performed by a set of application processes connected through a binary tree topology (fig.2). The processes in the leaf nodes of the tree obtain integer values which are sent to their parent nodes. At the other end, the processes in the internal nodes perform the partial sums of the values received from their sons, sending them to their parents. At the end of the computation the process in the root node sends the result to the host. A tree ODA is defined as a distributed object which implements the communication structure to connect these application processes.

3.1. ODA Definition part.

The identification of each replica and the number of replicas are the first parameters declared in the ODA header (fig.3). Other parameters can also be declared in the header and instantiated at configuration time. Constants of global use in the ODA module may be declared in the CONST section.

In the TOPOLOGY section, the user defines the logical structure of communication between the ODA-replicas (virtual topology). The concept of virtual link is used to specify connections between ODA-replicas. The virtual topology is described by means of connections between virtual links of different ODA replicas. The maximum number of virtual links is only limited by the implementation restrictions.

The syntax used is similar to the one used in the network description of the configuration language of the Occam toolset\textsuperscript{TM}. The virtual links of each ODA-replica are named {oda.name[i][LNK][j]}, where {oda.name} is the ODA identifier declared in the first line, {i} is the replica identifier or replicator, and {j} is the link subscript.

After the TOPOLOGY section, the protocols of the messages will be declared, using the same semantics of the protocol declarations as in Occam.

In the DIFFUSION.MESSAGES section, the channels which communicate the different ODA-replicas (remote communication channels) are declared. These channels are mapped onto the specified virtual links using the ONTO sentence in the channel declarations. Finally, the interface channels between the application processes and their respective ODA-replicas (local application channels) are declared in the SHARED.OBJECTS section.

3.2. ODA Implementation part.

The procedural aspects of the implementation of the methods implementation are described in this part. Each ODA-replica is structured in three concurrent processes, described in their respective sections (see fig.4):

(a) The IN.SECTION describes the input process, which receives messages from other replicas of the ODA and sends them to the output process, or to the object server.

(b) The OUT.SECTION describes the output process, which receives messages from the input process and from the object server and sends them...
to other ODA-replicas.

c) The **SHARED.OBJECTS** section describes the object server process, which maintains the local data and is connected to the application process (fig.5), implementing the methods of the interface with the application program.

While the object server is intended to implement the server methods, the input and output processes are used in the implementation of communication operations between replicas (diffusion methods). These two processes should be explicitly programmed by the user rather than being automatically generated, because their code could depend on the actual value of the replicator variable.

The communication between these three concurrent processes is performed using the internal channels \((\text{in.obj, obj.out})\) declared before the ODA sections.

A **PORT** construction is declared in each section of the ODA implementation part, which has the semantic of the iteration of a non-deterministic selection of guarded commands, using the syntax proposed in fig.1.

\[
\text{Appl.Process (REP \ rep.id, NODES replicas)}
\]

\[
\text{WHILE true}
\]

\[
\text{IF (rep.id >= (replicas/2)) -- leaf node}
\]

\[
\text{INT value:}
\]

\[
\text{SEQ}
\]

\[
\text{... obtain initial value}
\]

\[
\text{SEND (appl.tree, value)} -- send it to tree
\]

\[
\text{TRUE -- non-leaf node}
\]

\[
\text{INT partial.sum, value1, value2:}
\]

\[
\text{SEQ}
\]

\[
\text{RECEIVE (tree.appl, value1)}
\]

\[
\text{RECEIVE (tree.appl, value2)}
\]

\[
\text{partial.sum := value1 + value2}
\]

\[
\text{IF (rep.id <> 0) -- internal node}
\]

\[
\text{SEND (appl.tree, partial.sum)}
\]

\[
\text{TRUE -- (root) node connected to host}
\]

\[
\text{... send final result to host}
\]

\[
\]

Fig.5. Code for the application processes.

4. Implementation of the virtual topology.

4.1. Separating virtual from physical topologies.

Besides the definition of the distributed object class with its methods and the data type represented, the user defines the virtual topology. The virtual topology would not depend on the physical topology of the transputer network, but on the requirements of the data communication in the application.

The programmer can define virtual topologies without thinking about the physical topology and the number of processors in the target network. The number of replicas of an ODA-module could be defined as a generic parameter in the ODA definition and fixed at configuration time. These features allow the **reusability** and **scalability** of the ODA modules.

4.2. Implementation of a router.

A router kernel instance is executed in every processor in order to supply communication between nodes whether they are directly connected or not. So, the router provides the possibility of making virtually direct connections between all the processors in the network.

Each router kernel supplies a set of channels to send/receive messages to/from processes allocated in other processors (named **router interface channels**) to the processes connected to it (i.e., replicas of one or more different ODA-modules). The protocol of the router interface channels includes the target-processor and target-process identifiers in the messages.

The prototype of the router implemented up to now uses a **store-and-forward** technique providing an optimal, dynamic and deadlock-free routing. A new version of the router is being implemented using a **wormhole** technique for several physical topologies.

4.3. Interface between router and ODAs.

Implementation of the virtual links.

In the **TOPOLOGY** section of the definition of the ODA, connections between virtual links of different
ODA-replicas are specified. From the point of view of the router, these connections may be seen as paths through the router kernel instances loaded in each processor on the transputer network.

In the DIFFUSION.MESSAGES section, the remote communication channels in every ODA-replica are declared and mapped onto the specified virtual links. These channels are implemented by router interface channels. Special interface functions are used: (i) to translate the data between the protocols declared in the ODA definition and the router protocols, (ii) to calculate the destinations of the messages according to the topology definition. A scheme of the integration between a tree ODA-replica and a router kernel instance is shown in fig.6.

From the point of view of the integration between router and ODAs, the router is divided into two parts:

(a) The constants, protocols and functions are selected from a library to provide the interface between the ODA modules and the router (stressed above in i and ii). These are independent of the physical topology of the transputer network in which the final program will run, and they are included in the stages before the configuration phase.

(b) A specific code of the router kernel is used for each physical topology and integrated with the application at configuration time. The hardware descriptions and their respective router modules (corresponding to different physical topologies) are extracted from the hardware description library provided by the system.

5. Integration of the ODAs’ development with the Occam toolset.

The descriptions of the ODA-modules and the application programs are precompiled and translated to the Occam code. At this stage the appropriate references to the router interface libraries are made as shown in fig.7.

```occam
#INCLUDE "Router.inc" -- router protocols & constants
#USE "Router.lib" -- router interface library
...
PROC in.section(CHAN OF ROUTER.PROC from.lson,
from.rson,
CHAN OF tree.data in.obj,
VAL INT rep.id, replicas)
...
-- declarations
WHILE true
  ALT
    from.lson ? message containing data
    SEQ
      ... -- ((( translate data between
                      -- ROUTER.PROT and tree.data
                      -- protocols )))
      in.obj ? data
    from.rson ? message containing data
    SEQ
      ... -- ((( translate data between
                      -- ROUTER.PROT and tree.data
                      -- protocols )))
      in.obj ? data
PROC shared.objects(CHAN OF tree.data in.obj,
                    obj.out, appl.tree, tree.appl,
                    VAL INT rep.id, replicas)
...
-- declarations
WHILE true
  ALT
    in.obj ? data
    appl.tree ? data
    obj.out ? data
PROC out.section(CHAN OF ROUTER.PROT to.father,
                  tree.data obj.out,
                  VAL INT rep.id, replicas)
...
-- declarations
WHILE true
  ALT
    obj.out ? data
    SEQ
      ... -- ((( calculate message destination
                        -- according to the virtual
                        -- topology definition )))
      ... -- ((( translate data between
                        -- tree.data and ROUTER.PROT
                        -- protocols )))
      to.father ! message containing data
```

Fig. 7. Code generated for the input, output and object server processes in the example.

The Occam modules obtained are compiled and linked. Each linked unit corresponds to an ODA-module or to the application program. A linked unit corresponding to an ODA-module is self-consistent and independent of the physical topology of the network, that is, it may be reused in another application which assumes the same ODA interface.
The ODAs' precompiler also produces files with software information about each ODA module. This information is joined with a hardware description, and with the appropriate router (linked) module from the hardware description library to produce a configuration description file (fig.8).

```ocaml
VAL NODES IS ...: -- number of processors
... -- hardware description: ring of NODES processors
... -- mapping description
CONFIG
#INCLUDE "Router.inc"
#USE "Routing.lku" -- router for a ring network
#USE "Tree.lku" -- tree ODA code
#USE "ApplPgm.lku" -- application ODA program

(NODES) CHAN OF ROUTER.PROT suc,pred:
-- arrays of channels to build the physical ring
PAR k=0 FOR NODES
PROCESSOR Transputer[k]

CHAN OF ROUTER.PROT to.father,

CHAN OF oda.tree appl.tree, tree.appl:

PAR
Router ([suc[k-1],pred[k],pred[k]],
    [suc[k],pred[k]],
    [to.father],
    [from.son, from.son], k, NODES)
oda.tree ([to.father],
    [from.son, from.son],
    appl.tree, tree.appl, k, NODES)
appl.proc (appl.tree, tree.appl, k, NODES)
;
-- + (-.) represents + module N (- module N)
```

Fig.8. Configuration description file generated.

When the generation of the configuration description file is made the parameters of each ODA replica header (see fig.2, rep.id and replicas) are instantiated according to the transputer network in which the application will run.

The configuration description file produces a configuration data file to finally obtain a bootable program for a specific network. All the steps followed in the generation of a bootable program with the ODA description language are shown in fig.9.

6. Application to the transformation of sequential algorithmic techniques.

The transformation of the most useful algorithmic design techniques into their corresponding distributed versions presents serious drawbacks, mainly due to:
- The existence of irregular data structures which are necessary in the implementation of those schemes
- The need for global synchronization between the sequential processes of some of those schemes
- Implementation of global control operations, such as a share-work operation called by the processes to obtain load balancing.

With irregular data structures the need for a complex communication code could appear in the programs, thus disturbing the comprehension of the algorithmic scheme. Consequently, it should be established as a principle that the sequential scheme of the algorithms be maintained as much as possible in their distributed versions.

The implementation of global control and global synchronization operations in the distributed applications should be avoided because this code is usually hardware dependent. Thus, this kind of code must be hidden from the user programs.

The ODA based methodology has been used in [8] to derive distributed schemes for several sequential algorithmic design techniques. Some of these schemes are now being employed in realistic distributed applications, obtaining promising results.

As an application example of the methodology the
distributed Divide-and-Conquer (DC) algorithm scheme will now be presented, derived from the ODA based approach. The distributed sum, discussed in section 3, could be considered as a first attempt at programming an algorithm based on the distributed DC scheme. This algorithm would have been a complete instance of the scheme if the initial assignment of subproblems to processors had been included. A new version of this algorithm is discussed below, as it is a good, clear example of a distributed DC programmed with ODAs.


These algorithms use a top-down method, which decomposes one initial problem into many atomic subproblems, thus being easier to solve than the initial one. The subproblems obtained have the same form as that of the original problem, are solved in the same way, and can then be duplicated. The subproblems are divided as far as possible, and then an adhoc subalgorithm is applied to each atomic subproblem, which is known as the base function.

The sequential scheme of the DC algorithms may be summarized as follows:

i) Test if the (sub)problem X is indivisible then apply the base function.
ii) Divide X into a number of subproblems (this number is problem dependent) named correspondent subproblems
iii) Combine the solutions of correspondent subproblems recursively until the global solution is obtained.

6.2. Distribution scheme of DC algorithms.

The computation of a distributed DC algorithm is carried out in two phases:
1) initial loading phase
2) computation phase

After the initial loading phase of subproblems onto processors, the following computation cycle phase begins in each processor:

a) Communication between those which contain correspondent subproblems. This computation stage implies the need for global synchronization between the processes.
b) Local computation, which operates on the old value of the processor with the received value from the correspondent subproblem.
c) Combining operation, which maps the solutions of each pair of correspondent subproblems onto a combined (sub)problem solution.

The distribution of DC algorithms presents problems because the mapping of subproblems onto the nodes of the network must be dynamic [9]. This is a drawback for the implementation of the communication and combining phases of the algorithm. The methodology based on ODAs improves the programmability of the distributed DC algorithm scheme with the definition of two ODAs.

---

Fig. 6.1. Direct-Tree ODA implementation.
Direct-Tree. This ODA carries out the division into subproblems and their automatic loading into the processors (fig.10). Actually it does not imply a dynamic migration scheme of subproblems, but the number of subproblems generated depends on the parameters in the ODA header. This allows the use of this ODA in configurations with different numbers of processors and the ODA reusability is improved. The subproblems are assigned to the processors at the beginning of the computation in the initial loading phase (1) time.

Inverse-Tree. This has the functionality of a global distributed tree which provides operations to communicate correspondent subproblems transparently to the user programs. This ODA has already been implemented in fig.4 for the distributed sum algorithm. Its topology is a tree in which the arcs are directed from the leaves to the root; these arcs represent the channels of communication between processors which contain correspondent subproblems.

The scheme of the DC algorithms with ODAs consists in the two ODAs above and one sequential process replicated in all the nodes: its code is like that of fig.5, but calls to the Direct-Tree ODA operations could be included in the code to carry out the initial loading phase. The sequential process replicas synchronize the processors in the network because until the two subproblems are received the local computation cannot begin, and no data may be emitted from a processor until the local computation ends.

7. Conclusions and future work.

A methodological proposal and a programming language based on the object paradigm have been proposed to carry out the SPMD programming style in an easy and flexible way on an Occam-Transputer platform.

The proposed method is based on a class of distributed objects, named ODAs, which represent the global shared data, their operations and their communications protocols. Thus, the implementation of the distributed ODA modules has been discussed, following a scheme of programming based on abstractions. Finally, a distributed scheme for the DC algorithm design technique has been presented. The example given is mainly intended to introduce the ODAs based methodology and the language. In previous publications, other schemes have been derived with this methodology, such as distributed Branch-and-Bound and Backtracking algorithms. These distributed schemes will be applied to try to solve real world problems and they will be reported in future work.

REFERENCES