Programming Language Support for Multicast Communication in Distributed Systems

Eric C. Cooper

May 1990
CMU-CS-90-121 3

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Internet: ecc@cs.cmu.edu

Tenth International Conference on Distributed Computing Systems (ICDCS-10)
Paris, France
May 28–June 1, 1990

Keywords: programming languages, multicast communication, distributed systems

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract number N00039-87-C-0251, and monitored by the Space and Naval Warfare Systems Command. (ARPA Order No. 5993)

Abstract

Multicast or group communication is an important part of modern distributed systems, but programming language support for such communication is uncommon. Remote procedure call uses a familiar programming language abstraction to support unicast request-response communication; what should the corresponding abstraction be for multicast communication?

The essential and desirable properties of a language construct for multicast communication are presented first. Essential properties include type safety, expressive power, and efficiency. Desirable properties include use of familiar control and data structures, appropriate semantic level, and first-class treatment of multicast operations in progress.

The main contribution of the paper is the introduction of a spectrum of abstractions for multicast communication, in increasing order of both desirability and semantic level: functional mapping, iterators, and streams. Examples of distributed algorithms from the literature are used to illustrate the expressive power of each mechanism. Streams in particular provide first-class status for multicast communication in progress, and can be implemented efficiently in typical multicast communication architectures.

1 Introduction

Experience with remote procedure call (RPC) has demonstrated the importance of language-level support for communication in distributed systems. The use of a familiar, type-safe language construct—in this case, the procedure call—greatly simplifies the task of building distributed systems.

Multicast communication plays a natural role in many distributed algorithms, particularly those involving replication for high availability [4, 7, 8]. Low-level multicast communication is available in many distributed system architectures. At the media access control and data link layers, the IEEE 802 family (including Ethernet, Token Ring, and FDDI) all support multicast transmission [15]. At the network layer, the DoD Internet supports multicast IP datagrams [6]. At the transport layer, a number of multicast extensions to request-response protocols have been investigated, including Circus [4], MultiRPC [14], and VMTP [1].

Distributed applications need multicast communication, and the underlying communication architectures provide it, but programming language support for multicast communication is missing. Support for remote procedure call, on the other hand, is widespread. One of the reasons for this successful integration is hardly surprising—the natural language-level abstraction for unicast request-response communication is the familiar procedure call. What should the corresponding abstraction be for multicast request-response communication?

The remainder of the paper is devoted to answering this question. After a brief survey of related work, we introduce *essential* and *desirable* properties for language-level multicast communication; these provide the basis for evaluating proposed mechanisms. We then define three language constructs—functional mapping, iterators, and streams—that possess increasingly larger sets of these characteristics. Streams in particular provide first-class status for multicast communication in progress, and can be implemented efficiently in typical multicast communication architectures.

2 Related Work

The present work is similar in spirit to Liskov's and Shrira's work on *promises*, a linguistic abstraction of asynchronous RPC [10]. Both approaches are "bottom up", driven by the appearance of new transport protocols and the desire to incorporate them at the programming language level.

The author's previous work on replicated procedure call [3] introduced multicast request-response communication and identified the need for *collators* (functions that map multiple values into a single value) but proposed no linguistic support for these mechanisms. The author's dissertation [4] suggested the use of *iterators* [11] as a language construct for replicated procedure calls; that suggestion is reconsidered in the present work.

A multicast request-response protocol is used in the V system for communication with process groups [2], and has been incorporated into VMTP [1]. No description of language support for these mechanisms has appeared in the literature, although a stub generator for VMTP has recently been completed.

StarMod [9] provides linguistic support for *broadcast messages* by returning an array of replies instead of single value. Since this technique does not allow early completion, it is not as expressive as the abstractions proposed here.

MultiRPC [14] uses functional mapping over multiple results; this technique, described in more detail in Section 5.1, represents the starting point for the abstractions described in this paper.

The *cstub* stub generator [12] supports parallel remote procedure calls by adding a limited form of iterators to C and C++. This approach is discussed further in Section 5.2.

3 Criteria for Multicast Support

What properties should a language-level abstraction of multicast communication possess? Following Nelson [13], we separate essential characteristics from desirable ones.

3.1 Essential Properties

- Type safety is essential. The mechanism must not compromise the type system of the language, and so must potentially be as type safe as any modern programming language.

- Expressive power is essential. For example, multicast communication is often used in situations where the caller needs only the first *n* replies before it can continue; the mechanism must allow this kind of early completion. Even if all replies are processed, their temporal order may be important; the mechanism must make this visible.

- Efficiency is essential. The mechanism must map onto the underlying transport protocol with a minimum of overhead.

3.2 Desirable Properties

- Use of familiar control and data structures is desirable. If possible, the mechanism should use structures whose syntax and semantics are already well understood.

- Matching the semantic level of the language is desirable. Adding Scheme-style continuations to C solely to support multicast communication, or adding low-level asynchronous interrupt handlers to a language designed around concurrent processes, would hardly be appropriate.
- First-class status is desirable. To the extent possible, a multicast operation in progress (say, after some but not all of the replies have been received) should be represented as a first-class value that can be passed to other routines, stored in data structures, and so on.

4 System Model

This section describes the layers of a typical distributed system that are relevant to language support for multicast communication.

4.1 Transport Layer

We adopt a model of multicast communication at the transport layer based on the V primitives [2]:

- *MulticastRequest* is used by a client to send a multicast request to a group of servers.
- *GetRequest* and *SendReply* are used by a server to receive a request and send a response. The server handles multicast requests in the same way it handles unicast requests.
- *GetReply* is used by a client to receive the next reply to the current request.
- *AllReceived?* is used by a client to test whether all of the replies have been received. As long as this returns *false*, it is meaningful to call *GetReply*.

In the V model, a *MulticastRequest* has the side effect of discarding all unread replies to the previous request, so no explicit operation is required at the transport layer to support early completion.

When a remote operation returns no value, we assume that the underlying transport protocol still acknowledges its completion. This allows the client to use *GetReply* (which returns an empty message in this case) for synchronization purposes, to block until the request has been completed.

The *AllReceived?* operation is useful only in systems like Circus and MultiRPC, in which the size of groups is known by the transport layer. In a system like V, *AllReceived?* always returns *false*, and the client must know how many *GetReply* operations to perform. This distinction is reflected in the use, but not the design, of the language-level mechanisms presented in this paper: in a V-like system, the client must explicitly exit from the various iteration constructs; in a Circus-like system, the iteration will always terminate.

4.2 Session Layer

The issue of binding a client to a group of servers for one or more calls is completely orthogonal to the issues addressed in this paper. The same approaches possible with conventional RPC are possible with group calls:

- An explicit *BindGroup* operation can be used to bind all calls to an interface to a particular group of servers.
- The stub generator can add an extra parameter to each stub procedure, to allow the group of servers to be specified in each call.
- The programming language itself can support a syntax such as *Group . Proc*( ), where *Group* is a variable denoting a group of servers exporting *Proc*.

4.3 Presentation Layer and Stub Generation

Throughout this paper, we assume that a *stub generator* is used to hide the details of transmitting and receiving typed data. A typical stub generator transforms type declarations (such as array and record types) into procedures for transmitting and receiving values of those types, thus bridging the gap between the typed objects of a programming language and the untyped messages of a transport protocol. In the pseudo-code that follows, we will use the statement

```
msg : Message := Marshal (P, arg)
```

to indicate marshaling of procedure name and arguments into a request message for the transport layer, and the statement

```
reply : T := Unmarshal (msg)
```

to indicate unmarshaling of the results contained in a response message.

The stub generator approach is also an effective means of integrating the *control structures* inherent in a particular communication paradigm into an existing programming language. In the case of RPC, for example, the stub generator produces stub routines for each procedure in a remotely callable interface. On the client side, these stubs take care of transmitting the procedure and its arguments to the server, and then receiving the results. On the server side, the stub generator typically produces a top-level loop that repeatedly receives an incoming call, invokes the appropriate procedure, and sends back the results.

For each of the language constructs for multicast communication proposed in this paper, we will indicate how the stub generator approach can be extended to handle that mechanism. Typically, only the client side must be extended; each server in the multicast group handles what appears to be a conventional remote procedure call. The notation *P** will be used consistently to indicate a client stub that implements a multicast call to a procedure *P*.

4.4 Application Layer

We will use an example application, the two-phase commit protocol [8], to illustrate the expressive power of the approaches presented in Section 5. Simplified code for this application will be shown using each form of multicast communication.

The two-phase commit algorithm involves a coordinator and a group of participants, each of which export the types and procedures shown in simplified form in Figure 1. During the first phase of the algorithm, the coordinator must ascertain that all the participants are ready to commit; if any are not, the transaction is aborted at all sites. The abort decision can be made as soon as any participant indicates that it is not willing to commit the transaction; this is an example of early completion. Otherwise, the transaction is committed at all sites in the second phase.

```
status : type = (commit, abort)

Ready : proc ( ) return status
Commit : proc ( ) return void
Abort : proc ( ) return void
```

Figure 1: Two-phase commit participant interface

5 A Spectrum of Language Constructs for Multicast Communication

We are now in a position to answer our original question of what constitutes the appropriate language-level abstraction for multicast communication, by introducing three candidate mechanisms. Each of them has all of the essential characteristics defined in Section 3.1, but they possess increasingly larger sets of the desirable characteristics. Along with this increasing "desirability", however, goes a corresponding increase in "semantic level". As a result, the final mechanism (streams) is not necessarily to be preferred over the initial mechanism (functional mapping); rather, that choice also depends on the language to which the mechanism is added. The relationship between the multicast facility and the underlying programming language is discussed further in Section 6.

```
P* : proc (arg : A ; fn : proc (reply : T) return boolean) return void is
  msg : Message := Marshal (P, arg)
  reply : T
begin
  MulticastRequest (msg)
  while not AllReceived? ( ) do
      msg := GetReply ( )
      reply := Unmarshal (msg)
      if not fn (reply) then
        exit
      end if
  end while
end P*
```

Figure 2: Stub routine using functional mapping

5.1 Functional Mapping

The simplest mechanism that meets the essential criteria is functional mapping. In this scheme, a user-specified function is applied to each reply value, in order of arrival. It can be added to almost any programming language.

The type safety requirement is met by introducing a stub procedure that takes a properly typed user function as an additional parameter. Using a stub generator for group calls, a procedure *P* of type

P : **proc** (arg : A) **return** T

would be translated into a stub procedure *P** with signature

P* : **proc** (arg : A ; fn : **proc** (reply : T) **return** boolean) **return** void

Expressive power is provided by applying the user function to the replies in the order in which they arrive. Early completion can be supported in either of two ways. The simplest is to use the value returned by the user-supplied function (a boolean, say) to determine when the mapping should cease. This approach is used in MultiRPC [14], and is shown in detail in Figure 2. The alternative is to use some form of nonlocal transfer of control—an exception, Lisp *throw*, or C *longjmp*—from the body of the user function to the context surrounding the group call.

```
– stub procedures for group calls to the participants
Ready* : proc (fn : proc (reply : status) return boolean) return void
Commit* : proc (fn : proc ( ) return boolean) return void
Abort* : proc (fn : proc ( ) return boolean) return void

– algorithm performed by the coordinator
TwoPhaseCommit : proc ( ) return void is
  success : boolean := true
  Phase1 : proc (reply : status) return boolean is
  begin
    if status = commit then
      return true
    else
      success := false
      return false
    end if
  end Phase1
  Phase2 : proc ( ) return boolean is
  begin
    return true
  end Phase2
begin
  Ready* (Phase1)
  if success then
    Commit* (Phase2)
  else
    Abort* (Phase2)
  end if
end TwoPhaseCommit
```

Figure 3: Two-phase commit using fn. mapping

If the result type *T* is *void*, the user function is invoked with no arguments—purely for synchronization—each time a request is acknowledged, as discussed in Section 4.1. The two-phase commit algorithm using functional mapping is shown in Figure 3.

5.2 Iterators

An iterator is an abstraction of the familiar **for** loop [11]. In programming languages that support abstract data types, iterators allow operations to be performed on each element of an abstract sequence, without revealing any information about how the sequence is implemented.

A **for** loop involving an iterator can be viewed as syntactic sugar for functional mapping: the **for** loop

for x **in** Iter(a) **do** Body(x)

is considered equivalent to

Iter'(a, λx . Body(x))

where *Iter'* is obtained from *Iter* by replacing all occurrences of the **yield** keyword by application of the additional functional parameter. Occurrences of **exit** or **return** must be replaced by a non-local transfer of control to the appropriate enclosing activation.

Iterators can be used as an abstraction of multicast communication. A stub generator using iterators would translate the example procedure *P* above into an iterator *P** with signature

P* : **iter** (arg : A) **yield** T

The iterator yields successive values in the order in which they arrive. The implementation of such a stub is shown in Figure 4.

```
P* : iter (arg : A) yield T is
   msg : Message := Marshal (P, arg)
   reply : T
begin
   MulticastRequest (msg)
   while not AllReceived? ( ) do
          msg := GetReply ( )
          reply := Unmarshal (msg)
          yield reply
   end while
end P*
```

Figure 4: Stub routine using iterators

A client of *P** would look like

for val : T **in** P* (x) **do** . . .

The client can use a loop **exit** statement to terminate the iteration early.

If the result type *T* is *void*, the iterator yields no data, only control, each time a request is acknowledged. In this case, it is useful to define the syntactic sugar all *P** (x) as a shorthand for

for in P* (x) **do skip**

```
– stub procedures for group calls to the participants
Ready* : iter ( ) yield status
Commit* : iter ( ) yield void
Abort* : iter ( ) yield void

– algorithm performed by the coordinator
TwoPhaseCommit : proc ( ) return void is
   success : boolean := true
begin
   for vote : status in Ready* ( ) do
       if vote = abort then
          success := false
          exit
       end if
   end for
   if success then
      all Commit* ( )
   else
      all Abort* ( )
   end if
end TwoPhaseCommit
```

Figure 5: Two-phase commit using iterators

The **all** operator is only useful in Circus-like systems in which the *AllReceived?* operation eventually returns *true*; the iteration would not terminate in a V-like system. The two-phase commit algorithm using iterators and the **all** operator is shown in Figure 5.

The use of iterators as a language-level mechanism for multicast communication was advocated in the author's thesis [4]. This approach possesses all of the essential properties, and all but one of the desirable properties, but it does not provide first-class status for multicast communication in progress.

5.3 Streams

The constructor **stream of** *T* defines a data type that can be used to transfer a sequence of values of type *T* between a producer and a consumer. Streams can be viewed as a generalization of iterators. Since streams are data, iteration in progress acquires first-class status.

The consumer of a stream of values uses the following operations:

```
EndOfStream? : proc (s : stream of T) return boolean
Get : proc (s : stream of T) return T
```

The producer of a stream of values uses the following operations:

```
Put : proc (x : T ; s : stream of T) return void
Close : proc (s : stream of T) return void
```

The *Get* and *Put* operations are equivalent to *Dequeue* and *Enqueue* operations on a FIFO queue. The *Close* operation indicates that no further values will be written to the stream; the

EndOfStream? operation returns *true* after the stream has been closed and all of the values written to it have been read. Some means of creating multiple threads of control (represented in Figure 6 by the **fork** operation) is needed to specify producers and consumers of streams. This is in contrast to the implicit coroutine discipline provided by iterators.

```
P* : proc (arg : A) return stream of T is
   msg : Message := Marshal (P, arg)
   s : stream of T
   Receiver : proc ( ) return void is
      reply : T
   begin
      while not AllReceived? ( ) do
            msg := GetReply ( )
            reply := Unmarshal (msg)
            Put (reply, s)
      end while
      Close (s)
   end Receiver
begin
   MulticastRequest (msg)
   fork Receiver ( )
   return s
end P*
```

Figure 6: Stub routine using streams

A stub generator using streams would translate the example procedure *P* above into a stub procedure *P** with signature

```
P* : proc (arg : A) return stream of T
```

The stream produces successive values in the order in which they arrive. The implementation of the stub procedure is shown in Figure 6. A more efficient implementation is possible if (as is usually the case) the transport layer maintains a queue of replies for each request: the queue can serve as the stream itself, *Get* is implemented as *Dequeue* followed by *Unmarshal*, and *EndOfStream?* is just *AllReceived?*.

A client of *P** would look like

```
s : stream of T := P* (x)
while not EndOfStream? (s) do
      val : T := Get (s)
      . . .
end while
```

If the result type *T* is *void*, the resulting stream is used purely for synchronization, as discussed in Section 4.1. The *Get* operation returns only after a corresponding *Put* operation is performed, even though no data is transferred. In this case, it is useful to define the syntactic sugar **all** *s* as shorthand for

```
while not EndOfStream? (s) do Get (s)
```

```
Average : proc (s : stream of integer) return integer is
  sum : integer := 0
  count : integer := 0
begin
  while not EndOfStream? (s) do
       value : integer := Get (s)
       sum := sum + value
       count := count + 1
  end while
  return sum / count
end Average
```

Figure 7: Example of an averaging collator

The **all** operator cannot be used unless the *AllReceived?* operation provided by the underlying transport layer eventually returns *true*.

Any program that uses iterators can be trivially rewritten to use streams, but not vice versa. The first-class status of a stream allows collators [3], for example, to be written as regular functions with stream parameters. Figure 7 shows a function that averages its stream of inputs. It can be composed with a stub procedure for a group call to an integer-valued procedure, say *ReadTemperature*, simply by writing

Average (ReadTemperature* ( ))

If iterators were used, the collating code would have to be duplicated in each caller.

The two-phase commit algorithm using streams is shown in Figure 8.

6 Discussion

In addition to satisfying increasingly more of the desirable properties for language-level multicast communication, the mechanisms presented in Section 5 also demand progressively more from the underlying programming language.

The simplest mechanism, functional mapping, requires only the ability to pass a filtering function as a parameter. Procedure parameters are available in most languages, including (for example) C and Modula-2.

Iterators, as abstract **for** loops, are most appropriate in languages with good support for abstraction. Using iterators for multicast communication would be appropriate in languages like CLU and Standard ML.

Finally, streams presuppose a facility for creating multiple threads of control, in order to specify independent producer and consumer activities. The stream approach can be used in languages like Ada or C++ extended with C Threads [5].

6.1 Associating Responses with Servers

The multicast constructs presented in Section 5 offer no way for a client to determine which server produced a given result; this makes it inconvenient to express some algorithms. One possibility is to place the burden of identifying responses on the implementor of the server, and require each procedure to return both its result and its server ID. But this violates

```
– stub procedures for group calls to the participants
Ready* : proc ( ) return stream of status
Commit* : proc ( ) return stream of void
Abort* : proc ( ) return stream of void

– algorithm performed by the coordinator
TwoPhaseCommit : proc ( ) return void is
  success : boolean := true
  s : stream of status
begin
  s := Ready* ( )
  while not EndOfStream? (s) do
        vote : status := Get (s)
        if vote = abort then
          success := false
          exit
        end if
  end while
  if success then
    all Commit* ( )
  else
    all Abort* ( )
  end if
end TwoPhaseCommit
```

Figure 8: Two-phase commit using streams

modularity, by requiring the implementor of a procedure to know how it will be called by clients, and it is redundant, since the underlying multicast transport protocol already has the server ID information available.

A better solution involves a simple extension to the stub generators for the multicast constructs already presented. The stub generator that transforms the type signature of a procedure P into a multicast stub $P*$ can, as an option, replace all occurrences of the result type T by pairs of the form $\langle T, ServerID \rangle$. When the stub procedure unmarshals the result value (of type T), the server ID is also extracted from the response message, and both are returned to the client. Which form of stub to use is left to the implementor of the client program; the choice has no effect on the server. Equivalent facilities are found in MultiRPC [14] and the *cstub* stub generator [12].

6.2 Exception Handling

Distributed systems, unlike centralized ones, must deal with *partial failures* such as processor crashes and communication outages. Experience has shown that an exception handling mechanism is a useful way of allowing programs to deal with these failures at the language level. We assume a termination model of exception handling, as in CLU, Ada, and Standard ML, and turn our attention to the interaction between exception handling and the multicast constructs proposed in Section 5.

In the case of functional mapping, only procedure calls are involved, and the semantics of exception handling are well defined. Exception handlers specified in the filter procedure are only active while the filter procedure is being applied to a result (during evaluation of the expression *fn*(*reply*) in Figure 2). In particular, if an exception is raised by the transport or presentation layers during execution of the stub procedure, the filter procedure and its exception handlers are not active and do not affect the processing of the exception.

A simple way to derive the correct exception semantics for iterators is to de-sugar the **for** loop

for x **in** P* (a) **do** Body(x)

into the equivalent functional form

P* (a, λx . Body(x))

as discussed in Section 5.2. If an exception occurs while executing *Body* and is not handled there, control will pass first to the stub procedure *P**, and then to the context surrounding the **for** loop. An exception raised by the underlying protocol layers during execution of *P**, on the other hand, is not affected by any handlers in *Body*, because the closure λx . *Body(x)* is not active.

It is not obvious what the correct exception semantics should be for streams. The simplest approach involves closing the stream at the producer side when an exception occurs, and raising an exception on the consumer side when an attempt is made to read from the closed stream. The producer (the *Receiver* thread in Figure 6) can guarantee this, for example, by calling the *Close* operation within a Lisp-style **unwind-protect**. This is not entirely satisfactory, since information about the precise exception is lost: the producer might detect a transport-specific exception, say *ServerCrashed*, but the consumer receives only a generic *StreamClosed* exception. Some means of propagating the exception through the stream would alleviate this problem; this is an area for further research.

6.3 Future Work

There are several directions for further research in this area:

- The language-level multicast mechanisms described in this paper should be implemented in real languages, used in real applications, and analyzed in detail.
- First-class continuations appear promising as a unifying mechanism. They can be used to express all of the constructs discussed here, and may suggest additional control structures for multicast communication.

References

[1] David R. Cheriton.
VMTP: A transport protocol for the next generation of communication systems.
In *Proceedings of SIGCOMM '86 Symposium on Communications Architectures and Protocols*, pages 406–415, August 1986.

[2] David R. Cheriton and Willy Zwaenepoel.
Distributed process groups in the V kernel.
ACM Transactions on Computer Systems, 3(2):77–107, May 1985.

[3] Eric C. Cooper.
Replicated procedure call.
In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 220–232, August 1984.
Reprinted in *Operating Systems Review*, 20(1):44–56, January 1986.

[4] Eric C. Cooper.
Replicated Distributed Programs.
PhD thesis, Computer Science Division, University of California, Berkeley, April 1985.
Published as report UCB/CSD/85/231.

[5] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June 1988.

[6] Stephen E. Deering.
Host extensions for IP multicasting.
RFC 1054, Stanford University, May 1988.

[7] David K. Gifford.
Weighted voting for replicated data.
In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 150–162, December 1979.
Published as *Operating Systems Review*, 13(5).

[8] J. N. Gray.
Notes on data base operating systems.
In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1978.
Volume 60 of *Lecture Notes in Computer Science*.

[9] Thomas J. LeBlanc and Robert P. Cook.
Broadcast communication in StarMod.
In *Proceedings of the 4th International Conference on Distributed Computing Systems*, pages 319–325, May 1984.

[10] Barbara Liskov and Liuba Shrira.
Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems.
In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 260–267, June 1988.
Published as *SIGPLAN Notices*, 23(7).

[11] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert.
Abstraction mechanisms in CLU.
Communications of the ACM, 20(8):564–576, August 1977.

[12] Bruce Martin.
Parallel Remote Procedure Call Language Reference and User's Guide.
Computer Systems Research Group, University of California, San Diego, 1986.

[13] Bruce Jay Nelson.
Remote Procedure Call.
PhD thesis, Computer Science Department, Carnegie Mellon University, May 1981.
Published as CMU report CMU-CS-81-119 and Xerox PARC report CSL-81-9.

[14] M. Satyanarayanan and Ellen H. Siegel.
Parallel communication in a large distributed environment.
IEEE Transactions on Computers, March 1990.

[15] Andrew S. Tanenbaum.
Computer Networks.
Prentice-Hall, 2nd edition, 1988.