Generative Communication in Linda

DAVID GELERNTER
Yale University

Generative communication is the basis of a new distributed programming langauge that is intended for systems programming in distributed settings generally and on integrated network computers in particular. It differs from previous interprocess communication models in specifying that messages be added in tuple-structured form to the computation environment, where they exist as named, independent entities until some process chooses to receive them. Generative communication results in a number of distinguishing properties in the new language, Linda, that is built around it. Linda is fully distributed in space and distributed in time; it allows distributed sharing, continuation passing, and structured naming. We discuss these properties and their implications, then give a series of examples. Linda presents novel implementation problems that we discuss in Part II. We are particularly concerned with implementation of the dynamic global name space that the generative communication model requires.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network communications*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; D.4.4 [**Operating Systems**]: Communication Management—*message sending.*

General Terms: Languages

Additional Key Words and Phrases: Distributed programming languages

1. INTRODUCTION

In his introduction to the distributed programming language SR, Andrews writes that:

> there are only three basic kinds of mechanisms and three corresponding models of concurrent programming: monitors (shared variables), message passing, and remote operations [1, p. 405].

In the following we introduce *generative communication*, which, we argue, is sufficiently different from all three to constitute a fourth model. Our new technique is closest to message passing, but the differences between the two are as significant as the similarities.

How should communication be incorporated in a programming language? is our central question. The question has grown in interest as hardware has become cheaper and networks connecting large numbers of inexpensive processors cor-

This material is based upon work supported by the NSF under grant MCS-8303905.
Author's address: Dept. of Computer Science, Yale University, New Haven, CT 06520.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1985 ACM 0164-0925/85/0100-0080. $00.75

respondingly more important. Such networks fall into two main classes: local area nets connecting autonomous personal workstations; in "network computers," still purely experimental, large numbers of processors work simultaneously on a single problem. Both sorts of network call for the construction of programs that run on many processors at once. On local area nets, these multimachine programs are mainly network operating-system and utility routines such as file and mail servers. On network computers, they include the operating system and any applications that are amenable to parallel solutions. Applications that have been studied in this context include numerical algorithms, simulation and sorting [5]; parallel artificial intelligence programs are of rapidly growing interest. (See, for example, Fehling and Erman [8].)

Network operating systems and parallel applications may be written in conventional languages augmented with systems calls for interprocess communication. There is broad agreement, however, that distributed programming in a distributed language—in a language, that is, to which communication is intrinsic—is in principle more satisfactory. What should a distributed language look like? In particular, how should it deal with interprocess communication? The concurrent systems languages that were based on the work of Dijkstra, Hoare, and Brinch-Hansen (for example Brinch-Hansen's Concurrent Pascal [2] and Wirth's Modula [27]) have generally been taken as starting points. Programs in both concurrent and distributed languages may consist of many concurrent processes, but the two language categories differ in the types of interprocess communication they may incorporate. The multiple processes of a concurrent-language program are multiplexed on one machine, not distributed over many; accordingly, they need not execute in wholly-disjoint address spaces, and they may use shared variables or a generalization of conventional parameter passing such as the monitor call to communicate. The usual assumption in distributed languages, on the other hand, is that concurrent processes sometimes execute on separate machines and always in disjoint address spaces. They may therefore communicate with each other only by sending and receiving messages that are carried between address spaces by a runtime communication system.

Generative communication is the basis of a new distributed programming language called Linda, developed originally for the SBN network computer [11]. But many distributed or distributable programming languages have already been proposed: CSP [17], DP [3], Plits [9], ECLU [23], Starmod [4], Ada [6], and SR [1] are examples we discuss below. What need for another one? We argue first that, irrespective of the strengths and weaknesses of the many other proposals, Linda's unusual features make the language suggestive and interesting in its own right. Where most distributed languages are partially distributed in space and nondistributed in time, Linda is fully distributed in space and distributed in time as well, as we discuss. Second, Linda appears in many cases to be both simpler and more expressive than existing proposals within the application domains of interest. Linda attempts the same combination of simplicity and power in the distributed domain that the systems language C [20] achieves in the sequential one.

The Linda design encompasses a model of computation with implications beyond distribution and communication as such. In this paper we are concerned, however, with communication exclusively. We introduce other aspects of the

Linda design only when communication issues make this necessary, and then only in brief outline. Part I deals with generative communication in Linda. Part II briefly discusses the implementation of generative communication on networks, particularly on network computers.

A final point: We attempt in the following to place Linda in context by comparing it to other distributed languages, as noted. But there are a very large number of designs that are relevant to this work in one way or another, and not all can be discussed here. We have chosen to concentrate on those other proposals that are closest in conception to ours, because the differences in these cases are least obvious. Other reports discuss Linda in relationship to Shapiro's Concurrent Prolog [14] and to several AI languages [13]. We also omit any discussion of formal semantics here; some semantics issues are discussed elsewhere [13].

2. PART I: GENERATIVE COMMUNICATION

A Linda program is a collection of ordered tuples. Some tuples incorporate executing or executable code; others are collections of passive data values. We are interested only in passive tuples here.

The abstract computation environment called "tuple space" is the basis of Linda's model of communication. An executing Linda program is regarded as occupying an environment called "tuple space" or TS. However many concurrent processes make up a distributed program, all are encompassed within one TS. Consider two communicating processes A and B. To send data to B, A generates tuples and adds them to TS; B withdraws them. (Figures 1 and 2.)

This communication model is said to be generative because, until it is explicitly withdrawn, the tuple generated by A has an independent existence in TS. A tuple in TS is equally accessible to all processes within TS, but is bound to none. The remainder of this paper concerns the three operations defined over TS; out( ) adds a tuple to TS, in( ) withdraws one, and read( ) reads one without withdrawing it.

The following sections first define out( ), in( ), and read( ) in their simplest forms, then generalize them by introducing *structured names.* Next, we discuss the basic properties inherent in our definitions that distinguish Linda and set it apart from other distributed programming languages. Finally, we present a series of examples.

2.1 Simple out( ), in( ), and read( )

2.1.1 *Preliminary: The Type "name".* As a type-specifier, "name" refers to a character string acting as an identifier. Name-valued variables and constants are attached as identification tags to tuples, as we discuss.

2.1.2 *Definitions: out( ), in( ), and read( ).* The out( ) statement appears as

```
out(N, P2, . . . , Pj),
```

where $P_2, \ldots, P_j$ is a list of parameters each of which may be either an actual or a formal, and N is an actual parameter of type name. Assume for the time being that all of the P_i are actual parameters. Execution of the out( ) statement results in insertion of the tuple $N, P_2, \ldots, P_j$ into TS; the executing process continues immediately.

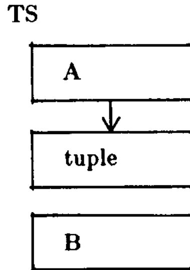


Fig. 1. To send to B, A generates a tuple...

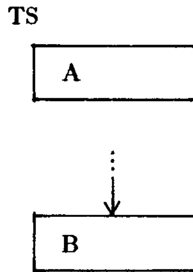


Fig. 2. ... and B withdraws it.

The in( ) statement appears as

```
in(N, P₂, . . . , Pⱼ),
```

where $P_2, \ldots, P_j$ is a list of parameters each of which may be either an actual or a formal, and N is an actual parameter of type name. Assume for the time being that all of the P_i are formal parameters. When the in( ) statement is executed, if some type-consonant tuple whose first component is "N" exists in TS, the tuple is withdrawn from TS, the values of its actuals are assinged to the in( )-statement's formals, and the expression executing the in( )-statement continues. If no matching tuple is available in TS, in( ) suspends until one is available and then proceeds as above.

The read( ) statement,

```
read(N, P₂, . . . , Pⱼ).
```

is identical to the in( ) statement except that, when a matching tuple is found, assignment of actuals to formals is made as before but the tuple remains in TS.

Nondeterminism is inherent in these definitions. Many in(N, ...) statements may execute simultaneously and suspend in several portions of a distributed program. Execution of out(N, ...) under such circumstances makes a single copy of the output tuple available. *Some* suspended in( ) statement will receive it, but *which* is logically nondetermined. The same holds for an in(N, ...) or read(N, ...) statement that executes subsequent to many out( )s; some stored tuple will be matched, but which one is nondetermined.

The definitions require that tuples be inserted into and withdrawn from TS atomically. If two in( ) statements contend for one tuple, one gets it and the other

does not; they cannot split it. (Note that if a read( ) statement and an in( ) statement contend for one tuple, the two statements are served, as always, in arbitrary order. If the read( ) statement is served first, in( ) may subsequently withdraw the same tuple. If in( ) is served first, it removes the tuple and read( ) will block until another is available.)

Tuple names are global to a given program's TS. A tuple added to TS may be removed by an in( ) statement occurring anywhere else in the program.

2.2 Structured Naming

Let $+t$ refer to any tuple added to TS by an out($+t$) statement. Then $-t$ designates the parameter list of any in( ) or read( ) statement that may legally receive tuple $+t$.

All of the list $F_2, \ldots, F_k$ cited by an in( ) or read( ) statement need not be formals. Any or all components on the list may in fact be actuals. The actual parameters appearing in a $-t$ tuple (including the initial name-valued actual) collectively constitute a structured name. Thus the statement

```
in(P, i:integer, j:boolean)
```

requests a tuple with name "*P*". But it is also possible to write

```
in(P, 2, j:boolean);
```

in this case a tuple with structured name "*P*, 2" is requested. The statements

```
in(P, i:integer, FALSE)
```

and

```
in(P, 2, FALSE)
```

are also possible. In the first case, the structured name is "*P*,,FALSE" and in the latter "*P*,2,FALSE". All actual components of a $-t$ list must be matched identically by the corresponding components of a $+t$ tuple in order for matching to occur.

Structured naming is similar in principle to the "select" operation in relational database systems, and may be said to make TS content-addressable. It also resembles in a rudimentary but significant way the pattern-matching features that are part of some AI and logic languages, particularly Prolog [26]. The in( ) statement and the Prolog assertion both deal with tuples whose unspecified components (the formals) may be inferred from some arbitrary specified subset (the values).

Just as components of $-t$ may be actuals, any component of a $+t$ tuple, except for the initial name-valued actual, may be a formal. Thus

```
out(P, 2, FALSE)
```

represents the simplest form of out( ) statement, but

```
out(P, i:integer, FALSE)
```

is also possible. The tuple *P*,*i*:integer,FALSE may be received by any in( ) statement that specifies first component "*P*", last component "FALSE", and middle component some actual of type integer. A tuple may in general be received

by any in( ) statement with an identical actual where the tuple has an actual, and a type-consonant actual where the tuple has a formal. Formals never match formals. The use of formals in out( ) statements is referred to as inverse structured naming.

Note. A formal declared in an in( ) statement's parameter list is defined throughout the immediately enclosing lexical block, exactly as if it were declared as a local to that block. Variables declared conventionally (rather than in parameter lists) may be used as formals by in( ) statements wherever they are lexically visible. In this context they must be prefixed by "var"—otherwise they will be interpreted as actuals and part of a structured name. Thus, in the sequence

```
i:integer . . . in(P, var i),
```

the integer variable *i* serves as an in-statement formal. In the sequence

```
i:integer . . . in(P, i),
```

i's value is one component of the structured name *P*, *i*. Finally, note that all formals declared within in( ) statements must include explicit type designators. The statement in(*P*, *i*, *j*:integer) requests a tuple with structured name "*P*, *i*"—it does *not* declare two formals, *i* and *j*, of type integer.

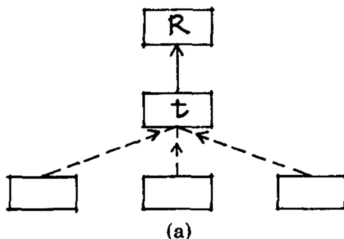
A formal declared in a +*t* tuple is defined only within that +*t* tuple. In the present context, formals in +*t* tuples serve only as place holders, and their names are unimportant.

2.3 Distinguishing Properties

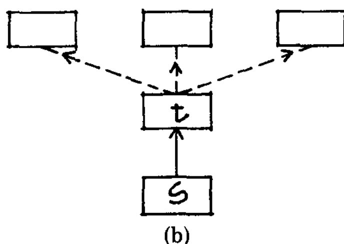
Generative communication as defined by in( ), read( ), and out( ) has two fundamental characteristics that give rise to a series of distinguishing properties.

The first fundamental characteristic is *communication orthogonality.* It is ordinarily the case that execution of an arbitrary distributed programming language's receive statement *r* presupposes no knowledge about which process is to execute the matching send-statement *s*; it is in fact usually the case, on analogy with procedures or subroutines in sequential languages, that *r* may receive input from any number of sending processes (as a procedure may be invoked from any number of points within a sequential program). Most send statements, however, name the receiver explicitly in some fashion—on analogy with a procedure-invocation statement, which invariably names the invoked procedure. In Linda, on the other hand, communication is orthogonal in the sense that, just as the receiver has no prior knowledge about the sender, the sender has none about the receiver. Communication orthogonality has two important consequences: space-uncoupling (also referred to as "distributed naming") and time-uncoupling. A third property, distributed sharing, is a consequence of the first two.

p1. *Space uncoupling* (distributed naming). Distributed naming refers to the fact that a tuple in TS tagged "*P*" may be input by any number of address-space-disjoint processes. In particular, *j* processes executing on *j* distinct network nodes may all accept tuples tagged with one name. Distributed naming means that Linda is fully distributed in space. Distributed programming languages ordinarily (as noted) allow a receive statement to accept data originating in any one of



(a)



(b)

Fig. 3. Distributed naming. (a) Tuple t input by R may have originated in any one of many address spaces; (b) tuple t output by S may be received in any one of many address spaces.

many distinct address spaces (Figure 3a). Linda allows a sender to direct data to any one of many distinct address spaces as well (Figure 3b).

p2. *Time uncoupling.* A tuple added to TS by out( ) remains in TS until it is removed by in( ). If it is never removed by in( ) it will, in the abstract, remain in TS forever. In practice, tuples added to TS by a given distributed program will be removed once all that program's processes have terminated, unless the programmer indicates explicitly to the contrary. Despite these practical considerations, Linda allows programs to be distributed in time insofar as process A in which some out( ) statement appears may run to completion before process B, in which the corresponding in( ) appears, is loaded. Thus, while distributed programming languages ordinarily allow communication between processes that are space-disjoint (Figure 4a), Linda allows communication between time-disjoint processes as well (Figure 4b).

A third effect results from space- and time-distribution:

p3. *Distributed sharing.* Linda allows j address-space-disjoint processes to share some variable v by depositing it in TS. The TS-operator definitions ensure that v will be maintained atomically. It is not necessary (as in other languages) that a shared variable be implemented by a process or module.

We refer to the second of generative communication's two fundamental properties as *free naming.* For expository purposes we introduced type "name" first

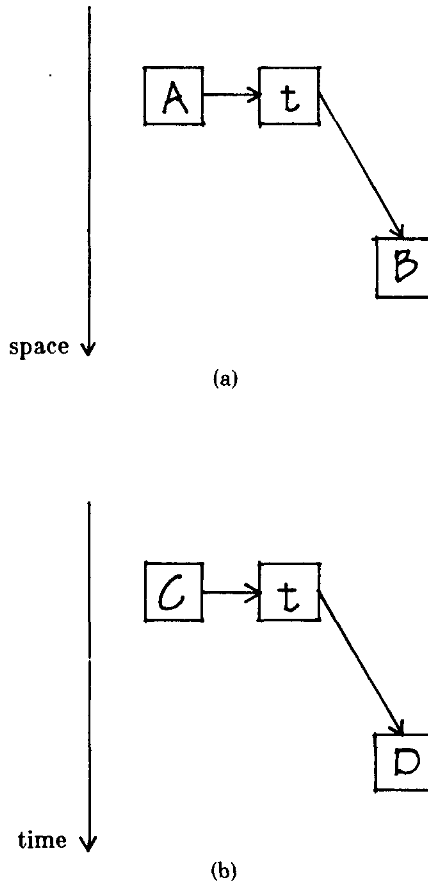


Fig. 4. Distribution in time. (a) Space-disjoint processes A and B communicate via TS; (b) time-disjoint processes C and D—C terminates before D starts—likewise communicate via TS.

and the idea of structured naming subsequently. In fact, the existence of type name is a logical consequence of structured naming. Other distributed programming languages distinguish between the name specified by an in( ) statement and its list of parameters. In Linda, this distinction is eliminated: the name specified by an in( ) statement is by definition *the sequence of actual parameters that appear in the in( ) statement's parameter list.* Actuals, however, are simply *values* that appear in a parameter list. It follows that, if an in( ) statement specifies only a single actual—by definition that single actual will appear at the beginning of the list—then that single actual must itself be a value. Values have types, and we say that the type of a value serving as an in( ) statement's single actual is "name". These facts about Linda are collectively designated "free naming", and they result in two further properties.

p4. *Support for continuation passing.* If there are values of type name, there must be variables of type name as well. Hence, names may be passed as tuple

components, be assigned to formals, and be stored in arbitrary data structures just as values of any other type may. We use "continuation passing" to refer specifically to one process's sending data to another, blocking in expectation of a reply, and being continued ultimately by a third (or some *n*th) process. Linda supports continuation passing in a simple and flexible way by allowing name-type values to be tuple components and to be stored in arbitrary data structures in the same way as values of any other type.

p5. *Structured naming.* Structured names allow in( ) and read( ) statements to restrict, and out( ) statements to widen, their focus. They also allow one name to generate a family of others.

The examples illustrate each of these properties together with the basic program structures that Linda induces. It is incumbent upon the designers of a language that, like Linda, maximizes power and flexibility in a broad solution space over simplicity and convenience in a more narrowly constrained one to provide for macro- and library-routine specification where simplicity and convenience are wanted. A macro definition facility is introduced and discussed in the course of the examples. Finally, the examples refer to the several other distributed or distributable programming language proposals mentioned above for purposes of comparison.

The following sections present preliminary definitions, then a group of examples using simple in( ) and out( ) without structured names, a group using structured names as well, and a final group using in( ) and out( ), structured names, and read( ).

2.3.1 *Preparatory to the Examples: An Outline of Expression Structure in Linda.* In order to give examples, we need first to consider briefly the structure of Linda expressions.

Linda expressions consist of simple statements and composed statements. The term "expression" is appropriate for a collection of statements because statements have values, but this does not concern us here. Simple statements include assignments, procedure calls, and control statements. These are again unimportant in the present context; they are mainly identical to simple statements in C. Composed statements on the other hand are germane because they determine program structure. Linda has four composed statements: sequence-statements, and-statements, or-statements, and star-statements.

Let each of $s_1, \ldots, s_j$ be either a simple statement or a composed statement enclosed in square brackets. A sequence-statement takes the form

```
s1; s2; . . . ; sj
```

and specifies that $s_1, \ldots, s_j$ are executed sequentially (or, in Linda's terminology, have sequential lifetimes). The and-statement

```
s1 & s2 & . . . & sj
```

specifies that $s_1, \ldots, s_j$ have concurrent lifetimes. For each pair of and-statement constituents, if both are executable statements, they execute concurrently; if one is a statement and the other is a variable declaration, the declared variable exists for as long as the statement executes and is accessible to all constituents of the and-statement within which it is declared. Thus the and-statement provides both

for variable declaration and for concurrency. The and-statement terminates when all of its constituents have either terminated or "quiesced" (as defined below). Variables are always quiescent.

Variables may be declared only within and-statements. The special notation

```
s1; . . . in(P, i:integer); . . . ; sn
```

is simply an abreivation for

```
i:integer & [s1; . . . in(P, var i); . . . sn].
```

(These rules are unconventional, but since their justification and implications lie beyond the scope of this paper, the reader is asked to suspend judgment. The and-statement is the basis of Linda's so-called symmetric structure, and it allows for a richer collection of assignment-statement forms and of binding-environment specifications than Algol languages ordinarily provide. These issues are discussed in [13], and at length in [15].)

The or-statement

```
s1 | s2 | . . . | sj
```

specifies that $s_1, \ldots, s_j$ have mutually exclusive lifetimes. A single enabled constituent of the or-statement is chosen for execution; the choice is made nondeterministically at runtime. If no constituent is enabled, execution suspends until one is. For present purposes, an or-statement constituent is not enabled only if it consists of or begins with an in( ) statement and no matching tuple for that in( ) statement is available in TS.

The star-statement

```
*s
```

executes "*s*" 0 or more times. If *s* begins with an in( ) statement (this is the intended case), *s* executes once for each matching tuple available in TS; when no matching tuples are available, the star statement quiesces. Ordinarily a quiescent star statement terminates immediately, but it is prevented from terminating if it is one and-statement component and other components of the same and-statement are still active. A quiescent star-statement is reawakened by the appearance of a new matching tuple in TS; it terminates when all other components of the encompassing and-statement have quiesced or terminated.

Star-statements of the form

```
*[in(-t); s],
```

where "*s*" is any statement, are referred to as "in blocks", and are so common that it is convenient to introduce the following shorthand notation for them:

```
in(-t) ⇒ s.
```

2.4 Examples Using Simple in() and out()

2.4.1 *Basic Communication Structures.* out( ) and in( ) may be used to implement message-passing in the obvious way. Remote procedures are implemented by combining out( ) and in( ):

remote procedure call: `out(P, me,` *out-actuals*`);`
`in(me,` *in-formals*`)`

```
remote procedure:  in(P, who:name, in-formals) ⇒
                   [body of procedure P;
                   out(who, out-actuals)
                   ]
```

The distinguished name "me" is replaced by the compiler with any name that uniquely designates the process within which it appears, where "process" is a sequentially-executing code block and a distributed program may encompass many processes.

The structure referred to here as a remote procedure may be equivalently referred to as an "active procedure"—a procedure-like block that is executed by a thread of control associated with itself, not with the caller. An active procedure is neither reentrant nor recursively callable. In Linda it is ordinarily one component of an and-statement whose other components are the processes that make use of it. It disappears, as per the definitions of and-statements and star-statements, when the encompassing and-statement terminates.

Note that Linda resembles Plits, ECLU, SR, and the language fragment described by Kahn [19] in providing an asynchronous output operation.

2.4.2 *Semaphores and Communication Orthogonality.* A single-element tuple is functionally equivalent to a semaphore. in(sem) is functionally equivalent to the semaphore operation *P*(sem), out(sem) to *V*(sem). A semaphore "sem" is initialized to value *n* by *n* repetitions of out(sem).

The semaphore-like nature of in( ) and out( ) is one result of communication orthogonality in Linda. The space- and time-uncoupling properties mean that singleton "sem" tuples exist independently of the processes that generate them, and outlive those processes if necessary. Distributed naming means that the semaphores defined by in( ) and out( ) are in fact distributed semaphores (as discussed, for example, by Schneider [24])—that is, semaphores defined over arbitrarily-many disjoint address spaces. It is an expression of distributed sharing that processes may share semaphores directly via TS; no executable code block need be programmed to implement *P*( ) and *V*( ) requests. That semaphores are primitive is also indicative of Linda's simplicity and flexibility. Languages that abstract at a level higher than semaphores rely, at least conceptually, on high-level constructs (such as tasks or processes) to implement these low-level objects.

Linda differs from all other surveyed systems except CSP in making communication orthogonal. Linda and CSP are, however, orthogonal in opposite senses. In CSP, communicating processes must name each other explicitly; data is transferred synchronously from sender to receiver with no system-provided buffering. Thus communication in CSP is space-coupled (because of the naming rules) and time-coupled (because of synchronous transfer).

2.4.3 *Distributed Naming.* Consider a maximally-simple idle-node allocation procedure for a network computer. Idle nodes might execute

```
in(idle, j:job_description);
  execute job "j".
```

To fork a new asynchronous task "*k*", processes execute

```
out(idle, k).
```

Tuples tagged "idle" may be input by any network node because of distributed naming. (Note that if the out( ) statement executes when there is no idle node in the network, hence none executing its in(idle, ...) statement, the "idle, *k*" tuple remains buffered in TS until some node becomes idle and picks it up. This is a result of time-uncoupling.)

Another example: In Shoch and Hupp's Ethernet worm programs [25] users sometimes need to communicate with a worm segment. Any segment will do, but the worm's whereabouts are unknown until runtime. In Linda, worm segments would execute

```
in(worm_segment, r: request);
  execute request r,
```

and users with request *k* would execute

```
out(worm_segment, k).
```

None of the other languages under discussion supports distributed naming. ECLU allows one name to be referenced by many processes, but only if all are confined within one Guardian and hence within one physical node.

2.4.4. *Synchronized Message Exchange: Calling-Routine Structures and Macro Definitions.* CSP provides (as noted) synchronizing message-exchange operators. The receive operation suspends until a matching send is executed and the send suspends until a matching receive. To implement these operations it is not sufficient for the sender to transmit as soon as it reaches the send statement, then suspend execution until the receiver forwards an acknowledgment. No message buffering is to be provided; the sender therefore cannot transmit until the receiver has suspended at its receive statement.

Presenting an implementation of these operations in Linda is a good occasion for the introduction of a macro facility. Programs that use synchronous communication specify or include the macro defintions:

```
def SYNCH_SEND(s:tuple)      [in(get); out(s); in(got)]
def SYNCH_RECV(s:tuple)      [out(get); in(s); out(got)].
```

To communitate synchronously, process A executes `SYNCH_SEND(B, message)` and B executes `SYNCH_RECV(B, m:message_type)`. Note that this implementation requires that the singleton tuple "get" be buffered in TS, but does not require that the message text be buffered.

These macro definitions are satisfactory only if they are used by a single pair of processes within a given TS—otherwise the names "get" and "got" will be ambiguous. More general definitions use structured names, as discussed below.

In conventional languages, called routines (procedures, functions, subroutines, and so on) may be arbitrarily elaborate, but calling routines (the protocol followed by one process in passing data to, and if necessary retrieving results from, another) ordinarily consist of a single invocation statement. Linda allows the programming of calling routines that may in general be as elaborate as the called routine itself. SYNCH_SEND( ) is one example of a calling routine; others follow.

2.4.5 *Resource Servers: The Continuation-Passing Property; the Filter and the Active-Monitor Structures.* Continuation passing and communication orthogo-

nality allow resource servers to be programmed using the familiar feeder-driver paradigm instead of the input guards that are common in distributed programming languages. In the latter case, request queues are maintained in the kernel and are not directly accessible to the server. In the former, they are constructed and maintained entirely by the server. We include two examples whose solutions are included in the Preliminary Ada Rationale [18]. The Ada solutions represent an input-guard programming style that is usefully compared to Linda's feeder-driver style.

Disk head scheduler. The Ada example assumes that client processes will communicate with the disk head scheduler by passing two parameters, one of type "track" and the other of type "data." We make the same assumption.

The logical structure of the Linda disk server is shown in Figure 5.

The user passes its name to the filter, blocks under this name, and continues once its request has been filled by the server. Requests accepted by the filter are passed on to the server directly if the server is free; otherwise they accumulate in the filter, and the filter passes them onward in whatever order is appropriate to the problem. The existence of the filter is transparent to the user. The filter structure may be programmed in Linda as a result of the continuation-passing property.

Users call the disk by executing the calling routine DISK_SERVICE(track, data), where

```
def DISK_SERVICE(t: track; d:data)
  [out(disk_request, me, t, d);
    in(me)].
```

The server itself takes the form

```
DISK_FILTER&DISK
```

where DISK is

```
def DISK
  *[{ask the filter for an i/o request:}
      out(get_diskrequest);
    {receive the request:}
      in(disk_driver, who:name, i:track, d:data);
      perform the physical i/o transfer;
      {tell the user to continue:}
      out(who)                  ]
```

DISK is an active procedure: It repeatedly asks FILTER for an i/o request, performs it, and notifies the user that service is complete.

DISK_FILTER is an active monitor, a structure that in general takes the form

shared variables `&*[r1|r2| . . . |rn]`,

where the r_i are routines that access the shared variables in mutual exclusion. An active monitor resembles a Hoare monitor (only the active monitor's routines are executed by the thread of control associated with the monitor itself, not with the user), or an Ada accept block.

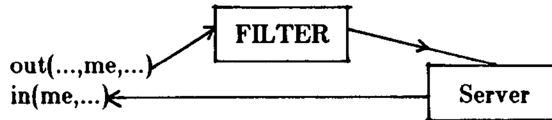


Fig. 5. A filter/server paradigm.

DISK_FILTER takes the form

```
DECLARATIONS&*[DISK_ENQUEUE|DISK_DEQUEUE]
```

where

```
def DECLARATIONS
    disk_idle:boolean = true
  & declarations required for building and maintaining
    the i/o requests queue; queue frames will have
    three fields,
    of type name, track, and data
def DISK_ENQUEUE
    [in(disk_request, who:name, i:track, d:data);
     {accept disk i/o request}
     if (disk_idle) {send the request on directly:}
       [out(disk_driver, who, i, d);
        disk_idle = false]
     else insert who, i, d in the i/o request queue
       using any desired ordering discipline
    ]
def DISK_DEQUEUE
    [{accept disk's request for the next i/o request}
     in(get_diskrequest);
     who:name& i:track& d:data& {local variables}
    [if request queue is empty
     {disk will be answered eventually by DISK_ENQUEUE:}
     disk_idle = true
     else [who, i, d = head entry on request queue;
           out(disk_driver, who, i, d)]
    ]]
```

The disk head scheduler is considerably more complex in Ada. The user calls a procedure which calls an entry on its behalf; the entry registers the user's request and then the user suspends on another entry call, this time to one of a family of entries. Meanwhile, a background task repeatedly calls another entry to determine which request to serve next, and ultimately continues the user by accepting the entry call on which it is suspended. The elevator algorithm used to service i/o requests is intrinsic to the tasking and communication structure of the Ada example. (It does not, because it need not, figure in the Linda example.)

The *multiresource controller* problem as stated in the Ada Rationale has users requesting a subset of resources from among a set designated "A" through "K". Users request a resource group by passing to the server a parameter "group" of type "resource_set", and return a group by passing a parameter of the same type. This example is simpler than the disk head scheduler. It requires an active monitor only, with no additional driver process.

The Linda solution takes the form

```
DECLARATIONS&*[MRC_ENQUEUE|MRC_DEQUEUE].
```

The declarations include, as in the Ada example, a function "try" that returns TRUE if the resource_set passed is free and can be allocated to a requestor, and a variable "used" of type resource_set that records which resources are unavailable. Resource sets are bit vectors, and we assume Ada's bit-wise logical operations. The request queue is built of frames containing a field "who" of type name and a field "asked" of type resource set. For MRC_ENQUEUE and MRC_DEQUEUE, we use

```
def MRC_ENQUEUE
    in(MRC_Request, who:name, asked:resource_set);
      [if (try(asked))
        {assign this group to the requestor:}
        [out(who); used = used or asked]
        else append who, asked to the request queue
      ]
def MRC_DEQUEUE
    in(MRC_Release, released:resource_set);
      [used = used and not released;
       for each frame q in the request queue:
       if (try(q.asked))
       {award the requested group to the blocked requestor:}
        [out(q.who); used = used or q.asked;
         remove q from the request queue]
      ].
```

To request resource set *s*, users execute

```
REQUEST_RESOURCES(s)
```

and to release *s*,

```
RELEASE_RESOURCES(s),
```

where

```
def REQUEST_RESOURCES(s:resource_set)
  [out(MRC_REQUEST, me, s);
   in(me)]
def RELEASE_RESOURCES(s:resource_set)
  out(MRC_RELEASE, s).
```

Note that the Linda example depends once again upon continuation passing.

The Rationale's Ada solution is, again, considerably more complex logically. The request queue consists of all nonaccepted calls to a single entry. To access the queue, the server accepts each waiting entry call and, if the request it represents cannot be satisfied, instructs the requestor to reenter the queue via another entry call. If the entire queue is to be inspected, the entire queue must be cycled into and then back out of the server's address space request-by-request. Note that maintaining the queue in some useful order or keeping pointers into the queue is impossible.

Discussion. This Linda example strongly suggests CSP, and it is easy to imagine that a CSP solution would be nearly the same as the Linda version. This

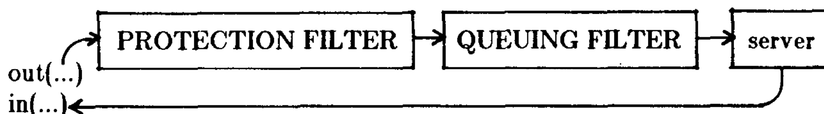


Fig. 6. A protection filter in place.

is not the case, because CSP lacks Linda's free-naming property and therefore cannot support continuations in a general way. The Linda solution depends on the requestor passing its name to MRC_ENQUEUE, which passes it via the queue to MRC_DEQUEUE. In CSP, names cannot be passed as parameters; the only way the requestor can identify itself to MRC_ENQUEUE is by passing either an index into a process array or some other parameter that allows the server to pick it out of a statically-encoded list of possibilities. (Note however that extensions to CSP that introduce ports and allow port names to be passed as parameters—see Kieburtz and Silberschatz [21], for example—do allow this kind of solution.)

Of the systems surveyed here, CSP, DP, Ada, and Starmod do not allow general continuation passing. ECLU evidently does not. SR and Plits do (although we argue elsewhere [12] that their formalisms are considerably more complex than Linda's).

We conclude this section by considering briefly the question of access to tuple names. A single distributed program encompasses many processes, and they all share a single name space and a single TS. But when two *different* distributed programs—say, a user program and the operating system—run on the same network, each defines its own name space, and constituent processes of the first program ordinarily have no access to tuples generated by the second. By prefixing the internal representation of tuple names with program id's, the kernel protects each program's set of tuples from reference, augmentation, or deletion by any other program. There are clearly times, however, when a user program must have access to names in the system's or in some other user's name space, and so it must be possible to import external names or to receive them as parameters. Programs may, however, specify, and the kernel must enforce, limitations on the uses to which exported names may be put. For example, the system might export the name "disk_request" for use in out( ) statements only; the kernel then traps any `in(disk_request, . . .)` or `read(disk_request, . . . )` statements that occur outside the system. The system is free to protect itself further by installing protection filters in front of servers. The name "disk_request" might for example be exported, but the statement `in(disk_request, . . . )` might appear in a protection filter that checks the identity of the requestor against an access list local to the protection filter. If the requestor has access rights, the public name "disk_request" is stripped off, and the request tuple is forwarded under a private (nonexported) name to the queueing filter. Otherwise, an exception is generated or bad status is returned (Figure 6).

2.5 Examples with Structured Names

2.5.1 *Conversational Continuity: Structured Naming.* "Conversational continuity" as described by Liskov [23] refers to the following systems-programming problem: (1) There are many identical instances of some server. (2) A user may start a conversation with any instance, but having chosen one, should converse

with that one exclusively for the duration of the conversation. (3) The multiplicity of servers, the process of selecting one, and the obligation to stick with it should be transparent to users.

A Linda solution:

servers execute:
initially:
```
in(server, U:name, start);
{from any user, accept a 'start' request}
```
for the duration of the conversation:
```
in(server, U, r:request)
{from user 'U', accept any request (except for 'start')}
```
users execute:
initially:
```
out(server, me, start);
```
for the duration of the conversation:
```
out(server, me, request).
```

2.5.2 *Node Allocation: Inverse Structured Naming.* As a refinement to the simple idle-node allocation scheme discussed above, task-generators may need to direct tasks to specific nodes in some cases and to any idle node in others. Network node i executes:

```
in(idle, i, j:job_description);
```
execute job "j".

To direct task "k" specifically to node i:

```
out(idle, i, k);
```

to direct task "k" to any idle node:

```
out(idle, n:integer, k).
```

2.5.3 *Reminder Messages: Time-Uncoupling and Inverse Structured Naming.* Consider the following routine:

```
def REMINDER
   [now = current date and time:
    in(tick) =>
      [now++; {increment the current time}
       in(msg, now, s:string) =>
       print string s
      ]
   ]
```

When REMINDER is started—we assume that it is one component of an and-statement, some of whose other components need to use it—"now" is initialized to the current date and time. (We assume that both are encoded as a single integer, but only for maximum simplicity of the presentation. Date and time could as well have been represented as separate fields and updated appropriately.) REMINDER now waits for ticks. Whenever the singleton tuple "tick" appears in TS, it is withdrawn and the value of "now" is incremented. After each incrementation, REMINDER looks for every tuple in TS whose first component is "msg" and whose second component is the current date and time. Each such tuple is withdrawn, and its third component is printed.

The statement out(tick) is executed in response to (or, is the logical equivalent of) a hardware clock-interrupt. REMINDER is a representative interrupt-handler. Correct realtime response depends, of course, upon scheduling policies implemented in the kernel. It will be useful to allow hardware-determined priorities to be associated with in( ) statements (as, e.g., they are associated with device modules in the Concurrent language Modula [27]), but we will not discuss this here.

REMINDER is the basis of a reminder-message system that works as follows. If I want to be reminded of "*s*" on date-and-time *t*, I execute out(msg, *t*, *s*). Thus, for example,

```
out(msg,
    at('June 5, 2:50PM'),
    'Faculty meeting in ten minutes. Beat it.'
    )
```

"at( )" is a function that translates a date-and-time specifying string into an equivalent internal representation as a single integer. (at( ) needs to be able to read the current time in order to do this; we discuss how to read the current time in Section 2.6.2.) On June 5 at 2:50 PM, REMINDER will fish this tuple out of TS and print the specified message. (Note that TS acts in this case as a content-addressable file system. We discuss some implications in Section 2.7.)

Suppose now that date-and-time is represented internally not as a single integer but as a series of fields—year, month, date, hour, minute, sec, ms. In this case,

```
out(msg,
    at('June 5, 2:50PM'),
    'Faculty meeting in ten minutes. Beat it.'
    )
```

is equivalent to

```
out(msg,
    84, 6, 5, 2, 50, 0, 0,
    'Faculty meeting in ten minutes. Beat it.'
    ).
```

(Note that we allow a function to return a tuple.) REMINDER will withdraw and print this message only if the user is logged on at 2:50 on June 5. Suppose instead that the message should appear *some* time on June 5, rather than at 2:50 precisely. In this case we can write:

```
out(msg,
    anytime_on('June 5'),
    'Faculty meeting at 3:00 today.'
    ),
```

and "anytime_on( )" can produce the following translation:

```
out(msg,
    83, 6, 5, i:int, j:int, k:int, l:int,
    'Faculty meeting at 3:00 today.'
    ).
```

Plits includes a specialized form of restricted-input statement. SR includes a restricted-input statement that is more general than Linda's in( )-statement actuals in allowing arbitrary boolean expressions to be enforced over input parameters, but less general in that, since SR does not allow distributed naming, the restricted-input statement cannot be used to steer parameters to certain nodes and away from others. None of the other languages surveyed here allows the equivalent of inverse structured naming.

A larger example of the use of time-uncoupling, as well as of continuation-passing and structured names, is given in the Appendix.

2.6 Examples Using read( )

2.6.1 *Global Variables: Distributed Sharing.* A checkers-playing program, implemented on the Arachne network computer, contains multiple processes that search the tree of future game states in parallel. Given parallel processes *i*, *j*, and *k*, processes *j* and *k* are to be stopped if and when they are known to be exploring the consequences of a less-good move than *i* has already found. Process *i* must notify *j* and *k* when this is so.

> If Arachne allowed shared variables between processes, the value of the best move found so far could be put in a global location. However, lacking global memory, we arrived at a different solution. [10]

Linda allows the simple solution to be programmed directly. To read the value of best-move:

```
read(best_move, m:move).
```

To change the value of best-move:

```
in(best_move, old_move:move);
out(best_move, new_move).
```

(The code actually required by a distributed game-playing program would be considerably more complex than this.)

2.6.2 *Delay Statement: Structured Names and Distributed Sharing.* A global clock might be stored in TS in the form of a clock, time-value tuple. Assume that time values are integers and that each tick increments the clock by one. To read the clock, processes execute

```
read(clock, now:integer).
```

To write the clock, the clock process executes

```
in(clock, now:integer);
out(clock, now+1).
```

Processes might find a delay statement useful, where DELAY(*d*) delays the executing process for (at least) *d* time units. A conventional implementation of a delay service might send delay requests to a central server that stores them on a wake-up-time-ordered queue. This solution is simple enough to implement, but there are simpler solutions in which the request queue is distributed throughout

TS. DELAY is the following calling routine:

```
def DELAY(d:integer){delay for at least d ticks}
  [ read(clock, now:integer);
    out(delay_until, now+d, me);
    in(me)
  ]
```

At each clock tick, the wake-up server increments the clock and awakens all processes delayed until "now":

```
in(tick) =>
  {update the clock}
  in(clock, now:integer);
  out(clock, now+1);
  {awaken processes delayed until time = now+1}
  in(delay_until, now+1, who:name) => out(who)
```

(Under some circumstances, the following is even simpler:

```
def DELAY(d:integer)
  [ read(clock, now:integer);
    read(clock, now+d)
  ]
```

The *first read( ) statement consults the clock and assigns the current time to* "now". The second specifies a structured name that will match the clock tuple and cause the read( ) statement to continue when the time reaches now +*d*. At every tick the wake-up server simply increments the clock. This solution will work correctly, though, only if the clock period is long relative to the amount of time it takes to execute an in( ) followed by an out( ). Otherwise the clock process may remove tuple `(clock, t)` from TS before all `read(clock, t)` statements have had a chance to see it—in which case those read( ) statements that have *not* seen it are stuck forever.)

2.7 Time Uncoupling, Tuple Persistence, and File Systems

Space-distributed computing is an innovation, and its investigation has barely begun. Time-distributed programming languages in Linda's sense are likewise unfamiliar, but time-distributed computing itself is nothing new. Communication between time-disjoint programs has long been provided for by file systems. There is also a linguistic model for time-distributed communication in any routine that retains state between time-disjoint invocations—coroutines, generators, procedures with Algol own variables, monitor routines with access to permanent monitor variables, and so on.

In Linda, extension of a linguistic model into fully space-distributed computing results in its extension into time-distributed computing as well; the linguistic aegis now covers certain filing functions that are traditionally extra-linguistic. Generative communication logically encompasses a tuple-structured, content-addressable file system. But the extent to which it is desirable or practical to incorporate filing functions in Linda is not yet clear. At present, we assume that when all processes of a distributed program have completed, whatever tuples generated by that program remain in TS will be deleted, with two exceptions:

tuples whose first names are in the system's name space and tuples whose first names appear on a "save" list specified by the programmer.

2.8 Summary and Conclusions

We noted at the start that generative communication results in a number of properties that distinguish Linda from other languages. We summarize and reiterate here.

Distributed naming. None of the other surveyed languages supports distributed naming. If multiple, identical, space-disjoint servers exist in a network, client processes must refer to each under a different name, or send requests to a concentrator process that sends them onward to the distributed servers. Note that the latter scheme imposes centralization in a distributed environment on linguistic grounds.

Time uncoupling. None of the other surveyed languages supports time-uncoupling in Linda's sense. Communication between time-disjoint processes is supported in other systems by the file system, not by interprocess communication mechanisms.

Distributed sharing. None of the other surveyed systems supports distributed sharing. Variables shared between disjoint processes are implemented by processes or modules.

Continuation passing. Linda, Plits, and SR all support continuation passing. Continuation passing style is to be contrasted in the systems domain to input guard style. Input guard languages admit to the server's address space only those requests that may be served immediately. Blocked requests, and consequently blocked processes and blocked-process queues, are the domain of the kernel and are inaccessible to the programmer. (Note that SR resembles the noncontinuation languages in providing input guards, and relying on them in presented examples, for resource-server implementation. Plits' strategy for programming request-queue problems is not clear.)

Structured names. None of the other surveyed systems supports structured naming in Linda's sense (although Plits and SR both include restricted-input statements).

Linda's weaknesses stem from its strengths: programmers are required to implement structures such as calling routines and queue-managing filters provided for them in other languages. Properties such as distributed naming and sharing make Linda inherently less secure than other languages: care must be taken to prevent unauthorized access to stored tuples. Finally, Linda presents difficult and novel implementation problems, particularly in a distributed environment. One central problem dominates the implementation question: How is TS to be implemented in the absence of shared memory? We address this question in Part II.

Several important topics have been omitted in this survey, and, in conclusion, we mention some of them briefly. *Modularization* is important in the construction of large distributed programs, and Linda supports it; the necessary mechanisms are discussed in [15]. *Type-checking* is also particularly important in building large systems. The definitions above do *not* require that a given name be used

with only one type of tuple; the two statements `out(P, FALSE)` and `out(P, 10)` may legitimately appear in the same program. This flexibility is sometimes useful, but it makes runtime type-checking impossible. A process that executes `in(Q, i:integer)` when it meant to say `in(Q, i:char)` is in danger of hanging forever. It would, however, be easy enough to allow user programs that so desire to run in a "type-checked" mode, in which the kernel signals an exception whenever a tuple and a blocked in( ) statement specify the same first name but do not match because of type incompability. *Dynamic process creation* is the one capability of the generative communication operators themselves that we have not discussed. Briefly, any side-effect-free expression may appear in a generated tuple, and tuples are added to TS in *unevaluated* form. As soon as a tuple enters TS, evaluation of all its unevaluated components begins simultaneously. Thus the statement `out(P, f(x))` generates a new process; its name is "*P*" and it executes `f(x)`. If `f(x)` returns an integer, the result tuple may be removed from TS by the statement `in(P, i:integer)`. (If in( ) executes while `f(x)` is still under evaluation, in( ) blocks until evaluation is complete.) Dynamic process creation is a powerful tool that entails difficult implementation problems, discussed in [13].

3. PART II: A VIRTUAL LINDA MACHINE ON A NETWORK COMPUTER

We are concerned here with aspects of the implementation of Linda on networks, particularly on network computers. Our only initial assumption about the target machine is that all of its nodes are essentially equal in power. Study of the Linda implementation problem leads in top-down fashion, however, to a class of interconnection topologies and communication architectures that are well suited to a Linda-supporting network computer.

The techniques described below were designed for and partially implemented on the SBN network computer. We do not, however, discuss this implementation, but the principles behind it. The following is a general outline; a full treatment appears elsewhere [16].

3.1 A Virtual Linda Machine

We propose to think of out( ), in( ), and read( ) as the primitive instructions of a virtual Linda machine (VLM) that is implemented by the hardware, the network communication kernel, and the compiler in cooperation. (The communication kernel is the program resident on each network node that implements internode communication.) Implementing relatively high-level computation operators as virtual-machine primitives is an established technique, but implementing communication operators in analogous fashion is not, and requires some introduction. Communication operators are usually regarded as tools provided by the system. But we intend to write operating systems *in* Linda. An implementation of Linda's communication operators must therefore underlie the system—must be provided, that is, by the hardware and the communication kernel.

In constructing a distributed VLM, the central problem is, as noted, how to implement TS in the absence of shared memory. This question resolves into two subproblems: how to find tuples (the major issue) and where to keep them.

3.2 How to Find Tuples: The Dynamic Global Name Space

A name space that is global to all modules of a distributed Linda program makes it impossible to resolve global-name references statically before runtime. Suppose a distributed program is running on j network nodes and out($+t$) is executed on one of them at time q. (Recall that if $-t$ is an in( ) or read( ) statement's parameter list, $+t$ is any tuple that can be legally received by that in( ) or read( ) statement.) At time q, processes are suspended at in($-t$) statements on some (possibly empty) subset S of the j nodes occupied by the program, but it is impossible to determine the identity of S at compile- or load-time. Some method must therefore be provided for matching $+t$ tuples to in($-t$) statements at runtime, or, in other words, for implementing a dynamic global name space.

Providing such a name space has an important additional benefit as well: it makes it unnecessary, so far as name resolution goes, to assign modules of a distributed program statically to network nodes. Allowing the network computer's operating system maximal freedom to locate and relocate distributed jobs within the network is in fact an important goal in itself. Static mapping of modules to nodes requires that distributed programs be relinked each time they are run and rules out the possibility of new modules being generated dynamically by running programs, making it impossible for the system to reposition modules in the network in response to hardware failure, changing resource requirements, or a changing network environment. Enforcement of a static mapping policy thus requires *ipso facto* the surrender of a significant portion of a network computer's potential power and flexibility—a loss roughly comparable to what might be engendered by a conventional system's requiring object modules to reside at fixed, absolute addresses.

Our VLM's first requirement, then, is to provide a dynamic global name space. Runtime resolution of name references may be handled in either of two general ways. Name-node mapping information may be concentrated in central directory nodes, or it may be distributed network-wide, for example by means of a network-wide broadcast.

The maintenance of a central directory might lead to congestion and vulnerability; it is furthermore inappropriate as a kernel function. Ideally, the kernel is a small, simple program basically identical on every node. Distinguishing certain kernels as directory servers, and requiring all others to submit service requests to these, involves the kernel in an undesirable degree of complexity and specialization. If the kernel is to implement the global name space—and it must, because Linda is to be supported directly by the kernel—then distributed storage of name-node mapping information is an attractive approach.

The combined weight of these design decisions is to make the mapping of logical to physical addresses a function of the (virtual) machine itself. Conceptually, program modules and the names they define may be mapped to nodes and moved between nodes as freely as pages may be moved among page frames in a paged memory system. To support this flexibility, the paged machine translates virtual to physical addresses in the hardware on a per-reference basis. The analogous function in the VLM is the translation by the communication kernel of tuple names to network addresses on a per-reference basis. Against the

potential benefits of this flexible architecture must be balanced the cost and complexity of implementing internode communication and runtime resolution of name references. The power of the VLM will be realized only if the communication kernel can be implemented efficiently. We address specifically the issue of efficient name resolution in the remainder of this section.

We define elsewhere [16] a method of network state storage called "uniformly distributed," where "network state" is the collection of all data descriptive of any particular node that may be of interest to the network generally. The fact that a tuple with name M is currently required by an in( ) statement on node i is an instance of a state datum.

Informally, a network state storage scheme is uniformly distributed if no node stores any more of the current state than any other node. Uniformly distributed state storage schemes fall into a discrete spectrum. At one end of the spectrum, state data is broadcast to every node, and each node maintains its own complete copy of the state of the entire system. At the other end, state information remains local to the node on which it originates. In this case, to discover the current state, a given node must broadcast a query to the entire network; nodes respond to this broadcast query with information extracted from their local states.

If state data is read much more frequently than it is written, then clearly the first scheme is more efficient in terms of total communication and processor bandwidth expended by all reads and writes. In this scheme, writing is expensive (a broadcast is required), but reading is cheap. If writes are much more frequent than reads, then the second scheme will be more efficient. If, on the other hand, reads and writes occur with roughly equal frequency, then a third scheme, intermediate between the first two, is more efficient than either. ("Roughly equal" has a precise definition: On an N-node net, roughly equal means equal within a factor of $N^{1/2} - 1$, as discussed in the general treatment [16].) In the intermediate scheme, assuming an N-node network, writing the network state consists of broadcasting state data to a prearranged set of $N^{1/2}$ nodes, called the write set. Reading the state consists of broadcasting a query to a different set of $N^{1/2}$ nodes, called the read set. If the read set is chosen correctly, its nodes will store among themselves the entire network state. This condition is assured if each node's read set has a non-null intersection with each other node's write set.

Consider now the problem of implementing in( ) and out( ). By definition, the operation writestate($-t, j$) will add to the network state the datum "$-t$ is available on node j"—in other words, that in($-t$) has been executed on node j and that some instance of $+t$ is accordingly desired. The operation readstate($-t$) will consult the network state and return the identity of some node on which in($-t$) has been executed and on which an instance of $+t$ is accordingly needed. If there is no such node, the readstate($-t$) request will remain outstanding until there is such a node and will then return that node's identity. (Note that "readstate" and "writestate" are internal to the communication kernel. They are not part of Linda.) Abstractly, then, node i's kernel may implement an out($+t$) operation as follows:

```
j = readstate(-t);
send +t to node j.
```

in($-t$) as executed on node j is implemented

```
writestate(-t, j);
await receipt of +t.
```

Note that at this level the implementation of in( ) and out( ) is not atomic. As a result, two instances of $+t$ may be sent to the same node j, in which case the second instance must be reinserted into the global buffer.

Let "$-t$-state" refer to all data in write sets pertaining to the whereabouts of $-t$-tuples (i.e., of in($-t$) statements) in the network. If in( ) and out( ) are executed with roughly equal frequency—this seems to be a reasonable assumption, although there are complicating factors, which we discuss below—then the network's $-t$-state will be read and written with roughly equal frequency. Given that the $-t$-state is to be stored in uniformly-distributed fashion, writestate($-t$, i) and readstate($-t$) are accordingly best implemented using the intermediate $N^{1/2}$-node broadcast technique.

Our basic tools for implementing in( ) and out( ) are readstate( ) and writestate( ). readstate( ) and writestate( ) may be implemented using any state storage scheme on any type of network, large or small. The $N^{1/2}$-node broadcast technique in particular suggests, however, one class of network topologies. Network-computer designers are generally free to specify an interconnection topology; designers of larger networks are not. The remainder of the discussion therefore applies specifically to network computers.

We describe the class of topology we have in mind as the W^D wrap-around hypercubes, where "wrap-around" refers to each dimension of the network's being connected either as a ring or by a broadcast bus, so that all nodes interface to the network in the same way; there are no edge nodes. Linda was developed, as noted, for the SBN network computer, which used a two-dimensional representative of this class. SBN was a linked torus—a square end-around grid in which each row and each column loops back on itself; each node has four nearest neighbors. In the following, we describe the Linda implementation designed for SBN. It is applicable to any square grid or, with appropriate simple generalizations, to any even-dimension W^D wrap-around hypercube.

For each node i, i's write set is its row and i's read set is its column. When in(P. . .) executes on node j, j's kernel executes writestate($-P$, j) by sending each of the $N^{1/2}$ nodes in its row a message reading "a tuple labelled 'P' is needed on node j". This process is referred to as *constructing a P-in-thread.* (Note that we deal at this point only with simple names. Structured names are discussed below.) When out(P. . .) executes on node i, i's kernel sends to each of the $N^{1/2}$ nodes in its column a message reading "a tuple labelled 'P' is available on node i"; this process is called *constructing a P-out-thread.* When a P-in-thread either intercepts or is intercepted by a P-out-thread on node k, a notification message is generated on node k and sent to node i instructing i to send tuple (P, . . .) to node j. When and if the tuple is delivered, accepted, and acknowledged, both P-in-thread and P-out-thread are removed. Some other (P, . . .) tuple may, on the other hand, arrive at node j before node i's does. If so, node i's tuple is not accepted and i's P-out-thread remains in place. (Figures 7 and 8.)

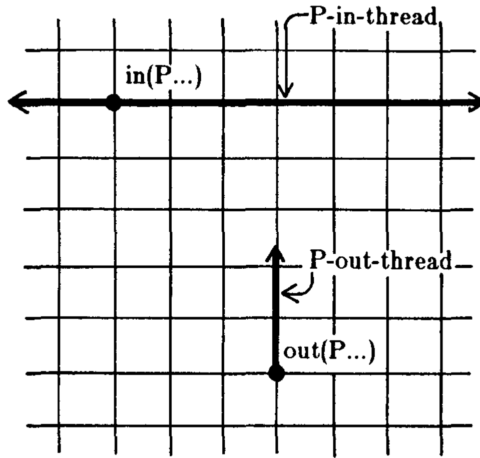


Fig. 7. out $(P \ldots)$ executes; in $(P \ldots)$ has already executed; a P-out-thread looks for a P-in-thread.

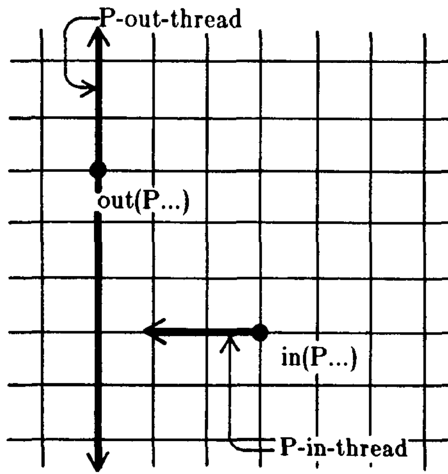


Fig. 8. in$(P \ldots)$ executes, but out $(P \ldots)$ has already executed, so a P-in-thread looks for a P-out-thread.

Note that an out-thread registers data in all nodes of a read set just as an in-thread does in a write set. But the data recorded in an out-thread are treated as a persistent state query, not as part of the network state.

We have described an implementation technique for in( ) and out( ) that is optimal if in( ) and out( ) are equally frequent within a factor of $N^{1/2} - 1$. The technique requires that an in-thread be constructed each time in( ) executes and an out-thread be constructed each time out( ) executes. The overhead of con-

structing in- and out-threads may, however, be substantially reduced in the following ways. First, once node i has determined that tuple $(P\ldots)$ is needed on node j, it associates "P" with "j" in a name-address cache and sends all subsequent $(P\ldots)$ tuples directly to j, bypassing out-thread construction until j rejects one such tuple. (This scheme is analogous to the runtime conversion of logical to physical addresses in a virtual memory system.) A related factor decreases in-thread construction. As discussed in Part I, active procedures are a basic structure in Linda programs. Active procedures are headed by in( ) statements to which execution returns repeatedly. In-threads associated with these in-statements need not be repeatedly torn down and reconstructed; they may be left in place for as long as the associated procedure block remains invocable.

read( ) statements are implemented precisely as in( ) statements are, so far as in-threads go. But regarding the associated out-threads there are two possibilities. Suppose read($-t$) executes on node j and out($+t$) has been executed on node i. Execution of read($-t$) might cause a copy of $+t$ to be transported to j, leaving the original undisturbed on i. On the other hand, the original might itself be transported to j, in which case subsequent read($-t$)s will find it there—not, in other words, on its node-of-origin i, but on its node-of-last-reference j. In the former case, the $+t$-out-thread remains undisturbed with its origin on i. In the latter, i's $+t$-out-thread is torn down and j must construct a new one with itself as origin. Clearly, the second scheme involves greater thread-construction overhead than the first, but it has what might be the desirable property of distributing tuples to the network sites where they are needed, rather than leaving them forever at their creation sites. A node that stores a collection of tuples that are referenced from many points in the network is a potential bottleneck. Note that the first scheme has the effect of unbalancing queries and updates over the $-t$-state by increasing in-thread construction (updates) relative to out-thread construction (queries). If this scheme is to be implemented and a significant imbalance is anticipated, the size of read and write sets may have to be adjusted as described in [16] in order to maintain good performance with uniform distribution.

Threads as described above perform only simple name matching, ensuring that a tuple is delivered to an in( ) statement whose first component matches the tuple's. But that in( ) statement may in fact specify a structured name, and if the arriving tuple fails to match all fields of the structured name identically, it will be rejected and must be retransmitted to another candidate in( ) statement. A more efficient implementation requires the propagation not just of names but of structured names in in-threads. A structured name is propagated as a list of fields that a tuple must match in order to be acceptable to a given in( ) statement; out-threads continue to propagate simple names only. Notification messages generated when in- and out-threads intersect contain a copy of the structured-name list associated with the in( ) thread, and no tuple that fails to match a structured name is transmitted.

3.3 Buffering

We close with a brief discussion of the second of the two questions raised at the start of Part II: Where should tuples in TS be stored?

The VLM is required to simulate Linda's infinite global buffer. System buffer space must expand as needed; buffer space must be provided for undelivered tuples. These requirements are satisfied by making the allocation of buffer space largely the function of compiler-generated code.

Allocation of buffers for arriving tuples is in principle a simple extension of the allocation of activation frames in the implementation of sequential languages. When in(P...) executes, a buffer for the actuals is allocated on the execution stack of the process executing in( ), and is referred to as buffer P. An arriving tuple is stored in buffer P after match and delivery have occurred. Outbound tuples are stored in per-process heaps maintained by compiler-generated code until they are removed from TS. Header information required by the communication system—dealing with the length of the tuple, its logical address, and other basic control characteristics—is prefixed to the buffered outgoing tuple by compiler-generated code.

If a process terminates with undelivered tuples, responsibility for the buffers in which they are stored devolves on the kernel, and these buffers are not deallocated until their contents have been delivered. (When *all* processes of a given distributed user job have terminated, however, that job's undelivered tuples may be erased, except for tuples whose first names appear on the "save" list or are imported from the system's name space, as mentioned earlier.)

3.4 Hardware Failures

The semantics of Linda in the presence of hardware failures and the implementation of TS in a robust way are important issues that are now under study. We include a few introductory comments only.

First, "reliable message delivery" is not *per se* at issue in Linda, because Linda does not undertake to deliver messages. out($+t$) works correctly if and only if subsequent to execution $+t$ has been inserted in TS. out( ) fails only if the node on which it is executed crashes before $+t$ can be handed safely to the kernel on the same node. $+t$ may subsequently be lost or destroyed, but this represents a failure of TS, not of out( ); no bad status or failure exception can or should be reported to the process that executed out( ). Similarly, in($-t$) fails only if an instance of $+t$ exists in TS but cannot be found. If an instance of $+t$ *used* to exist but has been lost, it is once again TS and not in( ) that has failed. What are the chances, then, that a $+t$ inserted by out( ) will be safely delivered to an in($-t$)? The system will not guarantee delivery, but should promise highly probable delivery. A system in which reliable delivery is critical may implement (in Linda) one of the handshake protocols that allow reliable communication over unreliable channels. We do not anticipate that many Linda programs will do this. In a usable implementation of TS, failures will be rare enough not to have to be taken into account in the course of normal program development—in the same sense in which system failures, file crashes, and so on take place but do not impede normal program development on a useful conventional installation.

Maintaining TS reliably is therefore the central issue, and it is made up of two subproblems: remembering where tuples are and remembering the tuples themselves. Note that the $N^{1/2}$ broadcast scheme for remembering tuple whereabouts is in general, because of its substantial redundancy, the potential basis of a highly

reliable system. A node in this scheme stores data on the whereabouts of $+t$ and $-t$ tuples; its $+t$ data are identical to the $+t$ data stored by the nodes above and below it in the grid, its $-t$ data to the $-t$ data stored on the nodes to either side. A node that fails and is rebooted can therefore restore the correct tuple-whereabouts state by appropriate queries to neighbors; while a node is down, out- and in-threads must be routed around it. In general, an entire row or column must fail for tuple-whereabouts data to be lost.

The $N^{1/2}$ broadcast scheme is redundant with respect to tuple-whereabouts data but not, as we have described it, to the values of the tuples themselves. Each is stored, until an in( ) statement seeks it, on the node where its associated out( ) was executed. If that node fails, the tuple is lost. We are currently investigating the ramifications of a scheme that would be equally redundant with respect to tuples and tuple whereabouts: each tuple added to TS by out( ) would be broadcast to every node in the executing node's read set (i.e., to every node in the executing node's column on a grid). To assure the atomic character of out( ) and in( ), all in($-t$) requests would still be directed to the node where out($+t$) executed; only this tuple-keeper node is authorized to satisfy in($-t$) requests, and once such a request is satisfied, $+t$ is deleted from every node in the read set. But if the tuple keeper fails, any other node in its read set may become the tuple keeper in its stead. Every node in the read set must fail before a tuple is lost. This scheme makes heavy space and communication-bandwidth demands on the network, and presents some complex implementation problems. On the other hand, it is highly desirable with respect to reliability, and has two other major advantages as well. First, it increases the efficiency of read( ): $N^{1/2}$ copies of each tuple are available in the network for reading purposes. Second, it increases the efficiency of implementing structured names: an in($-t$) with a structured name can check with any node in the read set to determine whether a $+t$ matches; it need not go to the unique tuple keeper to find out.

4. CONCLUSIONS

We have discussed generative communication and one technique for implementing it on network computers. The technique we developed has attractive properties relative to other possibilities—but we have not demonstrated, needless to say, that is is acceptably efficient in absolute terms. Generative communication might place unacceptable burdens on network communication systems. No evidence to the contrary will be available until implementation projects now in progress are more advanced.

We do nonetheless find Linda well worth pursuing. After all, thinking up good uses for ever-more-plentiful basic resources—two of which are short-haul communication bandwidth and communication-processor power—is contemporary system design's fundamental problem. We are willing to bet that it is a mistake, at least in the long term, to allow our thinking to be cramped by preconceptions about what is and is not implementable. If we are mistaken in our optimism, then at least we will have examined some interesting problems in a fresh light. We insist only (and others will dispute the claim) that languages do not become interesting because we can implement them. Implementation problems become interesting when new languages seem worthy of study.

A thread-addressing, packet-switched communication kernel to support Linda was implemented in part on the 4-node prototype SBN network computer at Stony Brook. But SBN's unsophisticated hardware proved unsatisfactory, and the project is quiescent. Linda now runs on an Apollo workstation at Yale; this uniprocessor implementation will be used to test the language's utility and expressivity and to study the runtime behavior of distributed programs. In a joint project involving the Yale Linda group and researchers at AT&T Bell Labs, Linda is being implemented on a network computer designed and built at Bell. The results of this project will tell us a great deal about whether generative communication can be made to work efficiently.

APPENDIX 4.1. Deadlock Detection in Two-Phase Locking: Time Uncoupling, Continuation Passing, and Structured Names

In the following we discuss an example that is short and fairly simple, but not trivial. Following it will require brief contemplation of two-phase locking, which is not the subject of this report. We include it nevertheless, to illustrate the way in which Linda encourages simple solutions that are much harder to express and, correspondingly, less likely to occur in other languages.

Consider a set of database transaction processes $\{T, U, V \dots\}$ and a set of data locks $\{A, B, C \dots\}$. Transaction processes set and clear locks according to the two-phase locking protocol [7]. An *access graph* for this system is a directed bipartite graph (V_1, V_2, E). Nodes in V_1 correspond to transaction processes and nodes in V_2 to locks (see Kohler [22]). Let $t \in V_1$ and $l \in V_2$. $E(t, l)$ (an edge from t to l) means that transaction t is currently requesting lock l. $E(l, t)$ means that lock l is currently assigned to transaction t. A deadlock exists in the system iff there is a cycle in the access graph.

The Linda solution detects deadlock by turning the set of transaction processes and lock processes into a life-size (as it were) transaction graph. Each process acts as a node in the graph, and processes exchange tokens in order to detect cycles. The solution depends upon the fact that at any given time (i) a transaction node may have more than one incoming edge but only one outgoing edge (because a transaction may own several locks but have no more than one unfilled lock request outstanding), and (ii) a lock node may likewise have more than one incoming edge but only one outgoing edge (because many transactions may be requesting a lock, but only one transaction owns it). The solution works generally as follows:

transactions execute: `in`(*from all currently-owned locks*);
`out`(*to unique currently-requested lock*).

locks execute: `in`(*from all currently-requesting transactions*);
`out`(*to unique current-owner transaction*).

A transaction T requests a lock by sending its name to the manager of the lock it is requesting. It then waits at an in( ) statement, expecting some name n in return. If it receives n and n = ok, then T now owns the lock and may continue. ("ok" is a reserved name that indicates awarding of a lock; no transaction may be named "ok".) If, on the other hand, it receives n and $n = T$, there is a deadlock in the system: T's name has travelled a cyclic path through the graph and

returned to its node of origin. At this point some corrective action must be taken, but we are not concerned with what it is. If T receives n and n is neither "T" nor "ok", T forwards n to the manager of the unique lock it is currently requesting and reexecutes the in( ) statement.

Transaction S will execute:

to acquire lock B:
`if (not LOCK(S, B))` *there exists a deadlock*;
to free lock B:
`UNLOCK(B).`

LOCK( ) is a calling routine that returns a value, just as a called routine might.,

```
def LOCK(T:name, A:name)
    [n:name
    &  [out(get, A, T); {enter a request in T's name for lock A}
        [done:boolean = false
        & do
          [{await word from A or from currently-owned locks:}
           in(T, var n);
           done = (n == ok) or (n == T);
           if (not done) out(forward, A, n)]
        while (not done)
       ]; val(n == T){the value of LOCK is the value of this
                       comparison}]
    ]
def UNLOCK(A:name)[out(free, A)]
```

The lock manager for lock A executes:

```
[ LOCK_DECL
&*[LOCK_ENQUEUE(A)|LOCK_FORWARD(A)|LOCK_DEQUEUE(A)]]
```

where

```
def LOCK_DECL
[free:boolean = true
 & owner:name
 & declarations for a request queue; frames of type name
]

def LOCK_ENQUEUE(L:name)
    [in(get, L, requestor:name); {accept a lock request}
      [n:name
      & [if (free) {award lock to the requestor:}
           [owner = requestor;
            free = false;
            n = ok] {prepare to notify the new owner}
         else [add "requestor" to the request queue;
           {pass the blocked requestor's name to the current
            owner:}
                n = requestor];
         out(owner, n)]
    ]

def LOCK_FORWARD(L:name) =
    [in(forward, L, n:name);
     out(owner, n)]
```

```
def LOCK_DEQUEUE(L:name) =
    [in(free, L);
      [head:name
     & [if the request queue is empty
          free = true
  else [head = head of the request queue;
         out(head, ok); {notify the new owner}
         owner = head;
         delete request queue head ]
    ] ]
```

This example illustrates several properties.

Time uncoupling. LOCK_FORWARD may execute out(owner, n) without ascertaining whether an in(owner, . . .) statement will ever be executed.

Continuation passing. Continuation passing is extended in this example through a chain of length bounded by the number of nodes in the transaction graph.

Structured naming. Structured names allow a family of names—here, "get, A", "forward, A", and "free, A"—to be generated from one.

None of the other languages surveyed here allows the generation of multiple related names in Linda's sense.

ACKNOWLEDGMENTS

Thanks to Professor Arthur Bernstein, who directed the thesis in which Linda was first described, and to the referees (particularly the indefatigable "A") for useful, lucid comments. Mauricio Arango was central to the Linda implementation project at Stony Brook. Without his enthusiasm and expertise, the work described here could never have been accomplished. Nick Carriero plays a similarly pivotal role in the implementation work at Yale. Thanks also to Suresh Jagganathan and Todd Morgan for their work at Stony Brook and to Brian Weston for his at Yale. So much for technical assistance; it was Jane Backus's metatechnical contribution that was decisive.

REFERENCES

1. ANDREWS, G. Synchronizing resources. *ACM Trans. Program. Lang. Syst. 3*, 4 (Oct. 1982), 405.
2. BRINCH-HANSEN, P. The programming language Concurrent Pascal. *IEEE Trans. Softw. Eng. SE-1*, 2 (June 1975), 199–207.
3. BRINCH-HANSEN, P. Distributed processes: A concurrent programming concept. *Commun. ACM 21*, 11 (Nov. 1978), 934.
4. COOK, R. Mod—a language for distributed programming. In *Proceedings 1st International Conference on Distributed Computing Systems*, (Oct. 1979), 233–241.
5. DEMINET, J. Experience with multiprocessor algorithms. *IEEE Trans. Comput. C-31*, 4 (Apr. 1982), 278–287.
6. U.S. Dept. of Defense. *Reference Manual for the Ada Programming Language.* July 1982.
7. ESWARAN, K., GRAY, J., LORIE, R., AND TRAIGER, L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624.
8. FEHLING, M., AND ERMAN, L. Report on the 3rd Annual Workshop on Distributed Artificial Intelligence. *ACM SIGART Newsl. 84* (Apr. 1983), 3–12.
9. FELDMAN, J. High-level programming for distributed computing. *Commun. ACM 22*, 6 (June 1979), 353.

10. FINKEL, R., AND SOLOMON, M. The Arachne distributed operating system. Tech. Rep. 439, Univ. of Wisconsin at Madison, Computer Science Dept., July 1981.
11. GELERNTER, D., AND BERNSTEIN, A. Distributed communication via global buffer. In *Proceedings ACM Symposium on Principles of Distributed Computing*, (Aug. 1982), 10–18.
12. GELERNTER, D. An integrated microcomputer network for experiments in distributed programming. Ph.D. dissertation, SUNY at Stony Brook, Dept. of Computer Science, Oct. 1982.
13. GELERNTER, D. Three reorthogonalizations in a distributed programming language. Tech. Rep., Yale Univ., Dept. Computer Science, Aug. 1983.
14. GELERNTER, D. A note on systems programming in Concurrent Prolog. In *Proceedings 1984 International Symposium on Logic Programming*, (Feb. 1984).
15. GELERNTER, D. Symmetric programming languages. Tech. Rep., Yale Univ., Dept. Computer Science, July 1984.
16. GELERNTER, D. Global name spaces on network computers. In *Proceedings 1984 International Conference on Parallel Processing*, (Aug. 1984).
17. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM 21* (Aug. 1978), 666–677.
18. ICHBIAH, J.D., et al. Rationale for the design of the Ada programming language. *SIGPLAN Not. 14*, 6, Part B (June 1979).
19. KAHN, G. The semantics of a simple language for parallel processing. In *Proceedings IFIP Congress 1974*. p. 471.
20. KERNIGHAN, B., AND RITCHIE, D. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
21. KIEBURTZ, R., AND SILBERSHATZ, A. Comments on communicating sequential processes. *ACM Trans. Program. Lang. Syst. 1*, 2 (Oct. 1979), 218–225.
22. KOHLER, W. Overview of synchronization and recovery problems in distributed databases. In *Proceedings Fall COMPCON 1980*. 433–441.
23. LISKOV, B. Primitives for distributed computing. In *Proceedings 7th Symposium on Operating System Principles*, (Dec. 1979), 353.
24. SCHNEIDER, F. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst. 4*, 2 (Apr. 1982), 125–148.
25. SHOCH, J., AND HUPP, J. The "worm" program—early experience with a distributed computation. *Commun. ACM 25*, 3 (Mar. 1982), 172–180.
26. WARREN, D., AND PEREIRA, L. Prolog: The language and its implementation compared with Lisp. In *Proceedings ACM Symposium on Artificial Intelligence and Programming Languages*, (Aug. 1977), 109.
27. WIRTH, N. Modula: A language for modular multiprogramming. *Softw. Pract. Exper. 7* (1977), 3–35.

Received April 1983; revised July 1984; accepted July 1984