A Simple Model of Distributed Programs Based on Implementation-Hiding and Process Autonomy

by K. M. Chandy and J. Misra

Department of Computer Sciences
University of Texas at Austin

TR-127 January 1980

This work is partially supported by AFOSR AF77-3409.

A SIMPLE MODEL OF DISTRIBUTED PROGRAMS BASED ON IMPLEMENTATION-HIDING AND PROCESS AUTONOMY*

K. M. Chandy and J. Misra
University of Texas at Austin, Austin, TX 78712

ABSTRACT

This paper presents a model for a network of communicating processes. We extend well known ideas in sequential programming such as procedures, parameter passing and binding, and recursion to distributed programs. We stress the notion of implementation-hiding, i.e. the invoker of a process or procedure has no knowledge of the implementation of the invoked computation.

1. INTRODUCTION

There are two approaches to distributed programming: one can attempt to develop the most general, most powerful, and often least understood mechanisms or one can develop simple easily understood extensions to sequential programming models. We take the latter approach. We present a model of parallel programming based on message communication. Our emphasis is on fundamental conceptual issues in parallelism and message communication. We are not proposing a complete language; however we are proposing constructs on which a language can be based.

A reasonable set of objectives for parallel programming is:

1.1 Generalization of sequential programming

There has been a great deal of investment in the design, specification and proofs of sequential programs; indeed an entire discipline has developed in recent years. Distributed programming must make maximum use of sequential programming concepts and techniques. Distributed programming models, tools and methodologies should be developed as simple, natural extensions of their sequential programming counterparts so that a substantial reinvestment in the new technology is avoided.

A distributed programming language ought to be derived from a sequential programming language with the addition of a minimum number of new features. Simplicity is the critical concern.

We extend the concept of procedures to include networks of processes. The concepts of parameter passing and binding,procedure invocation and recursion are generalized to distributed programs.

1.2 Process autonomy

Each component of the distributed program should be designed and proved independent of the rest of the program. In our model the goal of process autonomy is achieved; in particular, a process may not name another process.

* Work partially supported by AFOSR AF77-3409.

1.3 Hierarchical Proofs

It should be possible to prove properties of distributed programs from properties of the externally observable behavior of component processes [5].

1.4 Ability to guarantee determinism

Message communication systems are usually inherently non-deterministic. However a programmer may want to guarantee that his program is determinate: for example he may want to ensure that the sequence of messages output is a function of the sequence of messages input. A language must have simple constructs, which if used, guarantee determinism of computation; we present such a construct.

1.5 Referential transparency - implementation hiding

A computation may be implemented as a sequential or parallel program. The invoker of a computation should not be aware of how the computation is being carried out. A computation is specified merely as a relationship between its inputs and outputs. Information hiding is an accepted notion in sequential programming. The natural extension of information hiding to distributed programming is implementation hiding: the implementation of a computation is hidden from the invoker.

In our model the invoker of a process or procedure has no knowledge of how the invoked process is implemented.

2. FUNDAMENTAL CONCEPTS IN DISTRIBUTED PROGRAMMING

2.1 Communication with a computation: external view of a computation

The external view of a computation generally takes two forms.

(a) A computation may be "called" as in a procedure call; in this case the invoker is suspended until the invoked computation terminates. This form of computation is traditionally called a procedure. Communication between the caller and called computations is via parameters.

(b) Messages may be passed between computations. In this case the message does not necessarily invoke a new computation. This form of computation is called a process. A process sending a message is not necessarily suspended while the receiver process is computing [1,2]. The distinction between a procedure and a process is that procedures are "called" while messages are passed to/from processes.

2.2 Implementation of a computation: internal view of a computation

Traditionally, computations have been implemented as sequential programs (which may include procedure calls); the procedures are themselves implemented as sequential programs. Implementation of computations using messages (i.e. sequential processes) was suggested later (see [2] for a review).

2.3 Referential transparency of computation: implementation-hiding in distributed programs

The implementation of a computation is of no logical consequence to the invoker of the computation. This implies that the specification of a computation must define only the relationships between its inputs and outputs. A consequence of referential transparency is that a computation may be implemented either as a sequential program or as a distributed program. In particular, a procedure may be implemented either as a conventional sequential procedure or as a network of communicating processes. Similarly a process may be implemented as a sequential process or as a set of communicating processes.

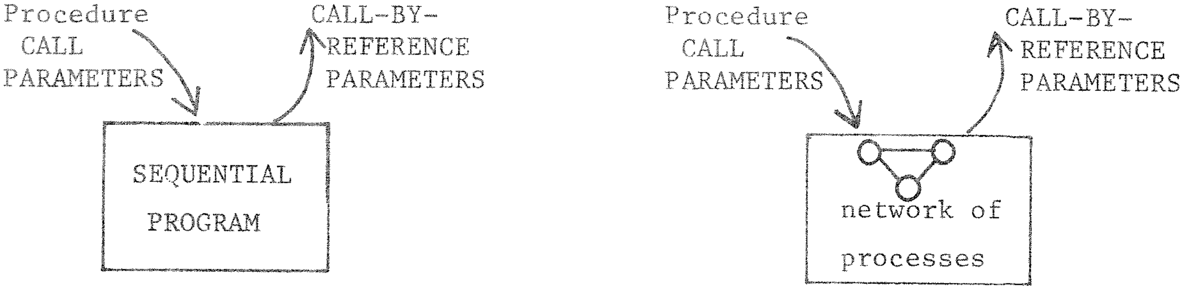
Procedure CALL PARAMETERS
CALL-BY-REFERENCE PARAMETERS
SEQUENTIAL PROGRAM
Procedure CALL PARAMETERS
CALL-BY-REFERENCE PARAMETERS
network of processes

Fig 1. Equivalent external view of a procedure: the implementation is hidden

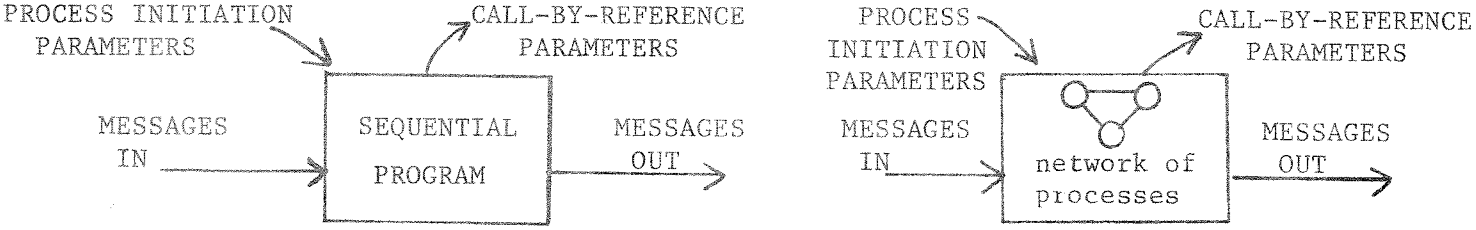
PROCESS INITIATION PARAMETERS
CALL-BY-REFERENCE PARAMETERS
MESSAGES IN
SEQUENTIAL PROGRAM
MESSAGES OUT
PROCESS INITIATION PARAMETERS
CALL-BY-REFERENCE PARAMETERS
MESSAGES IN
network of processes
MESSAGES OUT

Fig 2. Equivalent external view of a process: the implementation is hidden

2.4 Process autonomy

The definition of a process must allow it to be used in different contexts. Hence a process should not name other external processes. A process may make assumptions only about the sequences of messages it receives: we therefore use the concept of external variables.

2.5 External variables

A process declaration names certain local variables which can receive values from external processes or whose values may be transmitted to external processes. The former kind of variable is called an external input variable and the latter kind is called an external output variable. An external input or output variable is local to the process in which it is defined.

Input statements in any process h have the form

```
x:=?,
```

and output statements have the form

```
?:=y,
```

where x and y are external input and output variables (respectively) of h.

Each output (input) variable of a process is bound to exactly one input (output) variable of a process. (Only variables of the same type can be bound together.) The declaration of the binding will be described later. The binding is external to the processes. Let x be an output variable of process h_1 and let x be bound to y, an input variable of process h_2. Then h_1 will wait at an output statement

```
?=:x;
```

until control in h_2 reaches a corresponding input statement

```
y:=?;
```

A message transmission may take place only after h_1 reaches the output statement and h_2 reaches the corresponding input statement. h_1 and h_2 will both complete executions of their corresponding input and output statements simultaneously when the message transmission is over; at this point y in h_2 has the value of x in h_1. Of course the value of x in h_1 is unchanged by the message transmission. This protocol for message transmission has been suggested by Hoare [ ].

2.6 External view of a process/procedure

A process P interacts with its environment through one or more of the following:

(1) Call-by-value parameters passed to a process from its environment at process initiation; these parameters are treated as constants in the process body.

(2) Call-by-reference parameters passed to a process from its environment at process initiation and returned to the environment at process termination.

(3) External input variables.

(4) External output variables.

A process is specified by the relationship between the above parameters and variables.

The external view of a procedure is identical to that of a process except that there can be no external variables.

2.7 Binding

Processes may be constructed hierarchically: a process P may be defined to consist of several component processes $Q_1,\ldots,Q_n$. The construction of P from $Q_1,\ldots,Q_n$ must specify (a) the component processes $Q_1,\ldots,Q_n$ and (b) the following four relationships among P and $Q_1,\ldots,Q_n$.

(1) Distribution of call-by-value parameters. The call-by-value parameters of P may be distributed among the component processes $Q_1,\ldots,Q_n$, i.e. a call-by-value parameter v of P may be passed as a call-by-value parameter to any number of component processes.

(2) Partitioning of call-by-reference parameters. The set of call-by-reference parameters of P is partitioned among the component processes, $Q_1,\ldots,Q_n$, i.e. every call-by-reference parameter of P must be passed to exactly one component process Q_i, as a call-by-reference parameter.

(3) The connection between the external variables of the component processes $Q_1,\ldots Q_n$, and

(4) The relationship between the external variables of P and the external variables of $Q_1,\ldots,Q_n$.

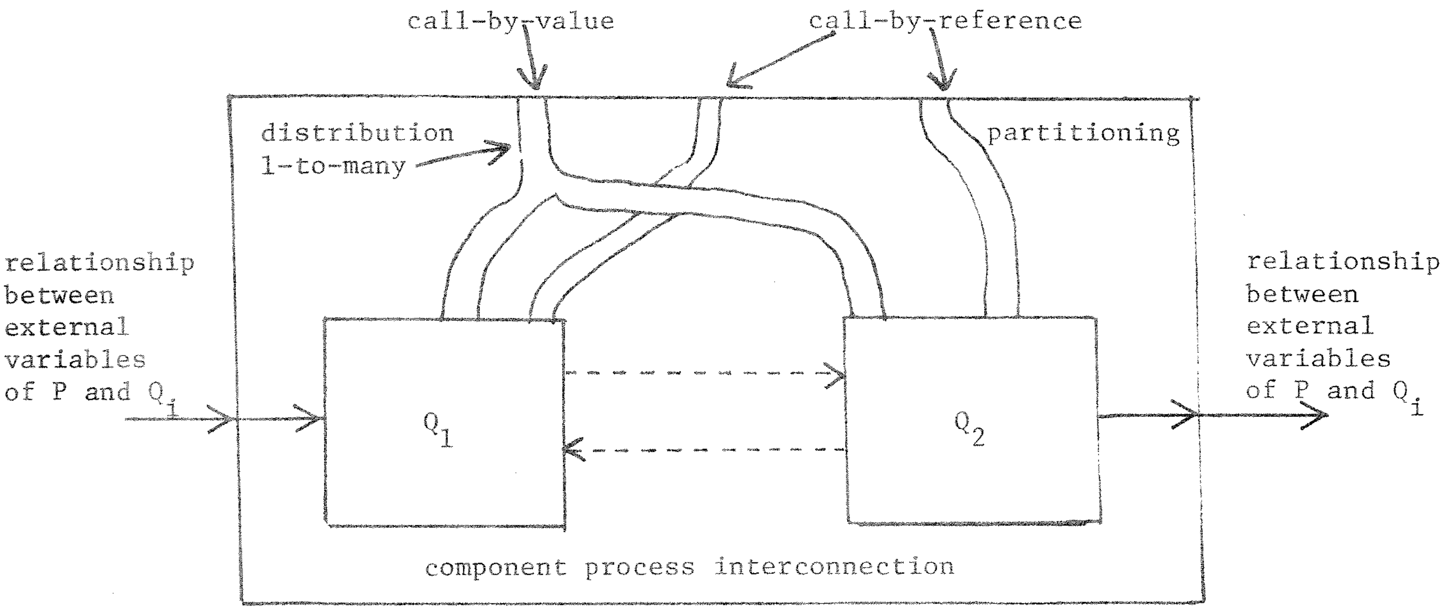
call-by-value
call-by-reference
distribution
1-to-many
partitioning
relationship between external variables of P and Q_i
Q_1
Q_2
relationship between external variables of P and Q_i
component process interconnection

Fig 3: Binding of a process P with component processes Q_1 and Q_2

These four relationships are collectively called the binding of $Q_1,\dots,Q_n$ to form P.

A procedure P may be constructed from component processes $Q_1,\dots Q_n$. The binding of $Q_1,\dots,Q_n$ to form procedure P is identical to the case where P is a process except that there can be no external variable of a procedure.

Binding allows processes to be defined autonomously and also allows implementation-hiding. Binding is a key concept. A binding is static, i.e. network topology cannot be changed during its lifetime.

2.8 Process operation

2.8.1 Instantiation

A process or a procedure can be instantiated only as a consequence of a call to some procedure. When a procedure P is called it is instantiated: instantiation of a procedure P is defined to be the instantiation of its component processes $Q_1,\dots,Q_n$, if any, and the implementation of the binding (if any, between $Q_1,\dots,Q_n$ and P) declared in P. Similarly, instantiation of a process Q_i is defined to be the instantiation of its component processes and the implementation of the binding of Q_i.

2.8.2 Termination

A sequential process or procedure terminates when it completes execution of statements in its body. A hierarchical process or procedure (i.e. one with a binding section) terminates when all of its component processes terminate.

Communication with a terminated process will be implemented as an indefinite wait as in [4]. This implies that normally a process will have to send explicit termination signals to processes wishing to communicate with it.

2.9 Determinate and indeterminate constructs for parallel waiting

It is crucial for absence of deadlock that a process have the ability to wait simultaneously on several external variables. Even though a process may wait in parallel for messages, the actual transmission of messages will be assumed to occur in sequence.

It is important for a programmer to be able to guarantee that his program is deterministic [6]. A program is deterministic if the sequence of values assigned to each and every variable in the program depends only upon the inputs to the program. We have two forms of parallel waiting in our model, 1) to give programmers the ability to guarantee determinism and 2) to allow programmers to choose nondeterminism.

2.9.1 Deterministic i/o command

This command consists of one or more elementary i/o statements which may be executed in arbitrary order. For example,

```
[x:=?, ?:=y, z[i]:=?]
```

denotes that inputs will be received on x and z[i] and the value of y will be output in some arbitrary order. The variables named in the command must be distinct. (This rule cannot be enforced by a compiler if there are subscripted variables.) The i/o command completes only when all elementary i/o statements within the command complete. If all other constructs in a language are deterministic, the inclusion of the deterministic i/o command will preserve determinism because at the instant at which an i/o command terminates, the values of all variables named in the deterministic i/o command are independent of the sequence in which the elementary i/o commands are executed.

2.9.2 Guarded commands

Our model includes guarded commands as in [1]. The use of this feature by any process results in potential nondeterminism. Note however that unlike Hoare's model, a guarded command in process h which has an input statement in the guard for communication with process g, cannot fail merely because g has terminated.

2.10 Recursion

Procedures may be written using recursion even though a procedure may be implemented as a network of processes. For example, a procedure P may consist of processes $Q_1,\ldots,Q_n$, and any component process Q_i may call P resulting in the initiation of a fresh instance of procedure P.

2.11 Other issues

2.11.1 Process parameterization

Value parameters are treated as constants and may be used to parameterize a process; for example, an array of external variables or component processes A[1..n] may be declared where n is passed as a value parameter.

2.11.2 Scope rules

A process or a procedure can only access the following names 1) formal parameters and 2) local variables and 3) names of other processes/procedures in the same block as its outer blocks as in Algol-60.

2.11.3 Explicit description of network topology

The topology of a network of communicationg processes can be represented by a labeled directed graph in which each process is represented by a vertex. An edge from a vertex representing a process P to a vertex representing a process Q is labeled x at its head and y at its tail if (1) y is an external output variable of P and x is an external input variable of Q and (2) these variables are bound (as specified in the binding). Note that there could be multiple edges with distinct labels between the processes.

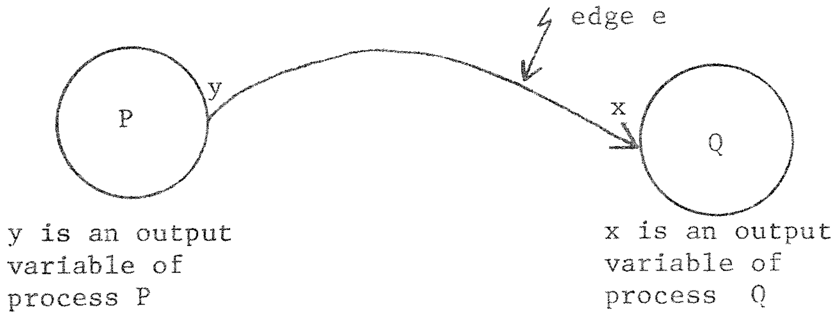
edge e
y
P
x
Q
y is an output variable of process P
x is an output variable of process Q

Fig 4: Edge e represents the binding of external variables x and y of process Q and P (respectively)

The definition of a network of communicating processes appears in two distinct sections: (1) a definition of the internal computation of each component process and (2) the definition of the structure of process interaction (i.e. the labeled graph) as in Fig. 4. Properties related to structure can be derived from the structural definition (in the binding section). Our model provides an explicit topological description which is useful in understanding network behavior.

2.11.4 Termination of a process/procedure

We have adopted the rule that a process/procedure terminates only when all component processes terminate. Since the effect of computation is determined solely by call-by-reference parameters, it is sufficient to run the computation until processes which have call-by-reference parameters have terminated.

2.11.5 Dynamic Network Topology

It may be convenient to have dynamic binding, i.e. to allow changes in communication paths during the computation. One possibility is to consider a supervisory binding process that runs concurrently with other component processes and modifies the bindings during the operation. Since it is extremely difficult to prove properties of such a computation, we favor the static approach outlined in this paper.

We may allow a process to dynamically equivalence one of its external input variables with one of its external output variables of the same type: this means that from that point onwards in the computation the process behaves as a "short circuit" for these variables transmitting the input directly to the output. A process may also cancel an equivalence. This feature allows a limited amount of dynamic binding while retaining process autonomy; however it makes proofs about computation more difficult.

2.12 Summary

We have evolved a model of distributed programming from key ideas in sequential programming. We feel that an evolutionary approach is preferable to the development of a radically new method for distributed programming. We extend well known ideas in sequential programming such as procedures, parameter passing, recursion and data types. Binding is merely a generalization of parameter passing. Sequential processes are merely sequential programs with message communication primitives. The only way in which concurrent computations can be initiated is by procedure call. Thus the number of concepts introduced solely for distributed programming is kept to a bare minimum.

We have developed an axiomatic approach to proving programs based on this model [5]. Our approach is a natural extension of axiomatic sequential programming techniques. It allows for the hierarchical development of proofs, i.e. a proof of a process may be derived in a natural manner from the proofs of its component processes. Our insistence on process autonomy results in simple proofs of the harmonious behavior of the component processes.

We have presented a necessary and sufficient condition for absence of deadlock of programs based on our model [4]. Its application has led to proofs much simpler than those appearing in the literature.

References

1. C.A.R. Hoare, Communicating Sequential Processes, CACM (21) 8, August 1978.

2. G. Kahn, The Semantics of a Simple Language for Parellel Programming, in Proc. IFIP Congress 74, North Holland , 1974.

3. W. A. Wulf, Languages and Structured Programs, in Current Trends in Programming Methodology (ed. Yeh), Prentice Hall, 1977.

4. K. M. Chandy and J. Misra, Deadlock Absence Proofs for Networks of Communicating Processes, Information Processing Letters (9) 4, Nov. 1979, pp 185-189.

5. J. Misra and K. M. Chandy, An Axiomatic Proof Technique for Networks of Communicating Processes, Technical Report, Computer Sciences Department, University of Texas.

6. R. E. Bryant and J. B. Dennis, Concurrent Programming in Research Directions in Software Technology (ed. Wegner), MIT Press, Cambridge, Mass. 1979.