# ORCA: A LANGUAGE FOR DISTRIBUTED PROGRAMMING

*Henri E. Bal†*
*Andrew S. Tanenbaum*
*M. Frans Kaashoek*

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

We present a simple model of *shared data-objects*, which extends the abstract data type model to support distributed programming. Our model essentially provides shared address space semantics, rather than message passing semantics, without requiring physical shared memory to be present in the target system. We also propose a new programming language, Orca, based on shared data-objects. A compiler and three different run time systems for Orca exist, which have been in use for over a year now.

## 1. INTRODUCTION

The growing interest in distributed computing systems has resulted in a large number of languages for programming such systems [Bal et al. 1989c]. Many of these languages are oriented towards systems programming and are typically used for distributed operating systems, file servers, and other systems programs. In this paper we will discuss a language designed for implementing distributed *user* applications. In particular, our language is intended for parallel, high-performance applications.

An important reason for implementing parallel applications on distributed systems—rather than on shared-memory multiprocessors—is the fact that distributed systems *scale* very well. It is relatively easy to build a distributed computing system (e.g., a hypercube, Transputer grid, or a collection of workstations connected through an Ethernet) with hundreds of processors. Connecting the same number of processors to a shared memory is difficult and expensive. In addition, distributed systems offer a good price/performance ratio and are available to many people.

On the other hand, programming distributed systems is potentially more difficult than programming shared-memory machines, since it is impossible for processes on different machines to share data. A process that needs information from a remote process has to send a message to that process to ask for the information. Moreover, message passing introduces noticeable delays, so the information may be out-of-date by the time it arrives. Multiprocessors do not have this problem, since shared data can simply be put in the shared memory, where they are accessible by every processor immediately.

† This research was supported in part by the Netherlands Organization for Scientific Research (N.W.O.) under grant 125-30-10.

Our approach tries to combine the advantages of distributed systems and shared-memory multiprocessors. We present a language for programming distributed systems based on *logically shared data* rather than message passing. In other words, the programmer can use shared data—much as on a multiprocessor—but the implementation of the language does not need physical shared memory.

The structure of the rest of the paper is as follows. In Section 2 we will describe the underlying model of the language, called the shared data-object model, and the language itself, called Orca. In Section 3 we will give examples of distributed programming in Orca. Finally, in Section 4 we compare our approach to those of others and we will briefly describe the current implementation status of the project.

## 2. AN OVERVIEW OF THE SHARED DATA-OBJECT MODEL

Our model uses processes for expressing parallelism and shared data-objects for communication and synchronization between processes. An object in our model is a passive entity; it only contains data. An object is an instance of an *object type*, which is essentially an *abstract data type.* The data stored in an object can only be accessed through the operations defined for the object's abstract type.

A *process* is an active component. Processes are created dynamically. Each process contains a single thread of control. Objects are created by declaring variables of object types. When a process spawns a child process, it can pass any of its objects as **shared** parameters to the child. The children can pass the object to *their* children, and so on. In this way, the object gets distributed among some of the descendants of the process that declared the object. All these processes *share* the object and can perform the same set of operations on it, as defined by the object's type. Changes to the object made by one process are visible to other processes, so a shared object is a communication channel between processes. This mechanism is similar to call-by-sharing in CLU [Liskov et al. 1977].

Each operation is applied to only a single shared data-object. Such operations are executed *indivisibly*.* Processes will not see intermediate states of an operation. The data of the object will never get in an inconsistent state due to simultaneous operation invocations.

An object type definition consists of a *specification* part and an *implementation* part. The specification part defines one or more operations on objects of the given type. The declaration of an object type *IntQueue* is shown in Figure 1.

```
object specification IntQueue;
   operation append(X: integer);
       # append X to the queue
   operation remove_head(): integer;
       # wait until queue is not empty; remove and
       # return head element
end;
```

**Figure 1**: specification of object type IntQueue.

The implementation part contains the data of the object, code to initialize the data of new instances (objects) of the type, and code implementing the operations. The code implementing an operation on an object can access the object's internal data. An operation that

* We use the term *indivisible* rather than *atomic*, as atomicity usually also implies *recoverability*.

does not block simply consists of a sequence of statements. Blocking operations consist of one or more *guarded commands*:

```
operation name(parameters);
begin
  guard expr_1 do statements_1 od;
  guard expr_2 do statements_2 od;
   ...
  guard expr_n do statements_n od;
end;
```

The expressions must be side-effect free boolean expressions. The operation blocks (suspends) until at least one of the guards evaluates to ‘‘true.’’ Next, one true guard is selected nondeterministically, and its sequence of statements is executed. An outline of the implementation of object type `IntQueue` is shown in Figure 2.

```
object implementation IntQueue;
   Q: list of integer;  # internal representation

   operation append(X: integer);
   begin  # nonblocking operation
       add X to end of Q;
   end;

   operation remove_head(): integer;
       R: integer;
   begin
       guard Q not empty do
             R := first element of Q;
             remove R from Q;
             return R;
       od;
   end;

begin
   Q := empty;  # initialization of an IntQueue object
end;
```

**Figure 2**: Outline of implementation of object type IntQueue.

Objects can be created and operated on as follows:

```
myqueue: IntQueue; # create an object of type IntQueue
...
myqueue$append(34);  # add 34 to myqueue
...
x := myqueue$remove_head();  # remove first element
```

Shared objects can be used to transfer data between processes as well as to synchronize processes. Synchronization in Orca is based on operations that block. As an example, a process executing the statement:

```
x := myqueue$remove_head();
```

suspends until `myqueue` is not empty. If the queue is initially empty, the process waits until another process appends an element to the queue. If `myqueue` contains only one element and several processes try to execute the statement simultaneously, only one process will succeed in calling `remove_head`. Other processes will suspend until more elements are appended to the queue.

In summary, processes communicate and synchronize via objects that are passed as call-by-sharing parameters when processes are created. Call-by-sharing is restricted to indivisible objects; other data can only be passed by value.

The distribution of objects among the participating processors is left entirely to the implementation. This decision significantly contributes to the simplicity of the language. The design of Orca allows the compiler and run time system to deal efficiently with the distribution of objects. There are no global objects (objects have to be passed as parameter), so the run time system can keep track of which processes can access which objects. The semantics of an operation invocation do not depend on whether the object and the invoker are on the same or different processors. This location independence makes it possible to move objects dynamically. Shared data can only be accessed via a well-defined set of operations. This enables the system to dynamically *replicate* objects. An operation can access only a single object, allowing indivisible operations to be implemented without using a complicated locking or version management scheme.

## 3. DISTRIBUTED PROGRAMMING IN ORCA

The shared data-object model described above is very general and supports several different programming styles. To illustrate this, we will show how the model can simulate shared variables, build distributed data structures, and emulate explicit message passing constructs.

As an introductory example, consider the specification of the abstract data type *IntObject* shown in Figure 3.

```
object specification IntObject;
   operation value(): integer;   # return current value
   operation assign(val: integer);  # assign new value
   operation min(val: integer);
      # set value to minimum of current value and ‘‘val’’
   operation max(val: integer);
      # set value to maximum of current value and ‘‘val’’
   ...
end;
```

**Figure 3**: specification of object type IntObject.

Instances (objects) of this type can be created and distributed among other processes. Such an object effectively becomes a shared integer variable, with several operations to read or change its value. The operations (`value`, `assign`, etc.) are all indivisible. If two processes simultaneously invoke `X$min(A)` and `X$min(B)`, the new value of X is the minimum of A, B, and the old value of X. On the other hand, a sequence of operations, such as

```
if A < X$value() then
     X$assign(A);
fi;
```

is not indivisible. This rule for defining which actions are indivisible and which are not is both easy to understand and flexible: single operations are indivisible, sequences of operations are not. The set of operations can be tailored to the needs of a specific application.

Programming with distributed data structures has already been studied extensively by Carriero et al., using the language Linda [Carriero et al. 1986]. An interesting example is the *replicated worker* model, which structures a program as a collection of replicated worker processes (one per processor) that repeatedly take a work item from a *taskbag*, perform the work, and (possibly) generate some more work. The taskbag is implemented as a distributed data structure, accessible by all workers.

the design of a distributed Traveling Salesman Problem (TSP)* algorithm, based on an earlier algorithm described in [Bal et al. 1987]. The algorithm uses one process to generate partial routes for the salesman (containing only part of the cities) and any number of worker processes to further expand (search) these partial solutions. A worker systematically generates all full routes that start with the given initial route, and checks if they are better (shorter) than the current best solution. Every time a worker finds a shorter full route, it updates a variable shared by all workers, containing the length of the shortest route so far. This variable is used to cut-off partial routes that are already longer than the current shortest route, as these will never lead to an optimal solution. The shared variable is implemented as an object of type IntObject (see Figure 2). As several workers may simultaneously try to decrease the value of this variable, it is updated using the indivisible `min` operation.

The work-to-do is stored in an ordered taskqueue, the order being determined by one of the many heuristics that exist for the Traveling Salesman Problem (e.g., nearest-city-first). The taskqueue is similar to the `IntQueue` data type of Figure 1, except that the elements are ‘‘routes’’ rather than integers. (Orca provides generic objects, to express this conveniently.) The basic algorithm for the worker processes is outlined in Figure 4. (Figure 4 does not show how termination of the worker processes is dealt with; this requires an extension). Conceptually, the distributed algorithm is as simple as the sequential TSP algorithm.

```
process worker(minimum: shared IntObject; q: shared taskqueue);
      r: route;
begin
      do  # forever
          r := q$remove_head();
          tsp(r, minimum);
      od;
end;

function tsp(r: route; minimum: shared IntObject);
begin
      # cut-off (partial) routes longer than the current best one
      if length(r) < minimum$value() then
           if ‘‘r’’ is a full solution (covering all cities) then
                # r is a full route shorter than the current best
                # route, so update the current best solution.
                minimum$min(length(r));
           else
                for all cities ‘‘c’’ not on route ‘‘r’’ do
                       # search route r extended with c
                       tsp(r||c, minimum);
                od;
           fi;
      fi;
end;
```

**Figure 4**: Algorithm for TSP worker processes.

* The Traveling Salesman Problem is the problem of finding the shortest route for a salesman to visit each of a number of cities in his territory exactly once.

Shared data objects can also be used to construct lower level message passing primitives. Asynchronous message passing, for example, can be specified using a message-queue data type, similar to the `IntQueue` data type of Figure 1. A message is sent by appending it to the queue and received using a statement like:

```
msg := msgqueue$remove_head();
```

Abstract data types of general use can be collected in a library, thus building a standard environment for the language. In this way, programmers can access a lot of useful primitives, but the language itself is kept simple.

## 4. DISCUSSION

Our technique extends the abstract data type model to distributed systems. The resulting language uses a minimum of features to support distributed programming. Abstract data types also have been used for several other parallel and distributed languages, especially monitor-based and object-oriented languages.

Concurrent Pascal [Brinch Hansen 1975] uses monitors to encapsulate data shared by multiple processes. Traditionally, monitors have been used for single-processor systems or for multiprocessors with shared memory. Monitors can also be used in a distributed system, by putting the data on one specific processor and invoking the operations as remote procedure calls. This approach, however, is not truly distributed, as data are still centralized. Process synchronization with monitors is based on *condition variables*, using an operation (*wait*) to suspend on a ''false'' variable, and another one (*signal*) to inform suspended processes that the variable has become ''true.'' Orca intentionally lacks such a feature. If an Orca process is blocked in an operation, the operation's guards are re-evaluated whenever the object is changed. Although adding a *signal*-like primitive could increase efficiency, it would not be in the spirit of keeping the language design as simple as possible.

Emerald [Black et al. 1987] is an object-oriented language, which considers all entities to be objects. Objects in Emerald can be active as well as passive (data). Objects can dynamically move from one processor to another, either under program or system control. By default, the compiler and run time system decide where objects are stored. To increase efficiency, the programmer can move objects around. Unlike Orca, Emerald does not entirely hide locations of objects from the programmer.

Several other languages and systems also provide logically shared data. One interesting example is Linda's Tuple Space [Ahuja et al. 1986]. Orca differs primarily by allowing programmers to define operations of arbitrary complexity on shared data structures. Linda supports a fixed number of low-level primitives for manipulating single tuples, which we feel is a disadvantage [Kaashoek et al. 1989a].

Many designers of distributed operating systems have also proposed or implemented abstract shared memory models. Examples are: Cheriton's Problem-oriented Shared Memory [Cheriton 1985], Li's Shared Virtual Memory [Li and Hudak 1986], the Agora Shared Memory [Bisiani and Forin 1987], and Mirage [Fleisch and Popek 1989]. In these systems, the unit of sharing typically is a physical page, rather than a logical, programmer defined entity (as in our model). For a comparison of several of these systems, we refer the reader to [Bal and Tanenbaum 1988].

We have built a compiler and three prototype run time systems for Orca. One RTS runs on a multiprocessor system of MC68020s connected by a VME bus. The second RTS runs on top of the Amoeba [Mullender and Tanenbaum 1986] distributed operating system.

The third RTS runs on the bare hardware, a collection of MC68020s connected by an Ethernet [Bal et al. 1989a]. This RTS uses an efficient reliable broadcast protocol [Kaashoek et al. 1989b] for updating all copies of an object. The replication techniques used for the two distributed RTSs are discussed in [Bal et al. 1989b].

The implementations have been in use for over a year now. Orca has been used for several parallel applications, including chess problem solving [Elias 1989], branch-and-bound, graph algorithms, matrix algorithms, and numerical algorithms [Bal 1990; Bal et al. 1990].

## 5. ACKNOWLEDGEMENTS

The authors would like to thank Wim van Leersum for implementing the Orca compiler and Robert-Jan Elias and Jack Jansen for implementing the distributed run time systems.

## 6. REFERENCES

Ahuja, S., Carriero, N., and Gelernter, D., Linda and Friends, *IEEE Computer*, Vol. 19, No. 8, pp. 26-34, Aug. 1986.

Bal, H. E., Programming Distributed Systems, Silicon Press, Summit, NJ, 1990.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., A Distributed Implementation of the Shared Data-object Model, *USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1-19, Ft. Lauderdale, FL., Oct. 1989a.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., Experience with Distributed Programming in Orca, *IEEE CS Int. Conf. on Computer Languages*, New Orleans, Louisiana, Mar. 1990.

Bal, H. E., Kaashoek, M. F., Tanenbaum, A. S., and Jansen, J., Replication Techniques for Speeding up Parallel Applications on Distributed Systems, Report IR-202, Vrije Universiteit, Amsterdam, The Netherlands, Oct. 1989b.

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S., Programming Languages for Distributed Computing Systems, *ACM Computing Surveys*, Vol. 21, No. 3, pp. 261-322, Sep. 1989c.

Bal, H. E. and Tanenbaum, A. S., Distributed Programming with Shared Data, *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, pp. 82-91, Miami, FL, Oct. 1988.

Bal, H. E., Van Renesse, R., and Tanenbaum, A. S., Implementing Distributed Algorithms Using Remote Procedure Calls, *Proc. AFIPS Nat. Computer Conf.*, Vol. 56, pp. 499-506, AFIPS Press, Chicago, IL, June 1987.

Bisiani, R. and Forin, A., Architectural Support for Multilanguage Parallel Programming on Heterogenous systems, *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 21-30, Palo Alto, CA, Oct. 1987.

Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L., Distribution and Abstract Types in Emerald, *IEEE Trans. Softw. Eng.*, Vol. SE-13, No. 1, pp. 65-76, Jan. 1987.

Brinch Hansen, P., The Programming Language Concurrent Pascal, *IEEE Trans. Softw. Eng.*, Vol. SE-1, No. 2, pp. 199-207, June 1975.

Carriero, N., Gelernter, D., and Leichter, J., Distributed Data Structures in Linda, *Proc. 13th ACM Symp. Princ. Progr. Lang.*, pp. 236-242, St. Petersburg, FL, Jan. 1986.

Cheriton, D. R., Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems, *Oper. Syst. Rev.*, Vol. 19, No. 4, pp. 26-33, Oct. 1985.

Elias, R-J., Oracol, A Chess Problem Solver in Orca, Master thesis, Vrije Universiteit, Amsterdam, The Netherlands, July 1989.

Fleisch, B. D. and Popek, G. J., Mirage: A Coherent Distributed Shared Memory Design, *Proc. of the 12th ACM Symp. on Operating System Principles*, pp. 211-223, Litchfield Park, AZ, Dec. 1989.

Kaashoek, M. F., Bal, H. E., and Tanenbaum, A. S., Experience with the Distributed Data Structure Paradigm in Linda, *USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 175-191, Ft. Lauderdale, FL., Oct. 1989a.

Kaashoek, M. F., Tanenbaum, A. S., Flynn Hummel, S., and Bal, H. E., An Efficient Reliable Broadcast Protocol, *Operating Systems Review*, Vol. 23, No. 4, pp. 5-20, Oct. 1989b.

Li, K. and Hudak, P., Memory Coherence in Shared Virtual Memory Systems, *Proc. 5th Ann. ACM Symp. on Princ. of Distr. Computing*, pp. 229-239, Calgary, Canada, Aug. 1986.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., Abstraction Mechanisms in CLU, *Commun. ACM*, Vol. 20, No. 8, pp. 564-576, Aug. 1977.

Mullender, S. J. and Tanenbaum, A. S., Design of a Capability-Based Distributed Operating System, *Computer J.*, Vol. 29, No. 4, pp. 289-299, Aug. 1986.