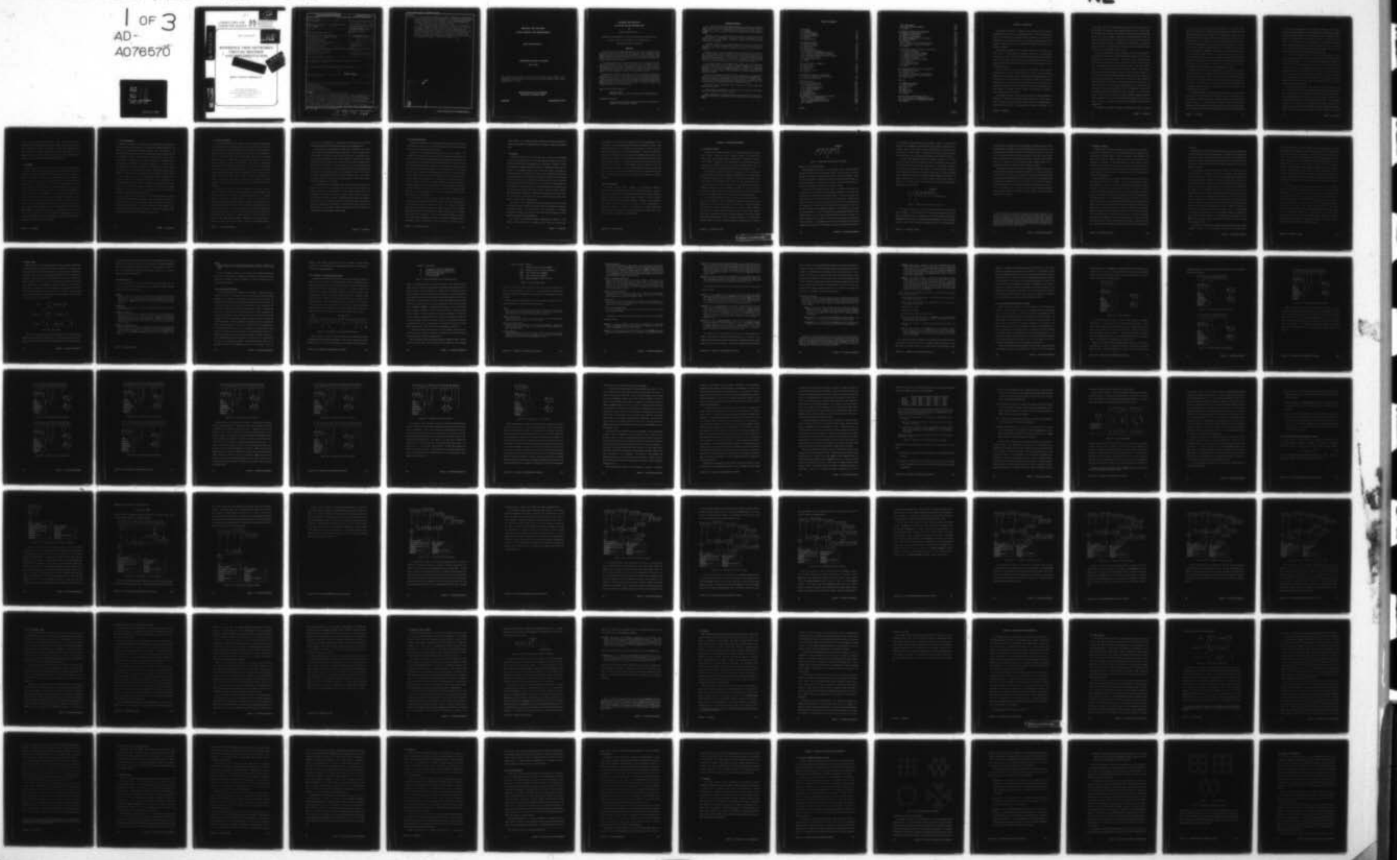
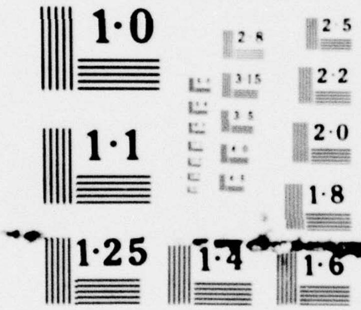
AD-A076 570 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 5/8
REFERENCE TREE NETWORKS: VIRTUAL MACHINE AND IMPLEMENTATION.(U)
JUL 79 R H HALSTEAD N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-222 NL

1 OF 3
AD-
A076570




1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
1·1
4·5
1·8
1·25
1·4
1·6

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-222

# REFERENCE TREE NETWORKS: VIRTUAL MACHINE AND IMPLEMENTATION

Robert Hunter Halstead, Jr.

This research was supported by
the Advanced Research Projects Agency of
the Department of Defense and was monitored by
the Office of Naval Research
under Contract No. N00014-75-C-0661

545 TECHNOLOGY SQUARE CAMBRIDGE MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS<br>BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>(14) MIT/LCS/TR-222 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>(6) Reference Tree Networks: Virtual Machine and Implementation. | | 5. TYPE OF REPORT & PERIOD COVERED<br>Ph.D. Thesis, July 2, 1979<br>6. PERFORMING ORG. REPORT NUMBER<br>MIT/LCS/TR-222 |
| 7. AUTHOR(s)<br>(10) Robert Hunter Halstead, Jr. | | 8. CONTRACT OR GRANT NUMBER(s)<br>(15) N00014-75-C-0661 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>MIT/Laboratory for Computer Science<br>545 Technology Square<br>Cambridge, MA 02139<br>409 648 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>(11) 2 Jul 79 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>ARPA/Department of Defense<br>1400 Wilson Boulevard<br>Arlington, VA 22209<br>(12) 248 | | 12. REPORT DATE<br>July 2, 1979<br>13. NUMBER OF PAGES<br>254 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)<br>ONR/ Department of the Navy<br>Information Systems Program<br>Arlington, VA 22217 | | 15. SECURITY CLASS. (of this report)<br>Unclassified<br>15a. DECLASSIFICATION DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(9) Doctoral thesis,

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

message passing
distributed computing
miltiprocessor systems
distributed object management
networks

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A current-technology computing machine may be roughly decomposed into a processor, a memory, and a data path connecting them. The interposition of this data path between processing and storage elements creates a bottleneck, which inhibits progress at the high-performance end of the technological spectrum. Additionally, the monolithic nature of present-day processors resists incremental addition or removal of processing power.

The research described here attacks the problem of constructing more powerful and more flexible computer systems along three fronts; the definition

→ next page

79 11 05 343

409 648

Cont.

20. of a virtual machine providing for parallel computation using objects and object references, the development of a distributed implementation mechanism ("reference trees") supporting object management functions including garbage collection, and the investigation of scheduling algorithms and collection of performance results.

A reference tree network using theses concepts is composed of a multitude of independent small processors, yet operates as a coherent programming system. Programs and data spread automatically and transparently through the network to occupy underused resources. The modular structure of the network provides many parallel data paths as well as allowing for easy addition or removal of modules, thus addressing some of the problems discussed here. A prototye reference tree network, the MuNer, is currently in operation.

# Reference Tree Networks:

# Virtual Machine and Implementation

Robert Hunter Halstead, Jr.

July 2, 1979

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge Massachusetts 02139

# REFERENCE TREE NETWORKS:

# VIRTUAL MACHINE AND IMPLEMENTATION

by

Robert Hunter Halstead, Jr.

Submitted to the Department of Electrical Engineering and Computer Science

on July 2, 1979, in partial fulfillment of the requirements for

the degree of Doctor of Philosophy.

## ABSTRACT

A current-technology computing machine may be roughly decomposed into a processor, a memory, and a data path connecting them. The interposition of this data path between processing and storage elements creates a bottleneck, which inhibits progress at the high-performance end of the technological spectrum. Additionally, the monolithic nature of present-day processors resists incremental addition or removal of processing power.

The research described here attacks the problem of constructing more powerful and more flexible computer systems along three fronts: the definition of a *virtual machine providing for parallel computation using objects and object references*, the development of a distributed implementation mechanism ("reference trees") supporting object management functions including garbage collection, and the investigation of scheduling algorithms and collection of performance results.

A *reference tree network* using these concepts is composed of a multitude of independent small processors, yet operates as a coherent programming system. Programs and data spread automatically and transparently through the network to occupy underused resources. The modular structure of the network provides many parallel data paths as well as allowing for easy addition or removal of modules, thus addressing some of the problems discussed above. A prototype reference tree network, the MuNet, is currently in operation.

Name and Title of Thesis Supervisor:

Stephen A. Ward,
Associate Professor of Electrical Engineering and Computer Science

Key Words and Phrases:

*message passing, distributed computing, multiprocessor systems, distributed object management, networks*

## ACKNOWLEDGMENTS

It is perhaps too easy, after immersing oneself in a project for a period of time, to convince oneself that the project is more important in the grand scheme of things than it really is. Nevertheless, if the work reported here has any significance at all, it is because of the selfless contributions of a large number of people.

At the top of the list must be my thesis supervisor, Steve Ward, who supplied crucial inspiration, encouragement, and support. Steve's receptiveness to new ideas, plus his continued interest, combined to make the most pleasant possible environment for this work.

Significant contributions by my thesis readers, Jack Dennis and Peter Elias, must also be gratefully acknowledged. Although I was often impatient with their suggestions, they have materially contributed to the content and presentation of the thesis.

Another unofficial "reader," Clark Baker, helped in important ways, not only by his careful reading of the thesis document, but as a sounding board and source of ideas throughout the research. Several of the examples and illustrations in the text were first suggested by Clark. Beyond these contributions, Clark took upon himself much of the drudgery of construction of the MuNet, and also generated much of the software technology used to include drawings in this document.

The remaining members of the "MuNet group" at M.I.T. were Jim Gula and Eric Strovink. Jim was the principal architect of the MuNet hardware, which didn't at all help him to finish his thesis (on operating systems for the MuNet). Eric, along with Clark, was my closest partner in MuNet debugging and testing, and also contributed the MuSpeak compiler, giving a welcome alternative to assembly language for programming the MuNet.

All the other members of the Real Time Systems group at M.I.T., and particularly Chris Terman and Tom Teixeira, aided in numerous tangible and intangible ways — with moral support, with stimulating ideas, with maintenance of the UNIX system used to develop MuNet software and produce this document, and with all the other little things that go into making a laboratory a pleasant place to be.

Others who helped formulate the ideas contained in this thesis include Carl Hewitt and Henry Baker.

I must express my gratitude to my parents for their support and encouragement over my twenty-year career as a student.

Finally, although all of the above people have done their best to improve the quality of this document, its shortcomings can be attributed to none other than its author.

## TABLE OF CONTENTS

# Chapter 1: Introduction

A good part of the history of computer science has been the story of man's struggle with the capacity of his computing machinery. At one extreme lies his drive to construct ever more powerful machines, at almost any cost. Within this expanding limit of technology, man faces the constantly changing demands on his computer systems, trying to adapt yesterday's computing structures to face tomorrow's challenges.

Both quests are hampered by the relatively monolithic nature of today's computer systems. A current-technology computing machine may be roughly decomposed into a processor, a memory, and a data path connecting them. The interposition of this data path between processing and storage elements creates a "von Neumann bottleneck"[1] which inhibits progress at the high-performance end of the technological spectrum.

Even at lower points on the spectrum, the bottleneck causes problems. Although memory size can be scaled reasonably successfully to match user requirements, the monolithic nature of the processor defies incremental modification. The best approach to this problem that is currently widely available is the concept of compatible families of processors. Using this approach, when a processor's capacity becomes inappropriate, it may be replaced by a more suitable compatible processor capable of executing the same software, thus avoiding the overhead of starting completely over from scratch.

One imaginable approach to giving processors more of the scaling properties of memories is to develop processors with multiple functional units. Thus, at least in theory, a computer's processing capacity could be altered by the simple addition or removal of functional units. This approach has a couple of drawbacks. First, commercially available processors with multiple functional units still attempt to give the appearance of executing a single sequential instruction stream, and this semantic constraint limits the number of functional units that can be used effectively. Second, even if this constraint is removed, the data-path bottleneck limits the number of functional units that can be kept busy. In essence, as more processing power and memory are added on opposite sides of the bottleneck, the effective usage rate of a unit of either must drop. The point at which this begins to happen is determined by the bandwidth of the bottleneck.

The C.mmp project[40,41], therefore, attempted to increase the bandwidth of the bottleneck by creating many parallel data paths between the bank of processing units and the bank of storage units. Still intact in C.mmp is the principle that every processing unit should be able to access every item in storage with equal ease. The programming environment presented by C.mmp is roughly one in which a number of concurrent processing units. share a common memory — not too different from the virtual machine presented by many of today's operating systems. If a user wishes to obtain the full potential throughput of the system, he must break his job into enough parallel tasks to keep all the processors occupied. Means for manually doing this are well known, if perhaps not all that easy to work with. The user is not forced to adopt any bizarre programming style, or face any situation where data is not equally accessible to all tasks. The drawback of C.mmp is that its widening of the processor-memory data path is expensive. Its 16x16 crossbar switch is a large piece of hardware, and engineering problems almost prevented it from ever working. Thus it is not clear how far that approach can be scaled up. Even if it can, its size grows as the product of the number of processing and memory units.

It is possible to generate a strongly connected (every memory unit equally accessible to every processing unit) data path without incurring quadratic expansion costs — Batcher sorting nets[4] and the arbitration and distribution nets from data flow computers[7,30] are examples of such organizations; however, the cost always grows more than linearly in the number of processing and memory units. An extreme reaction to this situation is to directly connect each processing unit to only its own private memory unit, and arrange some other mechanism for processor-memory pairs to exchange data with each other. This approach is evident in the DCS ring network[9,10], ARPANET[20], Ethernet[27], and countless other *ad hoc* telecommunications networks that exist. Such networks achieve a form of scalability in that processor-memory pairs may be added or removed relatively easily in most cases. However, there is very often a central medium which can only physically support a certain number of connections, and which can also cause a bandwidth bottleneck if large amounts of communication are required.

Perhaps more importantly, such networks often do not form coherent systems with good support for co-ordinated parallel processing. Indeed, this is not usually their purpose — they exist instead as vehicles for sharing information present at the various nodes, and accomplish even that by often arcane and special-purpose mechanisms. This detracts from their usefulness as solutions to the problem of the von Neumann bottleneck, limiting it to cases where many small and largely self-contained tasks are being run.

A compromise between these weakly connected networks just discussed and structures of the C.mmp class is the Cm* project[36]. This project avoids some of the problems of a central communications medium by attaching its processor-memory pairs as leaves of a treelike structure of "clusters." Any processor can still address any word of memory in the system, but the access time, though still fairly short, increases with the length of the path through the tree to the desired word.

As a result of this abandonment of the "equal access" philosophy, Cm* is able to achieve linear scaling behavior. The basic semantics of a shared-memory system are preserved, although there is some incentive for the user to arrange for computations to be performed near the data they reference. Two objections can be made to the Cm* approach. First, its scalability may still be limited, although the bound is probably loose enough to be primarily of theoretical interest. There is a set of "central" media that handle accesses crossing cluster boundaries; as the number of clusters grows, those media could become saturated.

Second, Cm* is an expensive way of connecting processing elements As processors and memories become smaller and cheaper, they may be dwarfed by the interconnection hardware. The cost of communication hardware will probably drop along with the cost of other components, but the ratio of communication to computing hardware in Cm* is still quite large. A principal reason for the cost of this hardware is the demand for real-time performance placed on it by the architecture of Cm*: when a processor requests a non-local memory access, it is prevented from continuing until the access has been performed. A system that did not make this demand would be able to use communication hardware chosen from a wider range on the performance spectrum.

That the Cm* system penalizes nonlocal memory accesses by a processor, relative to local references, implies that optimal performance by the network depends on a certain amount of locality among the tasks being executed on the network — the system will work poorly if memory accesses made by each processor are uniformly distributed throughout the entire memory of the system. But if locality of reference is, or can be made, a strong enough effect, perhaps the penalty on nonlocal accesses could even be increased without seriously degrading system performance. If a fairly high penalty is acceptable, it should be feasible to get by with much less exotic interconnection hardware than in Cm*. This, in fact, is the theory behind the loosely coupled networks discussed earlier. The tasks processed by these networks display a great deal of locality; internode communications are relatively expensive. Yet these networks often do their jobs quite well. The difference is that Cm* at least supports some kind of coherent methodology for making nonlocal accesses, while many networks offer none at all.

### 1.1: The MuNet

It seems plausible and useful, then, to consider constructing a network which is tailored for tasks that exhibit some locality of reference, but which still supports a coherent, transparent methodology for making nonlocal accesses to data anywhere in the system. Virtually any current networking scheme can probably be used as a base for such a system — the key element is the environment provided on each processor in the network for the execution of user software. The provision of such a software environment, which allows transparent access to, and motion of, data objects throughout the system, is a central part of the MuNet project[13,14,16,34,35,37,39]. The MuNet is a network built for the purpose of testing the various ideas reported in this thesis. Its hardware is composed of several LSI-11 processors, each with 28K words of private memory. Each processor has ports for up to four 125KBaud serial lines which can be connected to other processors to create a network configuration of some desired topology. The design of the MuNet hardware is not hailed as any advance in technology for constructing computer networks; quite to the contrary, it is simple in the extreme, since the principal goal of the MuNet project is the development of software methodologies and organizational principles for using such networks.

## 1.2: The Virtual Machine

We begin in Chapter 2 of this thesis by considering the virtual machine, or interface, through which programs can invoke internal network machinations; only later are the machinations themselves examined. This virtual machine is an abstracted and improved version of the environment provided for program execution on the MuNet and has its conceptual roots in earlier work on the "mu calculus" message-passing formalism[15,38]. A detailed description of the MuNet environment, along with a commentary on it, appears in Appendix A. Occasions to refer to the virtual machine described in Chapter 2 will arise frequently, so we succumb to the temptation to coin an acronym, and name it VIM (*V*irtual *I*nterface to the *M*uNet). This term should be understood to mean specifically the virtual machine outlined in Chapter 2, not the closely related one described in Appendix A. In broad outline, VIM is a garbage-collected object-reference system[5] which supports message-passing, or actor-style, computation. Our treatment of VIM is at two levels: first, an introduction, written in English prose, and second, an informal "blackboard interpreter," akin to the "contour model" used for block-structured languages, to make the specification more precise.

VIM is important in that it serves as a clean interface between the user's requirements and the system's capabilities, but its specification only begins to attack the problem of constructing a viable network. Nevertheless, the choice of this starting point underscores a key principle behind this thesis research: that the environment provided for program execution is a much more significant aspect of a system than the particular technology of its construction.

## 1.3: The Physical Machine

Every virtual machine must be supported by some underlying implementation, and an early hurdle that must be passed by any practical scheme is the demonstration that there exists a viable implementation of it. Details of an implementation of VIM will obviously be influenced by details of the structure of the underlying network. For concreteness, then, we are forced to make choices, some of them fairly arbitrary, about those details. We pick a network of identical processors, each with some amount of private memory attached. Each processor communicates directly with only a limited number of other processors (*e.g.*, four) in the network. There are two reasons for making this choice. First, it lends itself to extremely simple communication hardware, keeping communication costs in proportion to the costs of other parts of the network. Second, it avoids scaling problems. The absence of any central medium means the absence of any opportunity to saturate it, hence a processor array of our design could in principle be iterated to arbitrarily large size.

It can be argued that as our network is expanded, the distance, and hence delay, of nonlocal accesses will increase, defeating the scalability argument. This is true if accesses by a processor are uniformly distributed over the network. However, if tasks exhibit a suitable locality of reference, this need not be so. Expanding the network will always increase its capacity to handle more tasks concurrently, though at some point it will cease to affect the real time required to complete any particular task — at that point, the task will be spread thinly enough that any further spreading would increase communication delays more than warranted by the processing power gained. It is possible, however, to construct networks, such as binary trees, where the power gained is exponential in the increasing distance between the farthest-apart nodes serving a task. (The three-dimensionality of space does put an ultimate limit on the application of this approach, but only at the point where the length of wires connecting nodes becomes significant[37].)

It may be that a central medium is actually more cost-effective, up to some size, than the fully decentralized scheme proposed here. As such a network with a central medium is scaled up, though, a time will come when it is impossible to continue connecting processors to that central medium. At this point, the central medium will have to be split into two "central" media, with some link between them. After further size increases occur, a macroscopic view of the resulting network will look much like our proposal — a collection of tightly coupled processor-memory units (one such unit corresponding to each "central medium") connected so that each unit directly communicates with some number of neighboring units. Thus although our approach may not be the best for some small networks, it deals with the limiting case of any network as it is scaled up.

The ultimate validity of our approach stems from the fact that it supports an appropriate environment for program execution, but its usefulness depends on the practical viability of an implementation. The most basic problem here is the management of data and code objects, to allow nonlocal accesses, motion of objects from one processor to another, and garbage collection across the network. These functions are accomplished using data structures known as *reference trees*[15,16], leading any interconnection of processors using this scheme to be dubbed a *reference tree network*. Chapter 5 describes the use of reference trees to support the various operations required by VIM.

### 1.4: System Performance

Reference trees allow the basic network functions to be performed. However, a network will not operate up to its potential unless good decisions are made about how to allocate tasks and data among processors.

Inefficiencies in the MuNet implementation fall into two categories. One category includes overhead incurred during normal housekeeping chores inside processors, such as free storage management and task enqueuing and dequeuing; the other includes lost time resulting from unfortunate distribution of tasks or data, or from non-optimal reference tree configurations. The former kind of inefficiency might be called "tactical," the latter "strategic." Tactical inefficiencies may be remedied by more careful coding or more appropriate hardware, but do not present research questions along the general thrust of the thesis. For the most part, they relate to activities that would be required on any garbage-collected message-passing system, even a single-processor one. Strategic inefficiencies, on the other hand, are intimately related to strategies of operation of a particular network. Furthermore, strategic inefficiencies can have effects which are orders of magnitude greater than those of tactical inefficiencies, and also quite possibly grow faster than linearly in the size of the network.

For these reasons, we shall be primarily concerned with inefficiencies of the strategic kind. Our principal metric will not be whether the network performs better at some task than another configuration of similar cost could, given comparable efforts devoted to programming the task in suitable ways for each configuration (although this *would* be a reasonable way to judge a commercial product). Rather, we will compare the performance of similar message-passing programs (*i.e.*, subject to the same tactical inefficiencies) and see if employing the network brings an acceptable increase in performance over that afforded by a single processor or smaller network. The scheduling algorithms described in this thesis do exhibit reasonable scaling behavior over different sizes of networks, at least for some applications.

### 1.5: Summary

Present-day machines do not scale well. This is because of the monolithic nature of current processors, and (in the high-performance realm) because of the "von Neumann bottleneck" separating processor from memory. Attempts to "widen" the bottleneck by adding parallel paths within it are unattractive because the complexity of the hardware involved increases more than linearly in the degree of widening achieved. An alternative is to replace a big von Neumann bottleneck with several independent little von Neumann bottlenecks, in the form of small processor-memory pairs connected in a network. This alternative, which is a way of distributing processing power among the memory (or vice versa), has the advantage of being scalable to arbitrarily large sizes. If the network *nodes are sufficiently small* and numerous, the network solution may also offer an answer to our first complaint, about the difficulty of making fine adjustments to the capacity of today's processors. Given the current economics of LSI chip technology, it may also offer a more inexpensive way to deliver computing power.

The utility of such a network will be seriously compromised, however, without an appropriate semantic coherence among the nodes and reasonable transparency of network operations. This thesis reports on the specification of a standard environment for program execution which has these desirable properties, and the design and evaluation of an implementation.

The problem of constructing scalable, high-performance systems is thus attacked along three fronts: the definition of the VIM virtual machine, the development of the reference tree mechanism for object management, and the investigation of scheduling algorithms and collection of performance results. These efforts, although they all come together in the architecture of reference tree networks, are nevertheless to some extent independent. In particular, the VIM virtual machine does not presuppose in any way an implementation using reference trees, and should be suitable in any context where an object-oriented system able to handle concurrency is desired. Conversely, the reference tree scheme could be used to support a wide variety of virtual machines needing its abilities to deal with objects and object references, and to perform garbage collection and synchronization of access to objects. VIM and reference trees are a good match for each other, though, and come together in the evaluation of performance results in Chapter 7.

### 1.6: Thesis Overview

This thesis has eight chapters. Chapter 1 is this introduction. Chapter 2 describes the basic VIM virtual machine and its blackboard interpreter. Chapter 3 outlines some possible extensions to VIM to handle demands that might be made by operating systems running under VIM. Chapter 4 describes the kinds of physical networks on which the proposed implementations of VIM are to be constructed. The "reference tree" mechanism at the heart of these implementations is presented in Chapter 5. Chapter 6 is a discussion of network performance, Chapter 7 deals with strategies for distributing objects and tasks around a network, and Chapter 8 contains concluding thoughts.

# Chapter 2: The VIM Virtual Machine

## 2.1: Introduction to VIM

Programs running under VIM belong to a universe populated with two basic kinds of entities: *objects* and *events*. Data and algorithms are represented as *objects*, which serve functions similar to those of *files* in operating systems, or *cells* in LISP[25]. Computations to be performed are represented as *events*, which correspond very roughly to *processes* in traditional operating systems. The concepts of object and event derive from research into actor systems[3,18,23].

The information contained in an object is recorded in its *text*. A text may be represented as a series of words in memory, together with some format information. The format information indicates the length of the text, and also identifies certain of the words as *references*[5] to other objects (or even to the same object — circular structures are perfectly legitimate). The remaining words need not be intelligible to VIM and may contain arbitrary binary data. The space of objects is assumed to be garbage-collected, so no explicit de-allocation of objects need ever occur. (In fact, VIM provides no facility for explicitly de-allocating objects.)

In the blackboard interpreter, an object is represented as shown in Figure 2.1. Each object is drawn as a series of *slots* which represent the contents of that object's text and bear symbolic names (for convenience in using the blackboard interpreter). The names may be integers, such as **0** and **1**, ordinary identifiers, such as **a** or **b**, or special reserved symbols such as **∅**. Each slot may contain a reference to another object, drawn as an arrow originating in the slot and pointing at the object referenced (as in the slots **∅** and **b** of Figure 2.1). A slot may alternatively contain a special *null reference* drawn as a diagonal line (as in slot **0** of

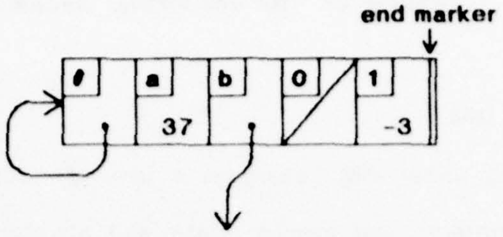


Figure 2.1: Blackboard representation of an object

Figure 2.1), or an arbitrary integer.

Another feature of every object is an *end marker*, drawn as a double vertical line, which indicates the end of the object's text. Only the contents of slots to the left of an object's end marker are accessible to executing programs. (Examples will be given later in which there are slots to the right of the end marker; for now, that possibility may be ignored.) The number of slots from the beginning of an object to its end marker is referred to as the object's *length*.

In form, an event resembles an object text. It too may be represented as a series of words in memory with associated format information, and it too may contain both object references and uninterpreted words. A particular slot within each event, however, is reserved for a reference to its *target object*. Semantically, an event is a request to execute code associated (in a way to be described) with the target object. During execution of that code, the references and words contained in the event will be available as parameters. Upon completion of execution, its mission fulfilled, an event is deactivated (and presumably reclaimed by the storage system). Thus if further computations ought to ensue, execution of an event should result in the creation of new events requesting the performance of those computations. It is possible to "fork off" several consequent events from a single event execution, and it is possible to construct "join" operators that wait for the completion of several events before continuing; indeed, it is expected that the use of such parallelism will help greatly in making the most effective use of a network. Within a single thread of events, however, there is only the simple control structure of event creation. Subroutine calls and returns, loops, and other more complicated control structures may be implemented using *continuations*[18].

An event in the blackboard interpreter is drawn as in Figure 2.2. As in the case of an object, an event is drawn as a series of named slots, each of which may contain an object reference (note that there is no such thing as a reference to an event), a null reference, or an integer. The reserved identifier $\tau$ marks the slot containing the reference to the event's target object. A special triangular area is drawn to the right of the end marker, containing a Boolean *active flag*. The value of this flag is true, or T, if the event should be executed, and otherwise is false, or F. This latter state can occur if the event in question has already been executed, or for other reasons.

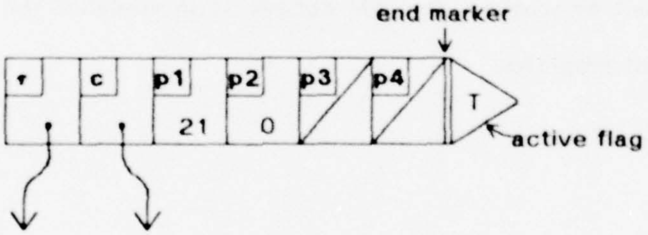


Figure 2.2: Blackboard representation of an event

In the course of executing an event, several special services provided by VIM may be invoked. It is often necessary to examine and/or modify the text of an object, so VIM offers facilities for gaining access to a text in either read-only mode (sharable with other readers) or read/write mode (sharable with no one). Requesting either service entails specifying to VIM a reference to the desired object.

Execution of the requesting event is not allowed to continue until access in the indicated mode is acceptable. Once granted, permission to access a text in a certain mode persists during the remainder of the execution of the event. Thus if an event has acquired non-shared access to an object, no other event execution may access the object while the original event continues to execute. These permissions are not inherited by any subsequent events, however, which must reestablish (via new calls to VIM) any access privileges they desire.*

The VIM style of access to object texts, analogous to locking protocols used in transaction-based database management systems[8,11,32], makes possible the construction of various kinds of synchronization operators without additional special provision within the VIM implementation itself. Operators such as semaphores need to examine, and possibly modify, object texts free from inconsistencies caused by concurrent activities of other processes, and without causing any other process to see any inconsistencies. By controlling the areas of possible interference between concurrent event executions, the VIM access style simplifies the construction and understanding of programs.

*It is important to note that this regulation of "access privileges" is not intended as a protection mechanism. Significant protection may inhere in the fact that it is impossible for an event to access the text of any object to which it cannot obtain a reference, but this protection is insufficient in many cases. Additional protection must be supplied either through higher-level constraints (*e.g.*, forced use of some high-level language compiler) or through augmentations to the facilities provided by VIM. The use of either alternative as a protection mechanism is only tangentially within the scope of this thesis.

## 2.2: Deadlock Avoidance

The VIM scheme for obtaining non-shared access to object texts sounds like a sure recipe for deadlock any time concurrently executing events attempt to gain such access to the same objects. In order to be able to resolve deadlocks, VIM must have the authority to *abort* an event execution any time additional access privileges are requested. (In fact, VIM's authority to abort events is slightly broader than this.) Aborting the execution of an event frees any resources (such as access to object texts) that the event might have accumulated, allowing other events waiting for those resources to proceed. Execution of the aborted event may then be tried again later.

When VIM aborts an event, it should create the appearance that execution of the aborted event was never attempted. If this is done, programmers working under VIM will never have to consider the possible influence of previously aborted executions on the final, successful execution of an event. Therefore, if an aborted event had performed any modifications to object texts, those modifications must be undone. This problem can be avoided by prohibiting an event from making requests for additional resources once it has performed any such modifications. If all events observe this discipline, no event that is aborted will ever have performed any side effects, and events can be aborted correctly without adjusting the contents of any object texts. Accordingly, we modify our previous statement concerning VIM's authority to abort events, stipulating that execution of an event may be aborted *only if it has performed no side effects* (the concept of "side effect" will be made more precise in the definition of the blackboard interpreter). So that an event can be aborted any time it requests additional access privileges, we must require that *such requests be made only when the requesting event has not yet performed any* *side effects.*

Requiring events to adhere to this discipline does not reduce the logical power of VIM; if additional resources are needed after a side effect has been performed, a new event may be created and execution of the current event brought to an end. This new event, when executed, may accumulate such resources held by the old event as are still needed, plus the new resources desired, and then proceed with the computation.

If an event were to acquire some resources, perform a side effect, and then execute forever, no opportunity would arise for aborting it and freeing the resources it had tied up, and hence other events needing those resources would be prevented from ever completing. If execution of every event is guaranteed to take only finite time, this problem can be avoided (actually, only the execution time *after the first side effect* need be finite), hence we require the user of VIM to obey this discipline. The possibility of events being "starved out" from access to needed resources is one reason for insisting that execution time of events not be infinite. There are other, pragmatic reasons for keeping the execution time for any one event fairly short. The longer the execution time for an event, the more resources it is likely to need to acquire. This increases not only the likelihood that the event execution will be aborted before it is able to complete successfully, but also the amount of work that is lost if such an abortion occurs. Thus although VIM is a perfectly suitable environment for running lengthy programs, the execution of such programs should be broken up into modest-length event executions. In the MuNet, there is actually a bound on the amount of time the execution of a single event is allowed to take.

The concept of event abortion, although strictly necessary only for deadlock avoidance, has other uses. In a multiple-processor system, an event might request a resource inconvenient to supply on the processor where the event is currently being executed. For example, an event might request access to an object text not currently present on that processor, or ask to create an inconveniently large object. In such a case, the best solution may well be to transfer the event to a more hospitable location. It will frequently be more economical to abort the event execution and start over from scratch in the new location than to attempt to transfer all the intermediate state information pertaining to the event execution. Therefore, under VIM, we include requests to allocate storage (create events, create or expand objects) along with requests to gain access to object texts in the class of operations that are only legal when the requesting event has not yet performed any side effects. Thus an event execution may be aborted during any such request.

The original motivation for aborting events was to allow resolution of deadlocks caused by requests for non-shared access to an object; the concept was then found useful for the various functions described in the preceding paragraph. If only shared (read-only) access to objects is allowed, then it will never be necessary to abort an event. But if the ability to obtain read/write access to objects is removed from VIM, and no new facility put in its place, it becomes impossible to perform "joins" or any kind of synchronization between parallel processes. There do exist mechanisms less general than the unlimited ability to perform side effects, such as *conduits*[38] or *tokens*[15], that might be used to solve this problem without introducing the need to be able to abort events. VIM's goal of being able to represent most general-purpose computations, however, is most easily served by the unencumbered side-effect mechanism proposed here.

### 2.3: Object Types

When an event is to be executed, the machine code to actually be interpreted by VIM is determined by the identity of the event's target object. Any object used as a target object must contain, within its text, an object reference designated as that target object's *type*. In the blackboard interpreter, the slot containing this reference is designated by the reserved symbol ø. The text of the type object (whose reference is found in this slot) is interpreted as containing the machine code to be executed. This relationship is depicted in Figure 2.3. Execution of an event begins by locating the text of the target object's type object, and transferring control to the machine code found therein. This machine code may contain ordinary arithmetic and logical manipulations, along with requests for special VIM services such as those discussed above.

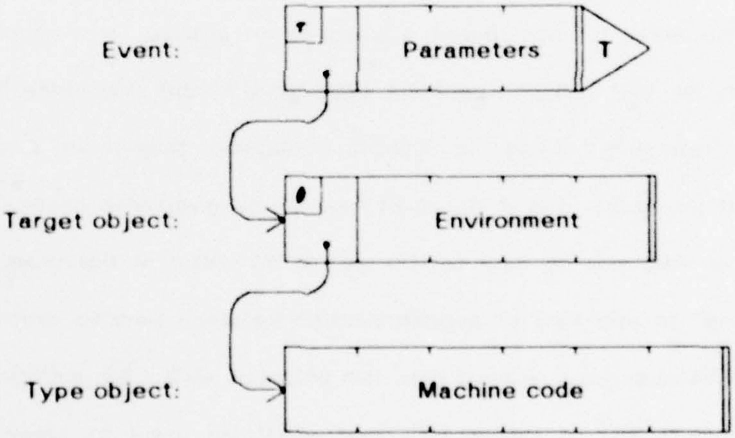


Figure 2.3: Object types

The reason for adding the extra level of indirection of the type mechanism, rather than simply finding the machine code to be executed in the text of the target object, is that it permits easy creation and manipulation of closure-like objects.

The text of the target object contains the execution environment (bindings of variables within the closure), and the text of the type object contains the algorithm. The type mechanism thus eliminates the need to copy the algorithm into every closure. In terms popularized by Hewitt, the type object contains the *script* and the target object the *acquaintances*[18].

### 2.4: System Services

For concreteness, this section enumerates the fundamental VIM system services. These are not "operating system" services, merely the most basic facilities provided by the virtual machine itself.

**gtext**(*ref*);

Obtains access, for the currently executing event, to the text of the object referenced by *ref*, in read-only mode. For the remaining duration of the event execution, *ref* may be used in specifying words or references to be read out from the text. A **gtext** request may not occur after a side effect has been performed.

**locktext**(*ref*);

Like **gtext** in every respect, except that non-shared (read/write) access to the object is obtained.

**newobj**(*size*) **returns** *ref*;

Creates a new object with a text of the specified size. A reference to the new object is returned in *ref*. Read/write access to the object is allowed during the remainder of the current event execution. A **newobj** call *can* abort the requesting event (this allows a VIM implementation to move the event to where more space is available), and hence may not be made after a side effect has been performed.

**newev**(*size*) **returns** *evpointer*;

Creates a new event of the size requested, and returns a pointer (which may be used for filling in the event) in *evpointer*. An event created by **newev** will not be activated if execution of the creating event does not complete successfully. Also as with **newobj**, a **newev** request may not occur following a side effect.

**done**();

Indicates that execution of the current event has completed. All objects and events created by it are officially installed, and the current event is deactivated.

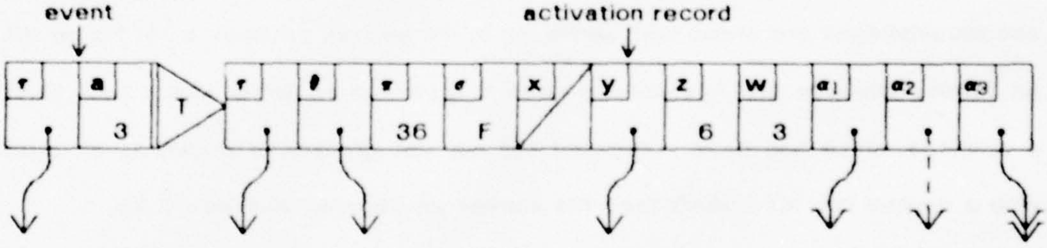
This set of primitives provides only a very basic and unoptimized interface to VIM. Calls providing additional capabilities are discussed in subsequent sections; modifications to VIM to improve the efficiency of possible implementations are touched upon in Appendix A.

### 2.5: The Blackboard Interpreter

The blackboard interpreter for VIM has two components: a graphical representation for the state of a computation, and a set of transition rules specifying what alterations to a computation state will yield a legal successor of that state. By starting with some initial state, and repeatedly applying the transition rules, one can generate a sequence of states that corresponds to a legal computation in VIM. Due to the fact that several events can simultaneously be active, as well as the fact that it is possible to write nondeterminate programs in VIM, a state will in general have several legal successor states, depending on the focus of the transition rule applied. Indeed, this attribute of VIM is necessary i' we are to imagine VIM programs executing on multiple processors. Not all *legal* successors of a state are equally *desirable*, though, since some involve regression rather than progression of the computation (aborting an event is an example of a regressive step). Some choices of successors may result in parts of a computation being "starved out" or ignored to the benefit of other parts of the computation. Avoidance of these problems is generally associated with "fairness" of scheduling, and is not dealt with by the rules of the blackboard interpreter. Thus the VIM semantics defined by the blackboard interpreter cannot guarantee *termination* of a computation. They can, however, assure *partial correctness*, in that no rule allows a transition which violates any of the guarantees which VIM makes with respect to the meanings of the various permissible operations.

### 2.5.1: Definition of the Blackboard Interpreter

The graphical representation of objects and events has already been given (in Figures 2.1 and 2.2, above). One additional entity appears in blackboard interpretations of VIM: this is the *activation record* (see Figure 2.4), one of which is associated with each event currently being executed. An activation record for an event appears when the event begins execution; an event's activation record disappears when it either successfully completes execution or is aborted. Every activation record is drawn just to the right of the event it corresponds to (an event may have at most one activation record at one time). Like an object or event, an activation record has several slots, each bearing a symbolic identifier and containing either an object reference or some kind of primitive value. Slots may be added to an activation record as execution of its associated event proceeds.



Figu're 2.4: An activation record (with associated event)

As in the case of an object or event, special slots in an activation record may be labeled with various reserved identifiers. In fact, every activation record has several such slots. The reserved identifiers used in activation records are given in

| *Identifier* | *Slot Contents* |
|---|---|
| $\tau$ | a reference to the event's target object |
| $\theta$ | a reference to the event's type object |
| $\pi$ | the current program counter for the event |
| $\sigma$ | a Boolean side effect flag |
| $\alpha_i$ | an access privilege |

Table 2.5: Reserved identifiers used in activation records

Table 2.5. Several new terms are found in the table. The *program counter* $\pi$ names the slot in the event's type object containing the next machine instruction to be executed. The *side effect flag* $\sigma$ is a Boolean which is initially false, but is set to true when the event performs its first side effect. In fact, we will *define* a "side effect" as any action which causes the side effect flag to be set to true; the specific rules governing the setting of the side effect flag will be given below. The slots $\alpha_i$ contain the *access privileges* obtained thus far by the current event (*e.g.*, via **gtext** or **locktext**). The graphical representation of an access privilege bears a superficial resemblance to that of an object reference — both are drawn as arrows. However, to denote their special status, the arrowheads associated with access privileges are drawn with serifs, as in the access privilege $\alpha_1$ in Figure 2.4. An access privilege that has been granted is drawn as an arrow with a solid shaft; a privilege which has been requested but not yet granted is drawn as an arrow with a dashed line for a shaft (see the access privilege $\alpha_2$ in Figure 2.4).

To distinguish between different kinds of access privileges, arrows with multiple heads are used. Table 2.6 summarizes the different kinds of access privileges and the monitor calls from which each may result.

Access privileges cannot be explicitly handled by programs in VIM. In particular, they cannot be copied out of the slots $\alpha_i$ into any other slots in an activation

| *Access Privilege* | *Meaning* |
|---|---|
| - - -> | **read-only access to an object (gtext)** |
| ——>> | **read/write access to an object (locktext)** |
| ——>>> | **newly created object (newobj)** |
| ——>>>> | **newly created event (newev)** |
| - - ->> | **gtext**-to-**locktext** conversion request |

Table 2.6: Access privilege types

record, nor can they be saved in event or object slots. Access privileges exist only to record permissions accumulated by an executing event, and when an event's activation record disappears, the associated access privileges must disappear also.

Before giving the state transition rules for the blackboard interpreter, it is convenient to define some "canned procedures" as follows:

ERROR:

A call to this routine indicates that an erroneous condition exists (*i.e.*, the user has violated some rule of programming in VIM). The response of VIM to such error conditions is not specified by the blackboard interpreter.

HAS-GTEXT(*event*,*object*):

If some slot $\alpha_i$ in the activation record for *event* has an access privilege (of any type) for *object*, then return true; else return false.

HAS-LOCKTEXT(*event*,*object*):

If some slot $\alpha_i$ in the activation record for *event* has a **locktext**- or **newobj**-type access privilege (serifed double or triple arrowhead) for *object*, then return true; else return false.

GTEXT(*event*,*object*):

If HAS-GTEXT(*event*,*object*) then return; else create a new slot $\alpha_i$ in the activation record for *event* and place in it a **gtext**-type access request (an arrow with a dashed shaft and a serifed single arrowhead) for *object*.

LOCKTEXT(*event*,*object*):

If HAS-LOCKTEXT(*event*,*object*) then return; else if some slot $\alpha_i$ in the activation record for *event* has a **gtext**-type access privilege (serifed single arrowhead) for *object*, convert that access privilege to a **gtext**-to-**locktext** conversion request (an arrow with a dashed shaft and a double arrowhead with bent serifs) for *object*; else create a new slot $\alpha_i$ in the activation record and place in it a **locktext**-type access request (an arrow with a dashed shaft and a double arrowhead with straight serifs) for *object*.

CREATE-ACTIVATION-RECORD(*event*):

Draw a blank activation record to the right of *event*. If *event* has no slot labeled $\tau$, or if the contents of slot $\tau$ is not a reference to an object, then ERROR; else copy the object reference from slot $\tau$ of *event* to slot $\tau$ of the activation record. Set slot $\theta$ of the activation record to the null reference. Set slot $\pi$ to zero and slot $\sigma$ to false. Perform GTEXT(*event*, object referenced by slot $\tau$ in the activation record).

ERASE-ACTIVATION-RECORD(*event*):

Erase the activation record to the right of *event*. Also erase all object reference and access privilege arrows (whether their shafts are solid or dashed lines) emanating from the erased activation record.

CAN-RUN(*event*):

If no access requests are outstanding in the activation record for *event* (*i.e.*, no arrows with dashed shafts emanate from any slot $\alpha_i$ in the activation record), then return true; else return false.

SIDE-EFFECT-PERFORMED(*event*):

If slot $\sigma$ of the activation record for *event* contains the value true, then return true; else return false.

The following transition rules can each be applied to any event e meeting the specified conditions:

**Rule R1:** If *e* has no activation record and the active **flag** of e is true, then CREATE-ACTIVATION-RECORD(*e*). This corresponds to beginning execution of the event e and **gtexting** the event's target object.

**Rule R2:** If *e* has an activation record and SIDE-EFFECT-PERFORMED(e) is false, then ERASE-ACTIVATION-RECORD(*e*). This corresponds to aborting execution of *e*.

**Rule R3:** If *e* has an activation record, CAN-RUN(*e*) is true, and slot $\theta$ of the activation record contains a null reference, then if the object referenced by slot $\tau$ in the activation record has no slot $\theta$ then ERROR; else if that slot $\theta$ does not contain a reference to an object then ERROR; else copy that reference into slot $\theta$ of the activation record for e, and perform GTEXT(e, object referenced by slot $\theta$ in the activation record for *e*). This corresponds to **gtext**ing the event e's type object.

**Rule R4:** If *e* has an activation record, CAN-RUN(*e*) is true, and slot $\theta$ of the activation record does not contain a null reference, then NEXT-INSTRUCTION(*e*). The routine NEXT-INSTRUCTION is described later in this section, and has the effect of executing the next instruction in the type object of the event *e*.

The following transition rules can be applied to any object *o* meeting the specified conditions:

**Rule R5:** If *o* has a **gtext**-type access request (dashed arrow with serifed single arrowhead) impinging on it, and no **locktext**-type access privileges (solid arrows with serifed double arrowheads) impinging, then convert the **gtext**-type request into a **gtext**-type access privilege (solid arrow with serifed single arrowhead). This corresponds to granting shared (read-only) access to an object that is not currently being held by anyone for non-shared (read/write) access.

**Rule R6:** If *o* has a **locktext**-type access request (dashed arrow with straight-serifed double arrowhead) impinging on it, and no access privileges of any sort (solid arrows with any number of serifed arrowheads) or **gtext-to-locktext** conversion requests (dashed arrows with bent-serifed double arrowheads) impinging, then convert the **locktext**-type request into a **locktext**-type access privilege (solid arrow with serifed double arrowhead). This corresponds to granting non-shared access for an object not currently being held by anyone for any kind of access.

**Rule R7:** If *o* has a **gtext-to-locktext** conversion request (dashed arrow with bent-serifed double arrowhead) impinging on it, and no other **gtext-to-locktext** conversion requests, or access privileges of any sort, impinging, then convert the **gtext-to-locktext** conversion request into a **locktext**-type access privilege (solid arrow with serifed double arrowhead). This corresponds to upgrading an event's shared access to an object to non-shared status.

This completes the list of transition rules for the basic VIM blackboard interpreter, except that the routine NEXT-INSTRUCTION, which is referenced in the definition of Rule R4 and describes the interpretation of machine code found in type objects, has not yet been described. The purpose of postponing this until last has been to abstract from the presentation the details of any particular instruction set for VIM. The goals of this thesis are little advanced by hypothesizing in detail, for example, the permissible set of addressing modes for an add instruction. Consequently, such nuances are avoided in the definition of NEXT-INSTRUCTION. Instead, the body of the definition has two main parts: first, a section dealing with the VIM system calls, whose format may be considered somewhat standardized, and second, a section describing the treatment of different kinds of accesses to objects and events, without detailing how these basic kinds of accesses are packaged into instructions. The definition of NEXT-INSTRUCTION follows.

NEXT-INSTRUCTION(*event*):

Fetch the contents of the slot in *event*'s type object (referenced by slot $\theta$ in the activation record of *event*) designated by the contents of slot $\pi$ (the program counter) in the activation record for *event*. Interpret the datum thus fetched as an instruction. If the instruction is a VIM system call, then it is one of the following:

**done**(): For every $\alpha_i$ in the activation record for *event* that contains a **newev**-type access privilege (arrow with serifed quadruple arrowhead), set to true the active flag of the event pointed to by the access privilege. Set to false the active flag of *event*. ERASE-ACTIVATION-RECORD(*event*).

**gtext**(*object*):* If SIDE-EFFECT-PERFORMED(*event*) then ERROR; else advance $\pi$ to the next instruction and perform GTEXT(*event*,*object*).

**locktext**(*object*): If SIDE-EFFECT-PERFORMED(*event*) then ERROR; else advance $\pi$ to the next instruction and perform LOCKTEXT(*event*,*object*).

*Although we have not specified the means by which the reference *object* is obtained, in this and all subsequent cases it is assumed that *object* came either from *event*, from the activation record for *event*, or from the text of some object for which at least **gtext**-type access has already been established by *event*. The intent of this stipulation is to ensure that all references used by an event are accessible from it via some chain of object references.

**newobj**(*$id_1$*,*$val_1$*,*$id_2$*,*$val_2$*,...,*$id_n$*,*$val_n$*): If SIDE-EFFECT-PERFORMED(*event*) then ERROR; else advance $\pi$ to the next instruction, draw a new object with *n* slots labeled $id_1,id_2,...,id_n$, with initial contents $val_1,val_2,...,val_n$, create a new slot $\alpha_i$ in the activation record for *event*, and draw a **newobj**-type access privilege (solid arrow with serifed triple arrowhead) from this slot to the newly created object. The **newobj** operation returns a reference to the newly created object.

**newev**(*$id_1$*,*$val_1$*,*$id_2$*,*$val_2$*,...,*$id_n$*,*$val_n$*): If SIDE-EFFECT-PERFORMED(*event*) then ERROR; else advance $\pi$ to the next instruction, draw a new event with *n* slots labeled $id_1,id_2,...,id_n$, with initial contents $val_1,val_2,...,val_n$, create a new slot $\alpha_i$ in the activation record for *event*, and draw a **newev**-type access privilege (solid arrow with serifed quadruple arrowhead) from this slot to the newly created event, and set the active flag of the newly created event to false. The **newev** operation returns an integer which may be used to refer to the new event.

If the instruction is not a VIM system call, then it may attempt some combination of the following operations:

read from activation record for *event*: if reading from one of the slots $\alpha_i$, then ERROR; else proceed.

write into activation record for *event*: if writing slot $\sigma$ or any of the slots $\alpha_i$, then ERROR; else proceed.

read from *event*: proceed.

write into *event*: ERROR.

read from an event other than *event*: ERROR.

write into an event other than *event*: if a **newev**-type access privilege for the destination event exists in some slot $\alpha_i$ in the activation record for *event*, then proceed; else ERROR.

read from text of *object*: if HAS-GTEXT(*event*,*object*) then proceed; else ERROR.

write into text of *object*: if a **newobj**-type access privilege for *object* exists in some slot $\alpha_i$ in the activation record for *event*, then proceed; else if a **locktext**-type access privilege for *object* exists in some slot $\alpha_i$ in that activation record, set slot $\sigma$ in the activation record to true, then proceed; else ERROR.

**One additional important requirement of the blackboard interpreter has not yet been stated in this section: this is the requirement that the execution of any event take only a finite number of steps after the first step that causes a side**

effect (*i.e.*, causes slot $\sigma$ of the activation record to be set to true). This requirement may be imposed on the user in one of two ways. It may simply be stated that any program which might execute forever after performing its first side effect is illegal, without any indication of how such an illegal program could be detected. This would impose on the user the onus of ensuring the finiteness of every event execution once it has performed a side effect. Alternatively, some number of steps, such as 10,000, could be picked arbitrarily as the maximum number of steps permissible after performing a side effect. Any event execution exceeding this limit could be terminated with an error condition, much as it would be if it attempted to obtain additional access privileges after performing a side effect.

### 2.5.2: Discussion of the Blackboard Interpreter

Before discussing the interaction of the various parts of the blackboard interpreter, it is best to illustrate the interpreter's operation by means of an example. In order to give an example, it is necessary to settle, at least informally, on some sample "machine language" with which to populate the slots of type objects. At the foundation of this language must be some notation for operand locations — we must be able to name the slots from which operands are to be fetched or into which results are to be put. For this purpose we settle on the following convention: a lone identifier $x$ denotes the slot (or its contents, depending on whether the identifier appears as a destination or as a source) bearing the label $x$ in the currently executing event's activation record. A pair $x{:}y$ denotes the slot labeled $y$ in the object referenced by the contents of slot $x$ in the activation record. A pair $\epsilon{:}y$ denotes the contents of slot $y$ in the currently executing event.

Slots are stored into by using the form *locn*←*value*, where *value* may denote the contents of some slot, as discussed above, or may be the result returned by some system call such as **newobj**. *locn* will denote the slot into which the value should be stored. If *locn* is a lone identifier, then a new slot by that name will be added to the current activation record, if no such slot existed before; otherwise, the slot named by *locn* must already exist.

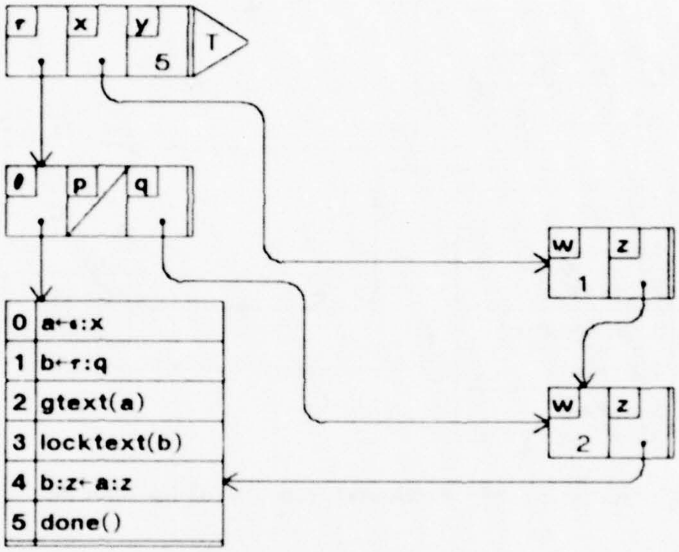
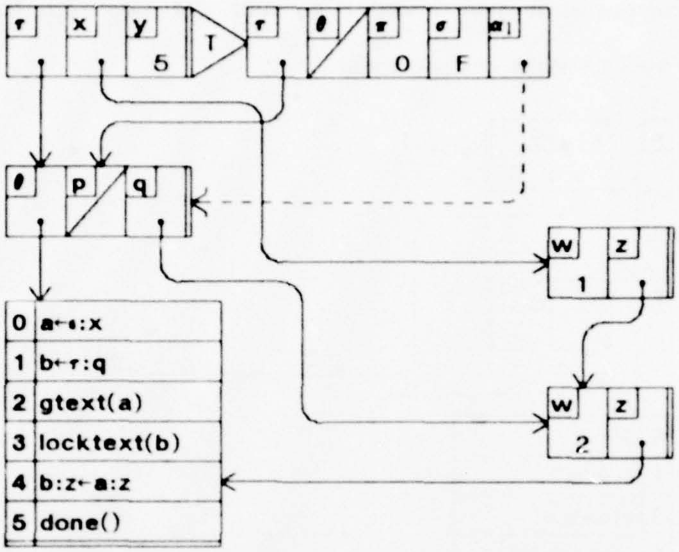


Figure 2.7(a): Initial configuration.

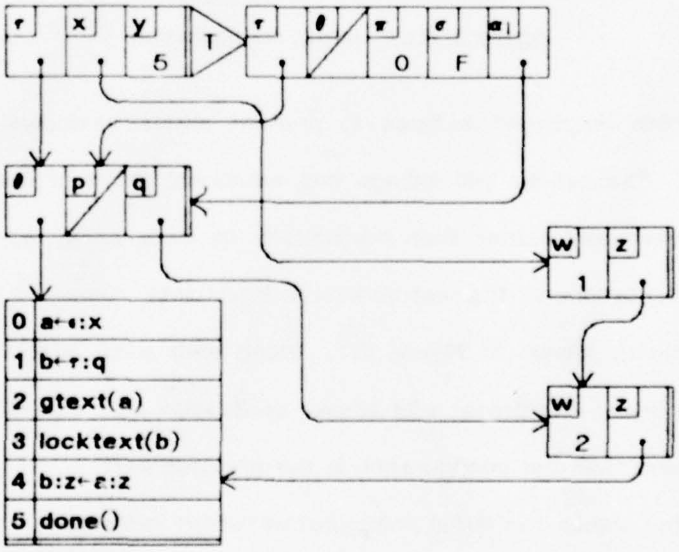
This "machine language" suffices to present simple examples of blackboard interpretations. The reader will indulge one additional piece of novelty — drawing type objects vertically rather than horizontally to more easily accommodate the symbolic representations of the instructions in their slots. Consider then the blackboard interpretation shown in Figure 2.7. Along with each part of the figure is listed the blackboard interpreter rule whose application produces the configuration shown in that part from the configuration in the previous part.

Figure 2.7(a) shows the initial configuration, which includes one event (with its active flag set to true) and a collection of objects. The event's type object, drawn vertically, contains the program to be executed. The slots in this object are labeled with numbers, corresponding to values that will be taken on by the program counter during execution.



```
0 a←ε:x
1 b←τ:q
2 gtext(a)
3 locktext(b)
4 b:z←a:z
5 done()
```

Figure 2.7(b): Produced from 2.7(a) by Rule R1.



```
0 a←ε:x
1 b←τ:q
2 gtext(a)
3 locktext(b)
4 b:z←a:z
5 done()
```

Figure 2.7(c): Produced from 2.7(b) by Rule R5.

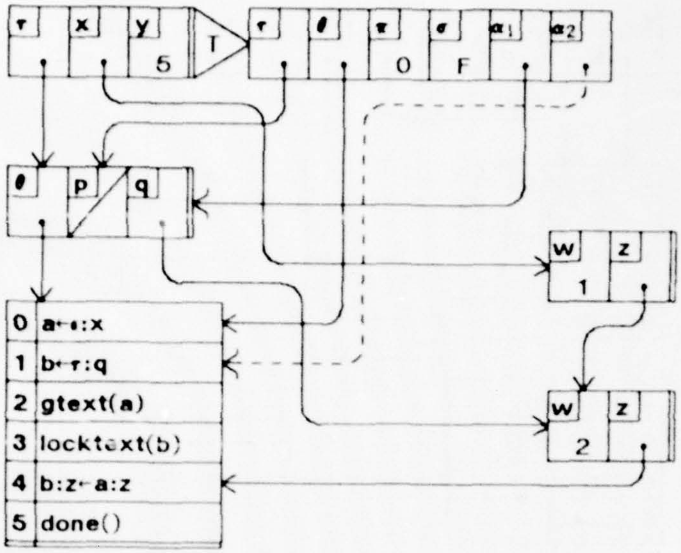


Figure 2.7(d): Produced from 2.7(c) by Rule R3.

Figures 2.7(b) through (e) show the standard start-up sequence for events; note that no step in this sequence depends on the program contained in the type object. This event start-up phase thus corresponds to the next-instruction-fetch phase typical of standard von Neumann machines, which is the same, up to some point, no matter what instruction is being fetched (since the nature of that instruction is not known yet!). In our blackboard interpreter, this "fetch" phase collects references and read-only access privileges for the target and type objects of the event.

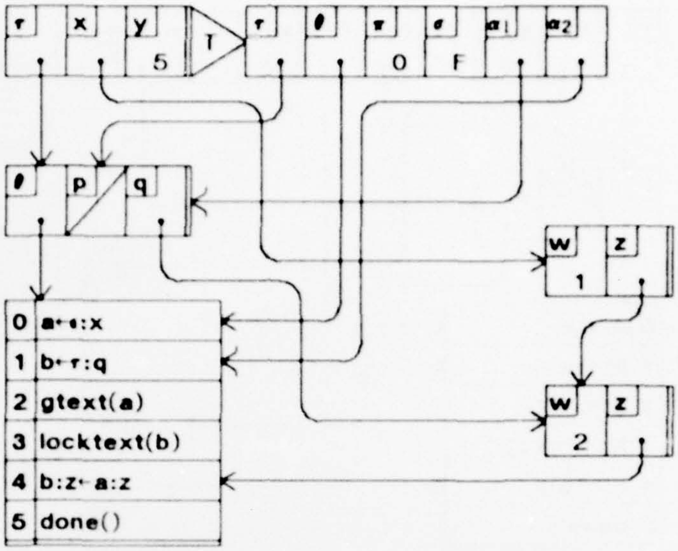


Figure 2.7(e): Produced from 2.7(d) by Rule R5.

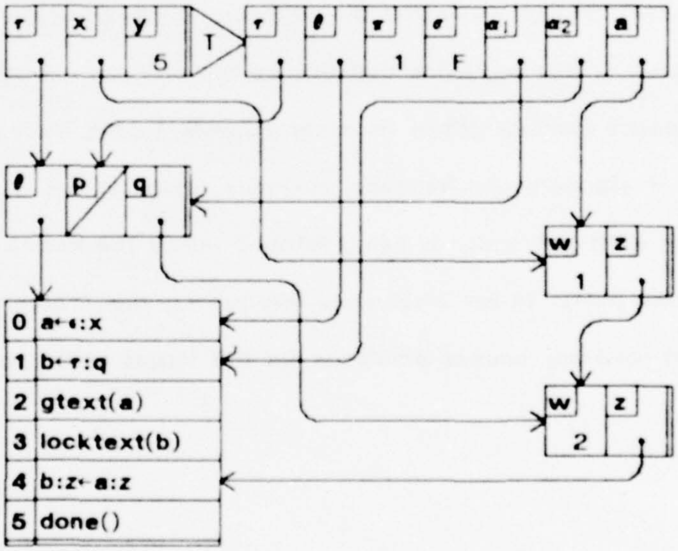


Figure 2.7(f): Produced from 2.7(e) by Rule R4.

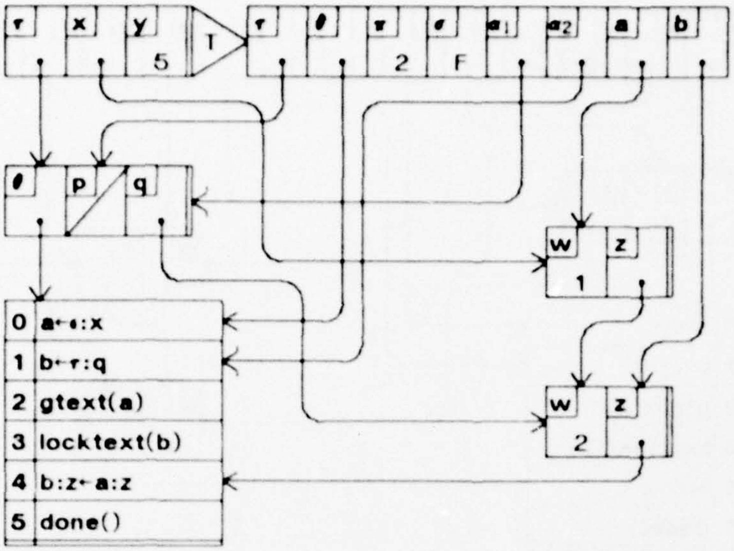


Figure 2.7(g): Produced from 2.7(f) by Rule R4.

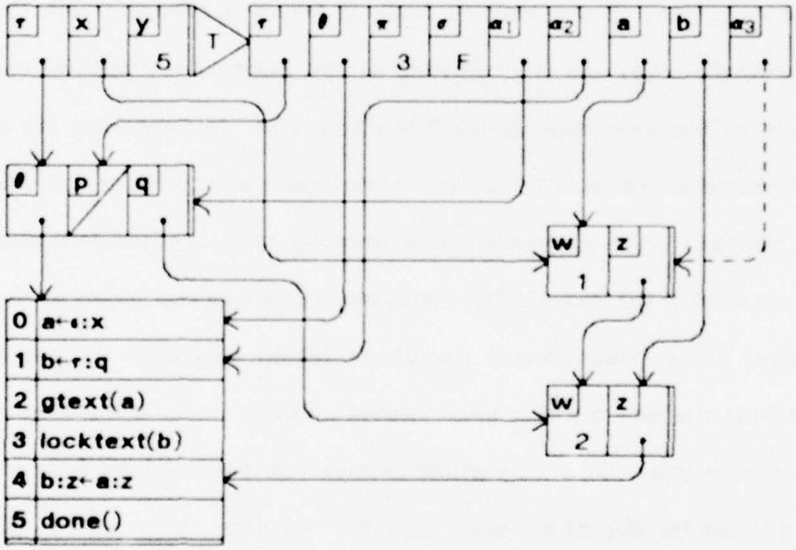


Figure 2.7(h): Produced from 2.7(g) by Rule R4.

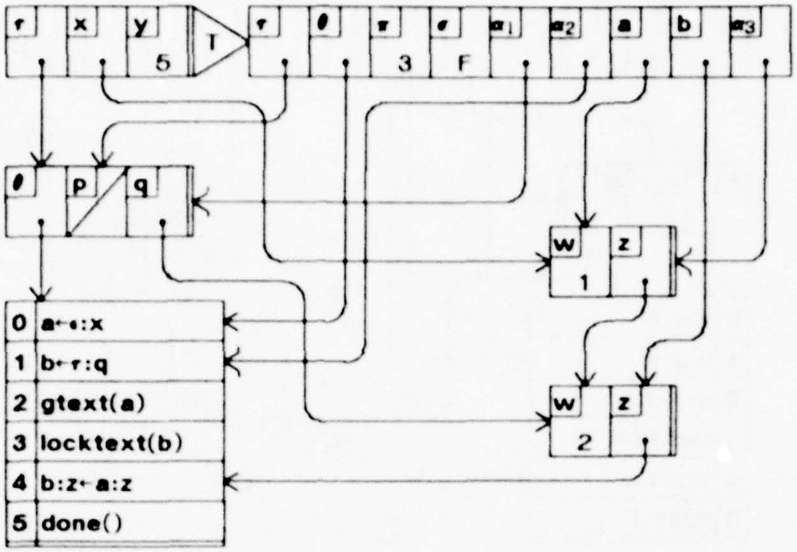


Figure 2.7(i): Produced from 2.7(h) by Rule R5.

Actual execution of the machine code in Figure 2.7 begins with the transition between snapshots (e) and (f). Starting at this point, every application of Rule R4 corresponds to the execution of another instruction. In snapshots (f) and (g), the activation record is extended with new slots **a** and **b**, and initial contents for those slots are supplied. The requesting and granting of **gtext** and **locktext** privileges occupies snapshots (h) through (k). Snapshot (l) shows the result of the first (and, in this case, only) "side effect" performed during execution of the event. The reference from the slot **a:z** has been copied into slot **b:z** and the side effect flag ε has been set to true. Note that **gtext** access has been obtained for object **a** and **locktext** access for object **b**; otherwise the operation "**b:z ← a:z**" would be illegal. Finally, snapshot (m) shows the result of executing the **done** system call — the activation record and all arrows emanating from it have been erased, and the active flag of the event has been set to false, ensuring that its execution will not be attempted again.

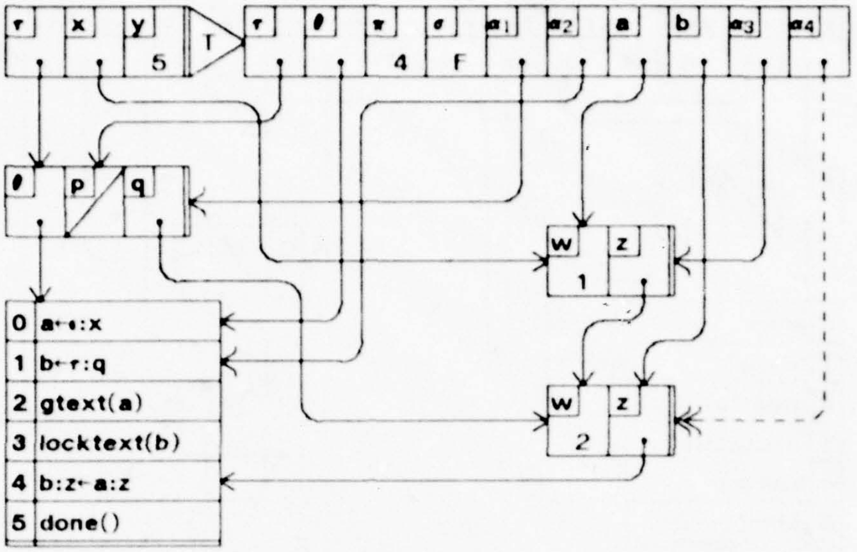


Figure 2.7(j): Produced from 2.7(i) by Rule R4.

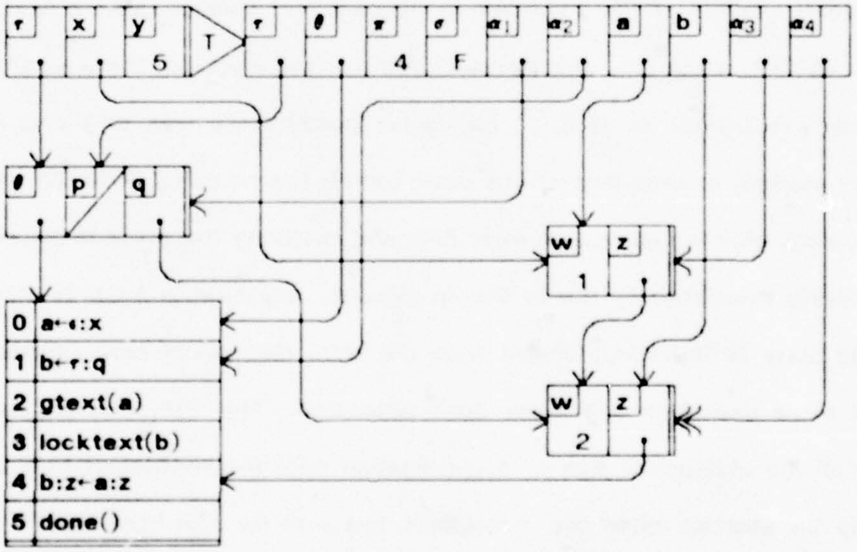


Figure 2.7(k): Produced from 2.7(j) by Rule R6.

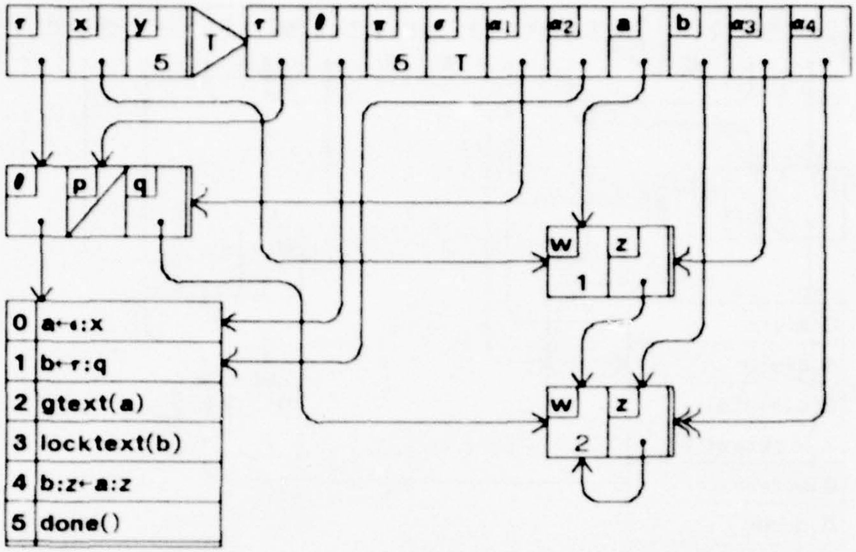


Figure 2.7(l): Produced from 2.7(k) by Rule R4.

Several aspects of the sequence in Figure 2.7(a) through (m) are worth noting. The first is that, since only one event execution was involved, there was no opportunity for parallelism. In fact, at any given point, there was only one rule that could be applied, except that at any point before Figure 2.7(l), Rule R2 could have been applied, aborting the event execution and restoring the state in Figure 2.7(a). The property illustrated by this is true in general: *any time an event is aborted, the resulting state is indistinguishable from the state that would have resulted if the aborted event execution had never been attempted.* The VIM rules regulating the setting of the side-effect flag σ, in combination with the restriction that an event can only be aborted when the side-effect flag σ in its activation record is false, ensure that this is the case.

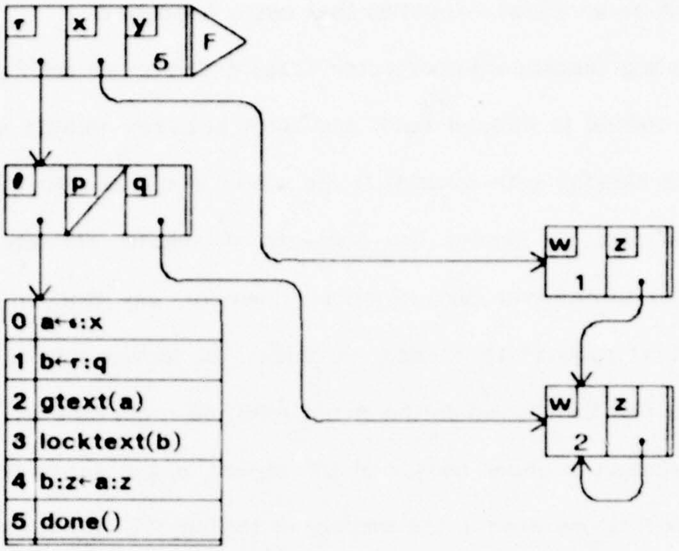


Figure 2.7(m): Produced from 2.7(l) by Rule R4.

More specifically, we can define an *accessible event* as being any active event, and an *accessible object* as being any object reachable by some chain of object references from some ac'ive event or its activation record. When an event execution is aborted, it may have created some events or objects, which will be left around afterwards, but these events and objects will always be inaccessible. The newly created events will be inaccessible because their active flags will still be set to false. Every newly created object will be inaccessible because, in order for it to become accessible, a reference to it would have to have been stored into the text of some previously accessible object. Any event performing such an operation, however, would have its side-effect flag ε set to true, rendering subsequent abortion of the event impossible. Thus if we consider only the states of accessible events and objects to be relevant (a reasonable assumption, since the others can never influence any future computation), an aborted event execution leaves behind the same result as an event execution that never happened.

An interesting blackboard-interpreter feature illustrated by Figure 2.7 is the way in which control is passed back and forth between events and objects. In general, *control resides with events*, in the sense that transition rules are applied at events, and primarily involve the contents of events, activation records, and their associated target and type objects. However, any time an event makes a **gtext** or **locktext** request to access an object, no further rules (other than Rule R2, aborting it) can be applied to the event until the requested access is granted. Granting this access is under control of the object, in the sense that the relevant rule (R5, R6, or R7) operates in the vicinity of the object, taking into consideration only the other access privileges already granted for that object. The form of arbitration among access requests specified by these rules is what guarantees that **gtext** access will only be shared with other readers, and that **locktext** access will be shared by no one.

The nature of these arbitration rules in turn guarantees a second fundamental property of VIM. If we define an *execution history* as a series of snapshots, such as those in Figure 2.7, then *for any legal VIM execution history there exists a history starting with the same initial state and ending with the same final state, which has the property that at most one activation record exists in any one snapshot*. In other words, any arbitrarily parallel execution of some set of events cannot yield a result that could not also be yielded by some sequential execution in which no two event executions are ever interleaved. This property of VIM simplifies the programmer's job, for he can treat each event execu*ion as though it were an atomic operation which cannot be interrupted by any complete or partial execution of another event.

Every VIM execution history can be viewed as a *sequence of primitive* operations, corresponding to the successive applications of the blackboard-interpreter rules to various events and objects. Each application of a rule can be associated with the execution of a particular event. In the case of a rule that applies directly to an event, this event is the event associated with the primitive operations performed. In the case of a rule that is applied at an object, to grant an access privilege, the event to which access is granted is the event associated with the primitive operation.

In a VIM execution history in which events are executed in sequence, the primitive operations corresponding to each event execution will be grouped together, with no interleaving of primitive operations belonging to different event executions. In a general VIM execution history, this will not be the case; however, *any legal VIM execution history can be transformed into one in which primitive operations pertaining to each event execution are grouped together*, without altering the initial or final state. The transformation is simple to perform: without any change in the ordering, relative to each other, of the primitive operations pertaining to each event execution, a new history is built out of these sets of primitive operations, in which the sets appear in the order in which the corresponding event executions in the old history ended (*i.e.*, either completed successfully or were aborted). In other words, all primitive operations in the original history are delayed as long as possible, until they run into the primitive operation representing the termination of the corresponding event execution (this operation should not be delayed). This strategy will produce the appearance that all operations associated with an event execution happen in a burst, immediately before the termination of that event execution.

Obviously this transformation can be made on any execution history, but the contention that the transformation is legal, *i.e.*, does not affect the state change performed by the execution history, needs to be justified. A sufficient condition for the transformation being valid is the following: *given an event execution* **B** *that terminates after another event execution* **A**, *and a primitive operation pertaining to* **B** *that occurs immediately before a primitive operation pertaining to* **A**, *the order of the primitive operations may be switched without altering the state change performed by the pair of primitive operations.* This property, if true, can be used to justify "percolating" operations pertaining to event executions with earlier termination times forward and those corresponding to later termination times back, until no operation appears before another one whose event execution ended later — precisely the situation we are trying to achieve.

In trying to establish this property, we must concentrate on the ways in which event executions can interact. All such ways involve reading or writing of object texts, or manipulation of access privileges to objects. Operations involving an event's activation record are of no interest, since the activation record can only be accessed by the event it belongs to. Similarly, operations that involve reading or writing events are of no interest: an event can only be written by its creator, during which time no other event execution can access it, and it can only be read during its own execution, which cannot overlap with its creation. Thus only manipulations involving objects can lead to interference between event executions.

The primitive operations that affect objects can be grouped into five classes: reading from an object, writing to an object, requesting access to an object, being granted access to an object (this includes **gtext** access, **locktext** access, and object creation which entails a grant of **newobj**-type access to the newly created object), and relinquishing access to an object (this only happens when an event execution ends). The only way events can conflict is on operations involving the same object; clearly, a pair of primitive operations on different objects may be performed in either order. Table 2.8 shows the status of the various combinations of possibly conflicting operations on the same object.

| | A read | A write | A request | A grant | A relinquish |
|---|---|---|---|---|---|
| **B** read | OK(1) | NP(2) | OK(4) | GT(6) | OK(5) |
| **B** write | NP(2) | NP(2) | OK(4) | NP(2) | OK(5) |
| **B** request | OK(4) | OK(4) | OK(4) | OK(4) | OK(4) |
| **B** grant | GT(6) | NP(2) | OK(4) | GT(6) | OK(5) |
| **B** relinquish | NP(3) | NP(3) | NP(3) | NP(3) | NP(3) |

*Event execution* **B** *is assumed to have ended after event execution* **A**; *each entry in the table applies to a pair of primitive operations on the same object in which the operation pertaining to* **B** *occurs immediately before that pertaining to* **A**. *A key to the table entries is*

NP: *this sequence of operations is not possible in a legal VIM execution history.*

OK: *this pair of operations performed in the reverse order will still cause the same state change.*

GT: *if this pair of operations is part of a legal VIM execution history, then all accesses granted must be of* **gtext** *access; in this case, performing the operations in the reverse order will still cause the same state change.*

*Parenthesized numbers following the table entries refer to more detailed explanations in the text.*

Table 2.8: Legality of primitive operation sequence changes

Explanations associated with parenthetical numbers in the table entries are as follows:

(1) Certainly interchanging the order of two reads will not change the data read by either.

(2) If either **A** or **B** has access to write the object, the other cannot possibly have or be granted access to read or write it until the first access privilege is relinquished.

(3) Since access privileges are only relinquished when an event execution ends, and event execution **A** is assumed to have completed before the end of event execution **B**, **B** cannot possibly relinquish any access privileges prior to any operation pertaining to **A**.

(4) The existence, or absence, or *requests* (as opposed to granted privileges) from an event does not interfere with any operation performed by another event. Thus the ordering of a request by one event and any operation by another event may be freely interchanged.

(5) If a read, write, or grant pertaining to **B** were legal *before* **A**'s relinquishment of access privileges, it will certainly be legal after.

(6) Both **A** and **B** have access to the object by the end of the pair of operations. The only way this can happen is if both have **gtext** access; otherwise, sharing would be prohibited. If the access grants are all of **gtext** access, then the order of the operations is irrelevant.

In summary, then, a VIM execution history may be viewed as a sequence of primitive operations. Any legal VIM execution history can be transformed into an equivalent one (performing the same state change) in which event executions happen in sequence, in the order in which the event executions terminate in the original execution history. This transformation can be performed incrementally by a "bubble sort" technique, interchanging the order of pairs of adjacent primitive operations where their order is the opposite of that desired in the final, sequential execution history. The considerations presented in Table 2.8 show that, if the original execution history is a legal one, such order changes will always be legal, even when they apply to pairs of operations on the same object.

In the scenario of Figure 2.7, there was always one transformation, other than aborting execution of the event, that could be applied. In the presence of several events, there might at any point be several possible transformations: on the other hand, there might be none, as illustrated in the snapshot of Figure 2.9.

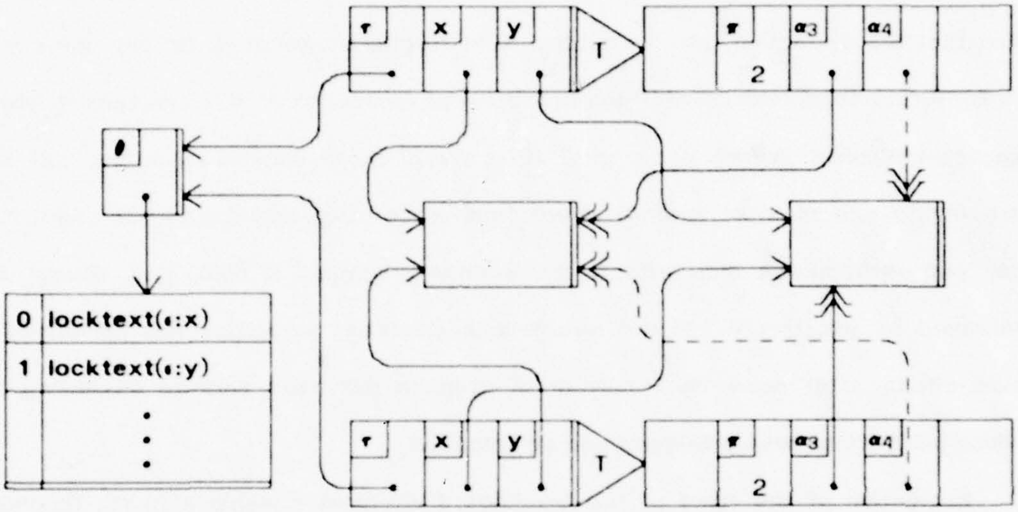


Figure 2.9: Deadlock in **locktexts**

This snapshot, from which inessential details have been omitted, shows a deadlock situation. Neither event can continue execution because each is waiting for a request to be granted, and neither request can be granted because each conflicts with another request that has already been granted (here is a case where the arbitration function of objects substantively affects the course of execution).* Thus the only choices available are to abort one or both events. Aborting an event will erase some previously granted access privileges, removing the obstacles to

*Situations similar to this conflict between **locktext** requests can arise out of conflicts between **gtext** and **locktext** requests.

granting other requests. Since an event is forbidden to make **gtext** and **locktext** requests after performing a side effect (*i.e.*, after performing any operation that sets the side-effect flag $\sigma$ to true), any event with pending requests that have not yet been granted can always be aborted. Conversely, any event that has performed a side effect (and therefore can no longer be aborted) cannot be prevented from executing because of unsatisfied access requests. Given our earlier assumption that the execution time, or number of instructions executed, for any one event is finite, the third fundamental guarantee about execution of VIM programs follows: *no unresolvable deadlock can occur*. If an event holds access privileges that are preventing execution of another event from proceeding, and the former event has not yet performed a side effect, the access privileges it holds may always be released by aborting it. If the event holding access privileges *has* performed a side effect, after some finite number of steps it will have finished executing, at which point its access privileges will be released.

Inspection of the rules of the blackboard interpreter yields a fourth theorem: *no event can conflict with itself* (*i.e.*, cause a "deadlock" with itself because of an unfortunate sequence of access requests). At first glance, it might seem that this could happen if, for example, the same event issued two **locktext** requests for the same object. The blackboard-interpreter routines GTEXT and LOCKTEXT, however, always check if the requested access has already been obtained by the current event, and thus avoid making any such redundant access requests. Therefore, if not interfered with, any event can eventually finish execution.

The various guarantees made by the rules of execution of VIM programs may be summarized as follows:

- since the effects of aborted event executions cannot be detected, the programmer need not concern himself with the possibility of that such executions could happen.

- since all deadlocks at the **gtext-locktext** level can be handled automatically, the programmer need not concern himself with the possibility of such deadlocks.

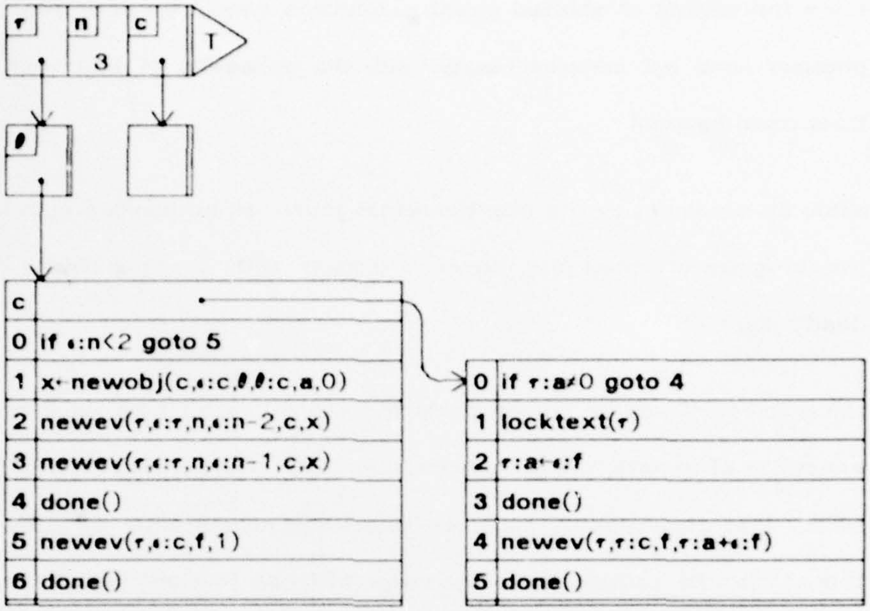
- since no result can be obtained which could not result from some sequential execution of events, the programmer need not concern himself with the possibility that other events might be executing concurrently with the execution of his. Of course, the programmer still has to worry about the consequences of other event executions intervening *between* executions of any two consecutive events of his.

### 2.5.3: An Extended Blackboard-Interpreter Example

As a more realistic example of the use of the blackboard interpreter, we now consider the sequence of snapshots in Figures 2.10(a) through (k). The program shown in these snapshots computes the *n*th Fibonacci number $\mathbf{f}_n$ using the relation

$$\mathbf{f}_n = \begin{cases} 1 & \text{if } n < 2 \\ \mathbf{f}_{n-2}+\mathbf{f}_{n-1} & \text{if } n \geq 2 \end{cases} \qquad (2.1)$$

If $n \geq 2$, the algorithm spawns parallel computations of $\mathbf{f}_{n-2}$ and $\mathbf{f}_{n-1}$. In the particular case of Figure 2.10, the program is used to compute $\mathbf{f}_3$.



```
c
0 if ε:n<2 goto 5
1 x←newobj(c,ε:c,ϕ,ϕ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

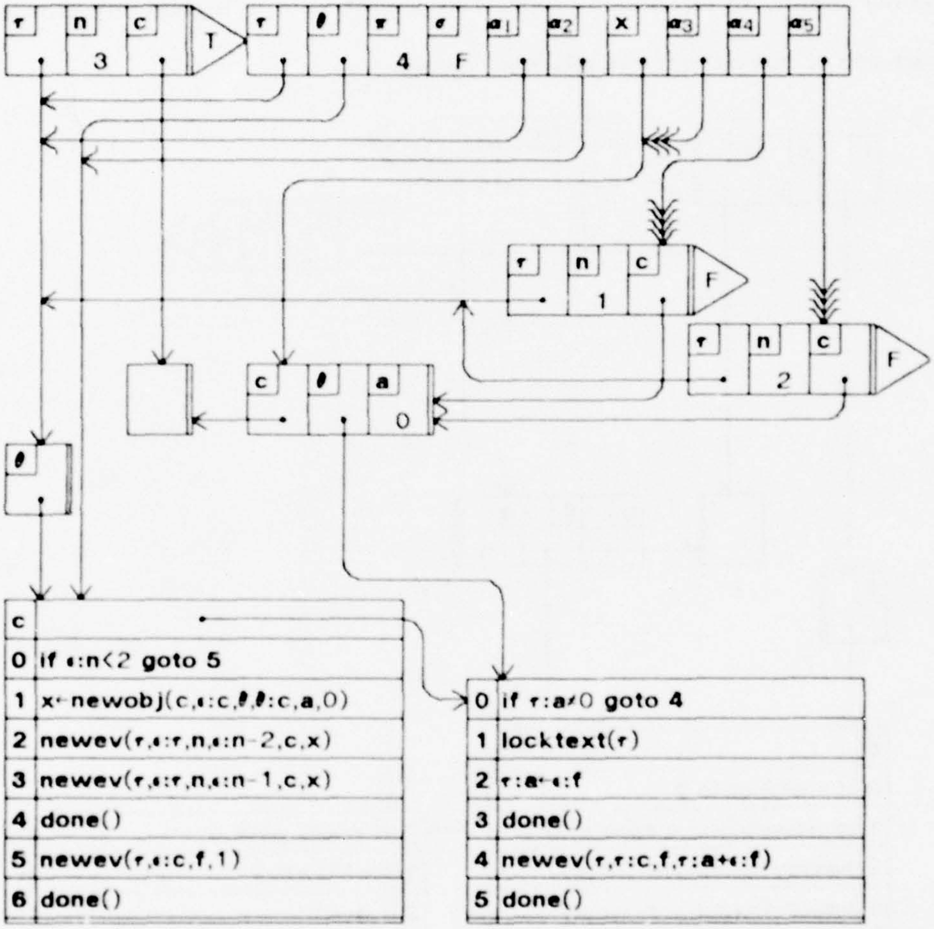
```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(a): Initial configuration for Fibonacci example.

In Figure 2.10(a), the sole event names the Fibonacci actor as its target object, gives the value 3 for the argument **n**, and contains in slot **c** a reference to a continuation object to receive the result. The contents of the continuation object are not shown here. The target object in Figure 2.10(a) contains only one reference, to its tyne object. The type object contains in slots **0** through **6** part of the Fibonacci program, and in slot **c** a reference to another object containing the remainder of the program. This arrangement illustrates a general property of type objects, namely that some slots may contain data even while other contain instructions. The program is best understood by watching its execution unfold, hence its logic is not explained here. The "machine language" used in it has a few new features, though. First of all, arithmetic expressions, rather than just slot identifiers, are used on the right-hand sides of assignment statements and as parameters to system calls. Secondly, the form

**if** *expression* **goto** *n*

is used to cause the program counter π to be set to *n* if *expression* is true. This provides a primitive facility for conditional branching.

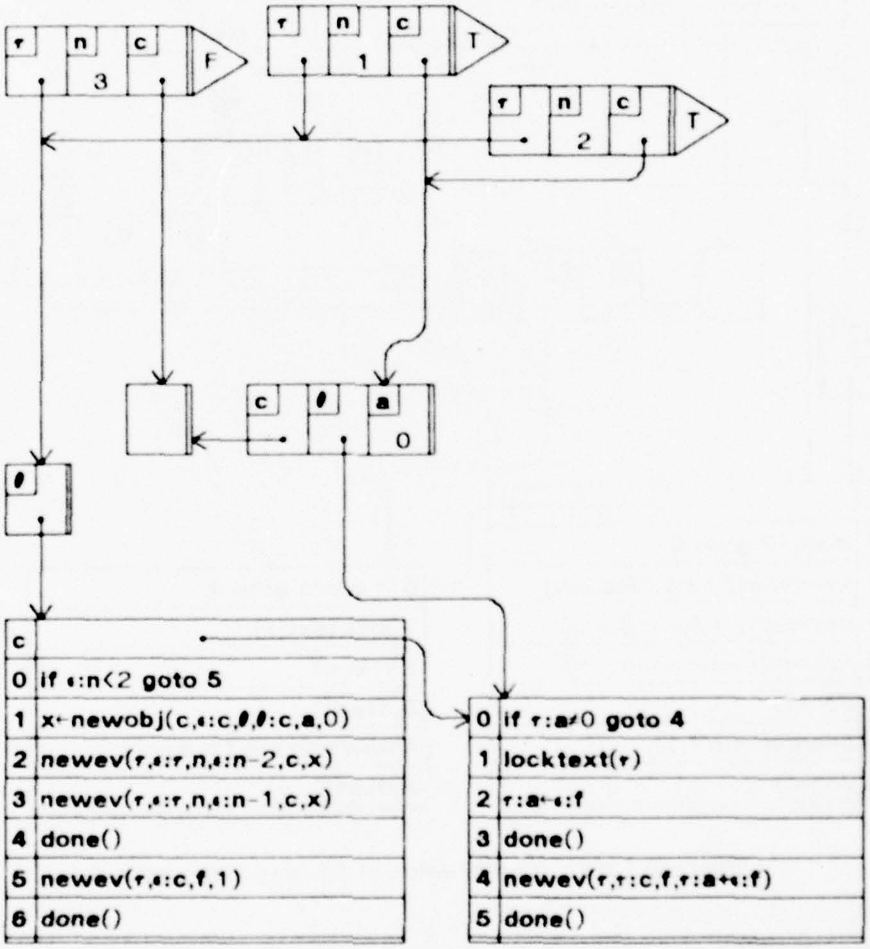


```
c
0 if ε:n<2 goto 5
1 x←newobj(c,ε:c,θ,θ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(b): Continuation of Fibonacci example.

*Figure* 2.10(b) shows the state of affairs after execution has proceeded past slots **0**, **1**, **2**, and **3** of the type object. The figure shows the new object and two new events created, and the activation record entries and access privileges created in the process. Note that no operation performed during this event execution caused the side-effect flag σ to be set to true. To keep the diagram from becoming even more cluttered, the convention has been established that drawing an arrow to the shaft of another arrow pointing at an object is as good as drawing an arrow directly to the object itself. This applies both to regular object reference arrows and to special access privilege arrows.

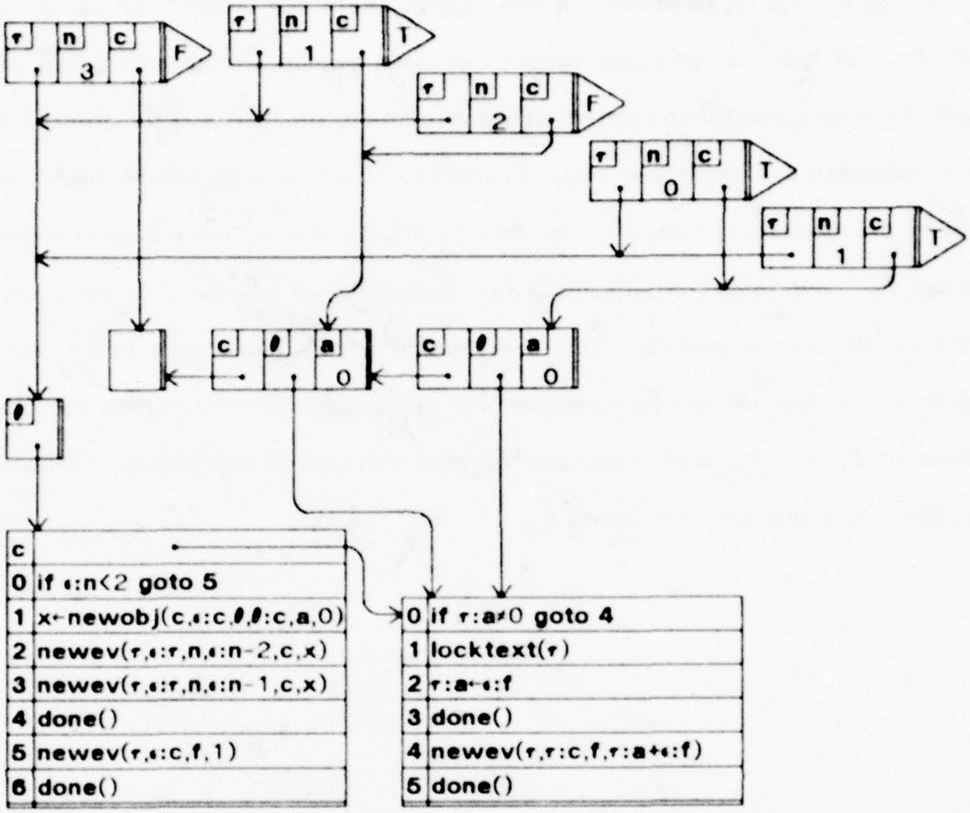


```
0 if ε:n<2 goto 5
1 x←newobj(c,ε:c,θ,θ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(c): Continuation of Fibonacci example.

In Figure 2.10(c), execution of the original event has finished, as can be seen from the fact that its activation record has been erased and its active flag set to false. Upon successful completion of its execution, the two events created by it were activated and they now await execution. These events represent recursive calls to the Fibonacci program to compute $f_1$ and $f_2$, whose sum will be the desired answer $f_3$. The target object for these events is, of course, still the Fibonacci actor. Both events give as their continuation a reference to a newly created object whose function will be to collect the two results generated from the computations of $f_1$ and $f_2$, add them, and forward the sum to the original continuation supplied in the request to compute $f_3$.



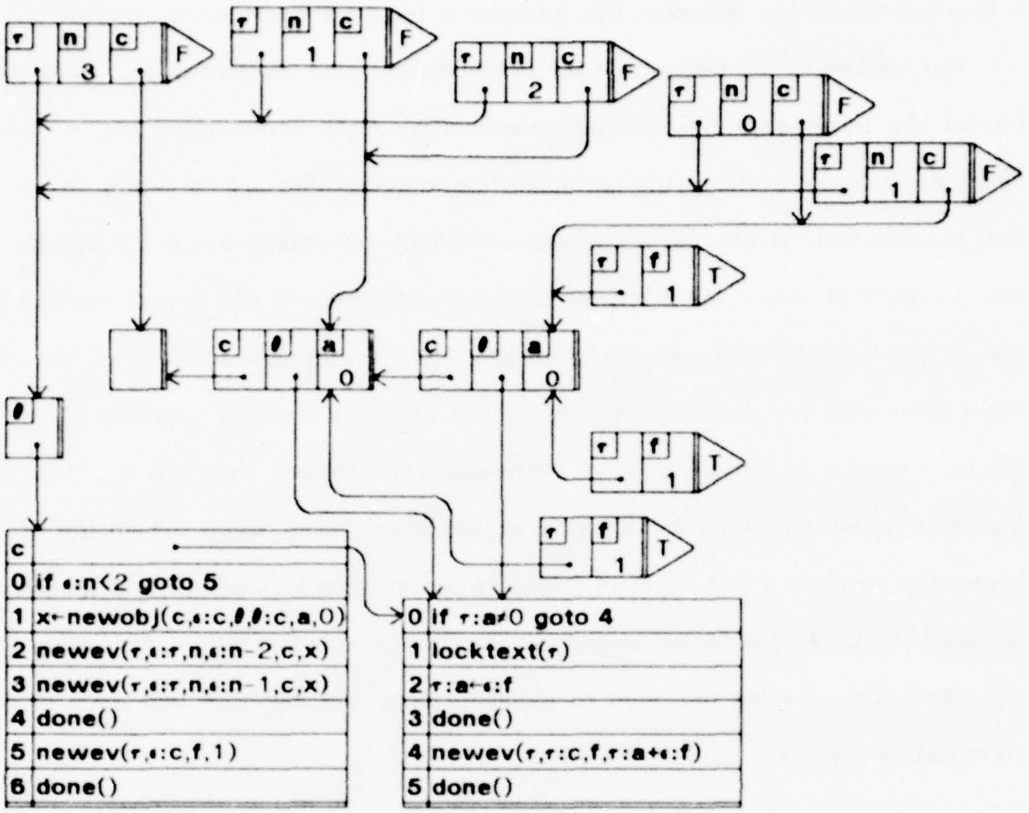
```
c
0 if ε:n<2 goto 5
1 x←newobj(c,ε:c,θ,θ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(d): Continuation of Fibonacci example.

For the transition to Figure 2.10(d), we have arbitrarily chosen to execute to completion the event whose job was to compute $f_2$. As before, this has led to the creation of two new events (for computing $f_0$ and $f_1$) and a new object. This new object is named as the continuation of each of the two new events, and will function to collect the values returned by them and return their sum as the value of $f_2$. The fact that recursive calls to the Fibonacci actor are occurring is evidenced by the stack-like structure of continuations that is developing. Since each call to the actor generates *two* new calls if $n \geq 2$, the stack-like structure is really, in general, a tree-like structure; however, this example is too small to demonstrate that.

The existence of these multiple continuations also illustrates the motivation behind the target-object/type-object mechanism. Each continuation has a slot **a** which records the state of the particular computations that are to return values to that continuation. If the slot **a** contains zero, both computations are still active. If the contents of slot **a** are nonzero, then the slot contains the result returned by one of the computations. At the completion of the other computation of the pair associated with that continuation, the sum of its value and the contents of slot **a** will be returned to the higher-level continuation referenced from slot **c**. Thus the algorithm followed by each continuation object is the same, even though the state information in slots **a** and **c** varies from one continuation to another. VIM allows us to place in slot ***ℓ*** of a target object a reference to another object containing the algorithm, thus avoiding the need to explicitly copy the algorithm into each of our continuation objects.

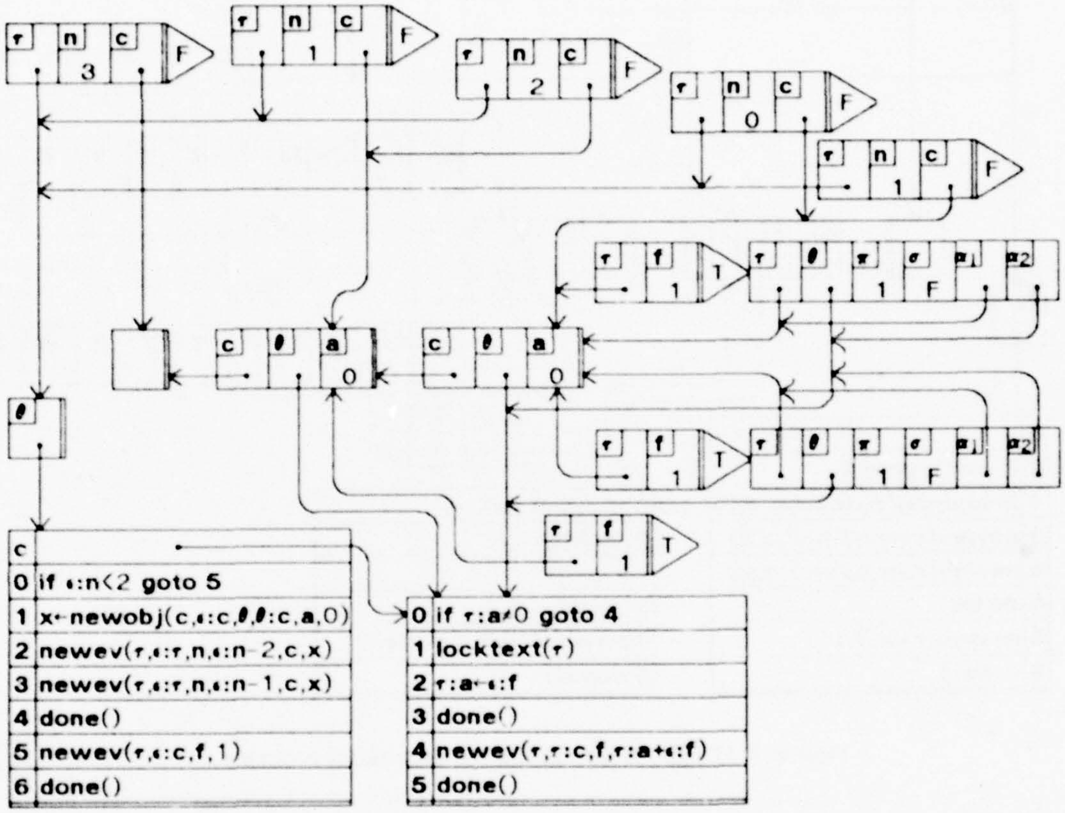


```
c
0 if ε:n<2 goto 5
1 x← newobj(c,ε:c,ℓ,ℓ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(e): Continuation of Fibonacci example.

All the remaining active calls to the Fibonacci actor have $n < 2$, and therefore cause a different flow of control through the actor's type object, through the **newev** request in slot **5**. The event created by this request simply returns 1 to the continuation specified in the event, furnishing the basis for our recursion. Figure 2.10(e) shows the result of executing, in any order, the three remaining active calls to the Fibonacci actor. Each generates a specific event returning 1 to its continuation. Note that every event from which it would be possible to reach the main body of the Fibonacci actor is now inactive. Not only could all these inactive events be reclaimed by a garbage collector, but since part of the Fibonacci actor is now inaccessible (unless it is referenced from other events not shown), the two objects associated with this part of it could also be garbage-collected. The code object on the right-hand side, however, is referenced by continuation objects that are still accessible, and hence must be retained for now.

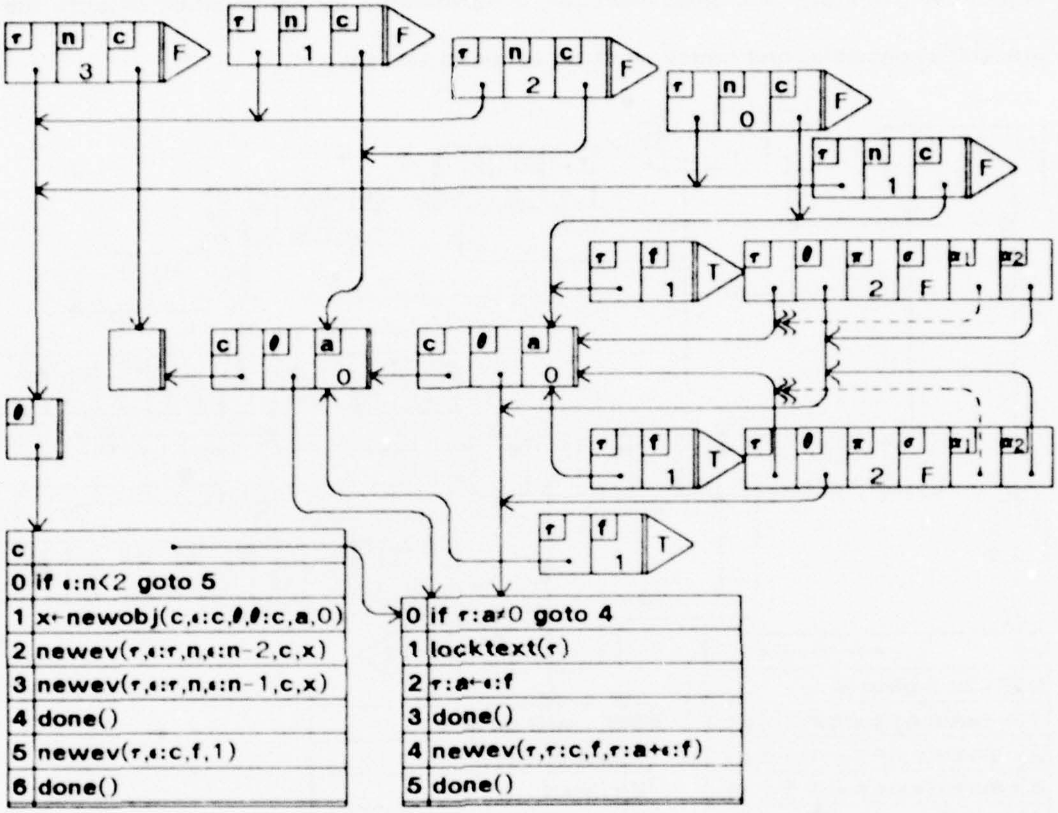


```
c
0 if ε:n<2 goto 5
1 x←newobj(c,ε:c,θ,θ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(f): Continuation of Fibonacci example.

In Figure 2.10(f), the two events returning to the rightmost continuation have concurrently started execution, and each has reached, but not yet begun to execute, the **locktext** request in slot **1** of their type object. Both executions having obtained (through the standard start-up sequence) **gtext** access to their target object, and both having determined that slot **a** in that object contains zero, each will now attempt to lock the object and write its result (found in slot **f** of the event) into slot **a** of the target object.



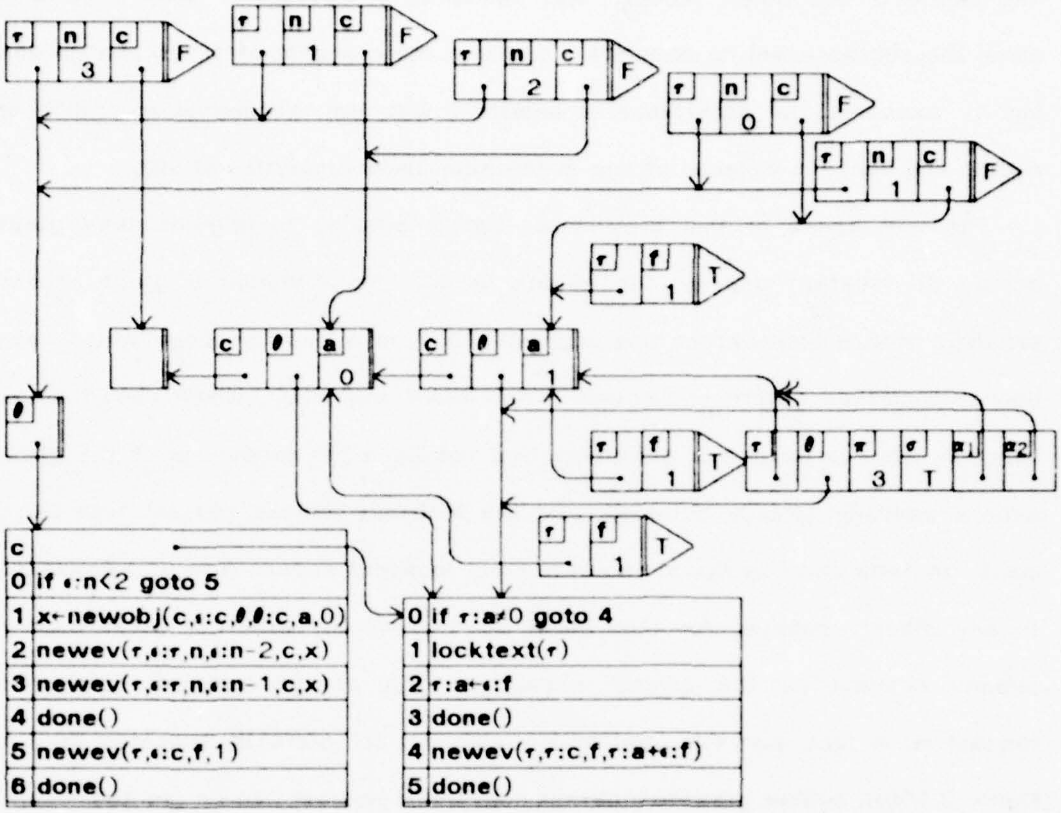
```
c
0 if ε:n<2 goto 5
1 x←newobj(c,ε:c,θ,θ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(g): Continuation of Fibonacci example.

Figure 2.10(g) shows the result of executing the two **locktext** operations. Since the object being locked, namely the target object, is an object to which **gtext** access had already been obtained, the access requests shown in Figure 2.10(g) are **gtext-to-locktext** conversion requests, rather than simple **locktext** requests. If the requests were simple **locktext** requests, one of them could be granted, execution of that event could complete, then the other request could be granted, and execution of that event could complete. Each event would write a 1 into slot **a** of the target object. This sequence of operations would be wrong, since the second event to complete would see the contents of slot **a** change during its execution, an occurrence inconsistent with any sequential execution of events and hence in violation of one of the advertised properties of VIM.

The root cause of this problem is that in seeking to upgrade their **gtext** access to **locktext** access, the events would have replaced a **gtext** access *privilege* with a **locktext** access *request*. Thus an access privilege would have been relinquished before completion of an event execution, which should never happen. What is needed is something that combines the properties of the **gtext** access privilege already obtained, and the **locktext** access request now being made. In particular, this hybrid should prevent **locktext** access from being awarded to any other requestor, for this would be inconsistent with the **gtext** access already enjoyed by the original requestor. The **gtext-to-locktext** conversion request is, in fact, just this combination, and Rule R7 correctly indicates that, in *Figure 2.10(g), neither* ***gtext-to-locktext*** *conversion* request can be granted.

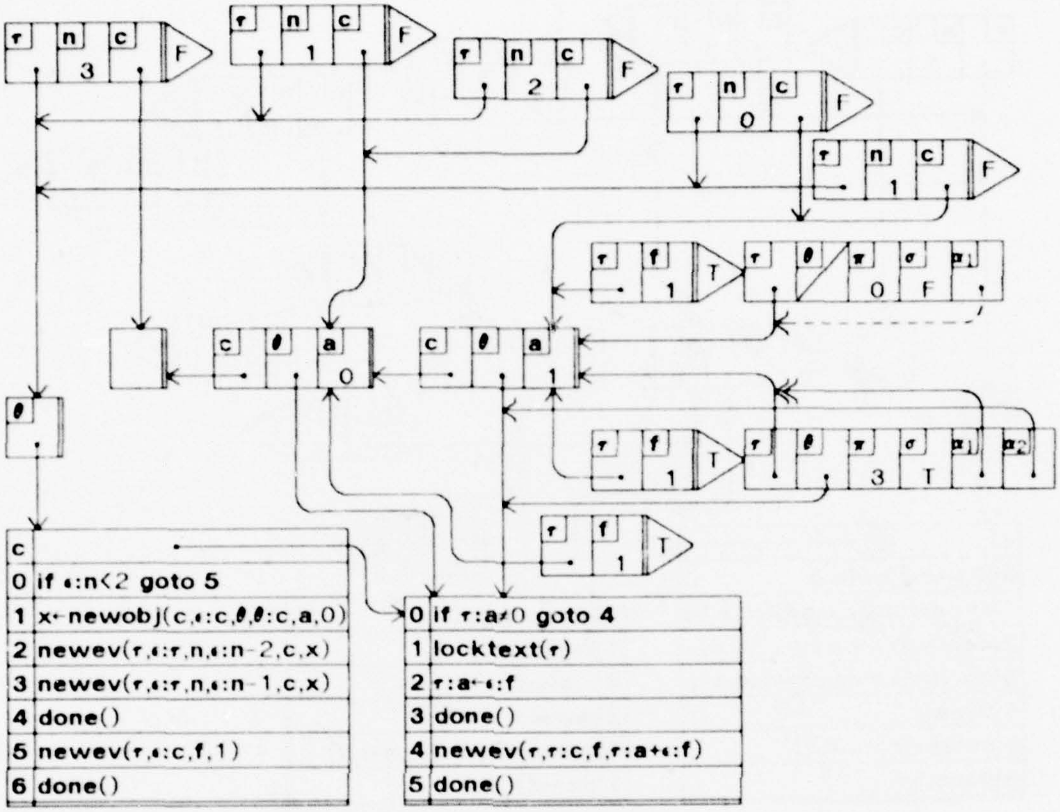


```
c
0 if ε:n<2 goto 5
1 x←newobj(c,ε:c,θ,θ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(h): Continuation of Fibonacci example.

The only alternative, then, is to abort an event. In Figure 2.10(h), the upper event has been aborted, whereupon the lower event was able to obtain **locktext** access to its target object and execute the side effect specified in slot **2** of its type object. Note that slot **a** of the target object now contains a 1 and that as a result the side effect flag of the event has been set to true. This means that the lower event can no longer be aborted.

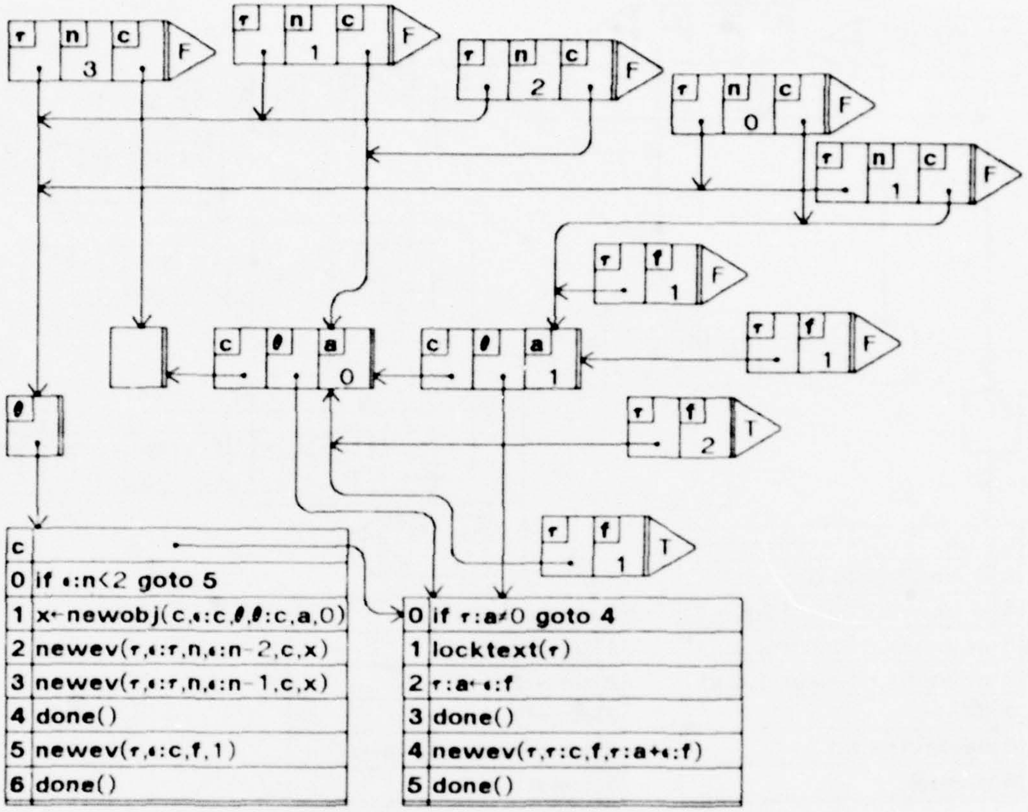


```
c
0 if ε:n<2 goto 5
1 x←newobj(c,ε:c,θ,θ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(i): Continuation of Fibonacci example.

In Figure 2.10(i), the upper event has again begun to execute, but has been stymied early in its start-up sequence by its inability to obtain **gtext** access to its target object. This access cannot be obtained because, under Rule R5, it would conflict with the **locktext** access already held by the lower event. This lower event can no longer be aborted, since it has performed a side effect, so the upper event will just have to wait until execution of the lower event completes successfully.

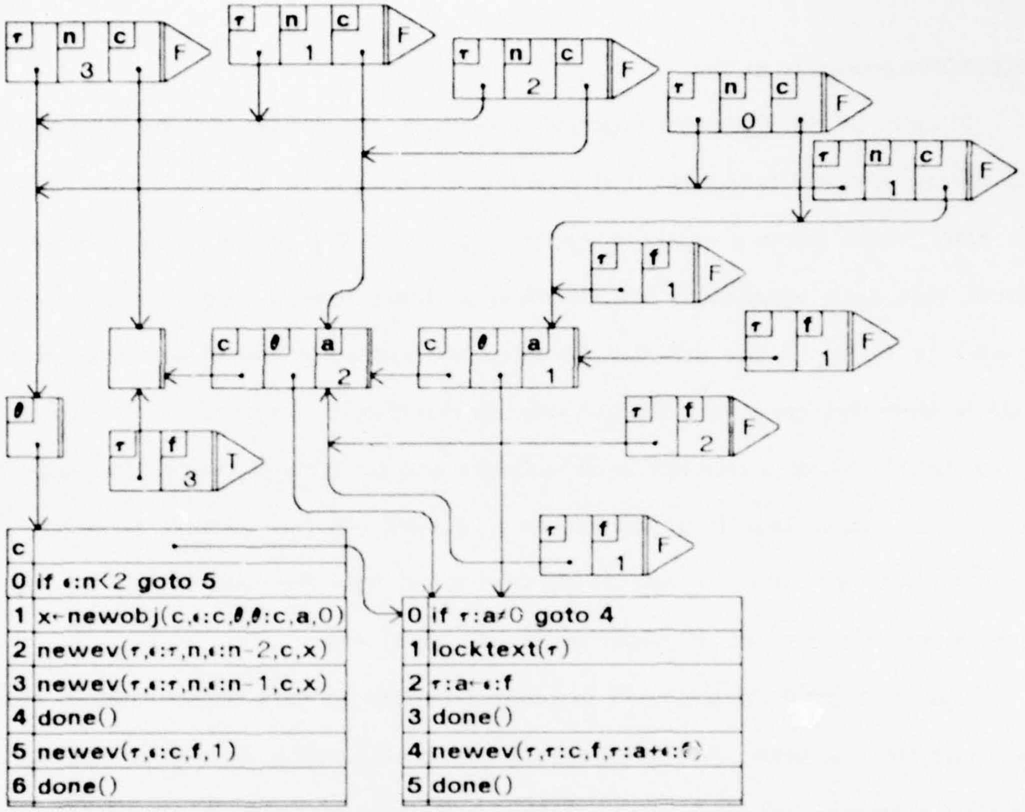


```
c
0 if ε:n<2 goto 5
1 x←newobj(c,ε:c,θ,θ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(j): Continuation of Fibonacci example.

In the transition to Figure 2.10(j), execution of the lower event has completed, allowing execution of the upper event to proceed. This execution, finding the contents of slot **a** in the target object to be nonzero, proceeds to the **newev** operation in slot **4** of the type object. The event created by this operation contains in its slot **f** the sum of **τ:a** and **ε:f**, which in this case is the value of $f_2$ that was desired.



```
c
0 if ε:n<2 goto 5
1 x←newobj(c,ε:c,θ,θ:c,a,0)
2 newev(τ,ε:τ,n,ε:n-2,c,x)
3 newev(τ,ε:τ,n,ε:n-1,c,x)
4 done()
5 newev(τ,ε:c,f,1)
6 done()
```

```
0 if τ:a≠0 goto 4
1 locktext(τ)
2 τ:a←ε:f
3 done()
4 newev(τ,τ:c,f,τ:a+ε:f)
5 done()
```

Figure 2.10(k): Final configuration for Fibonacci example.

Figure 2.10(k) shows a final configuration resulting from the execution of the two active events in Figure 2.10(j). The value of **f** from the upper of these two events has been copied into slot **a** of its target object, by the same mechanism illustrated in Figures 2.10(f) through (h). The one remaining active event in Figure 2.10(k) contains the sum of the results contained in the two active events of Figure 2.10(j), and in fact gives the correct answer $\mathbf{f}_3 = 3$. Note that at this point the only items that remain accessible are this one event and the caller's original continuation. All other items in the snapshot can be garbage-collected, unless referenced from other active events not shown.

### 2.5.4: Programming in VIM

In spite of the Fibonacci example just given, the reader may be forgiven for wondering whether (and how) it is possible to translate arbitrary programs written in other, more ordinary programming languages into VIM programs. An existence proof that such translations are possible, at least from a programming language similar to C[21], is the compiler for programs written in the MuSpeak language, which generates code that actually runs on the MuNet. The reader is referred to Strovink[34,35] for a description of MuSpeak and for a much deeper discussion of translation issues than it is possible to give here. In this section, however, we give an overview of the complications that result from the rules of the VIM interpreter and of how these complications may be dealt with.

One complication is the need to request access to data objects before actually operating on them. This calls for a modicum of planning, so that all necessary access privileges will have been obtained before an operation is attempted. This planning is complicated further by the fact that access requests (and other resource requests) by an event execution are illegal after a side effect has been performed.

These properties of VIM require every computation to be organized into quanta, where each quantum first obtains relevant access privileges and then performs any necessary side effects. It would indeed be feasible in some cases to organize an entire program, or at least a whole subroutine, into one such quantum, accumulating first all access privileges that will be needed, and then in effect executing the program. Optimization questions arise here — an event accumulating a large number of access privileges and taking a long time to complete is more likely to be aborted — but, more seriously, the single-quantum approach is in general not applicable because of other constraints imposed by VIM.

One such constraint is the time bound on execution of an event once it has performed a side effect. If a program contains loops of indefinite duration, such loops must be broken up into multiple event executions to ensure that no event execution continues forever.

Another factor favoring the decomposition of programs into smaller quanta is the impossibility, in VIM, of "suspending" execution of an event in the sense that execution of a routine in a procedure-oriented programming language may be thought of as "suspended" during a call to a subroutine. If a subroutine call is to be encoded as an event, which is a reasonable thing to do, then the body of the caller must be split into at least two quanta: one which contains preparation for the call and culminates in creation of the subroutine-call event itself, and one which contains the processing that should follow a return from the subroutine.

Since in VIM there is no implicit notion of a "return" to the previous activity when execution of a subroutine has completed, an explicit *continuation* object must be passed along with every subroutine call (as was done with the Fibonacci subroutine in our example). Invocation of the continuation as a target object should cause the resumption of the routine being returned to; that is, it should cause the execution of the instructions following the procedure call to which the continuation was supplied. Thus the continuation object will be the second of the two objects discussed at the end of the preceding paragraph. If the procedure being called returns one or more values, these may be included as parameters in the event that names the continuation as its target object. These results will then be available during execution of the continuation object.

The considerations discussed above concern differences in *control structure* between VIM and other languages. The object-orientation of VIM also makes for differences in *data structure*. For example, VIM has nothing quite like a stack, which is a concept relied upon in the implementation of many programming languages. Fortunately, VIM's version of "heap" storage is general enough to model stacks or virtually any other kind of data structure. If a language's environment structure is oriented toward frames on a stack, each such frame can be represented as a VIM object. Static, dynamic, and any other desired links can be handled easily by using object references. The VIM target-object/type-object mechanism can be used to make closure objects that bind together a piece of executable code and an environment for its execution. Such closure objects can be used for passing and returning procedure parameters; thus VIM is fully capable of handling even this relatively exotic operation.

VIM's lack of explicit support for stacks, and consequent reliance on linked lists of objects, undoubtedly has some cost in run-time overhead. However, in systems with many co-operating tasks executing concurrently, such as the multiprocessor systems for which VIM is designed, a stack is quite an inadequate environment structure anyhow. The environment structure associated with such situations will generally be that of a tree, with an active computation associated with each leaf node, and various parts of the tree "trunk" shared by tasks whose common ancestor used that environment. Linked lists of objects are a very reasonable way of implementing this kind of environment structure, and probably cost little more than "spaghetti stacks"[6] or other ways of achieving the same semantics.

Back in the realm of control structures, the notion of a continuation object, introduced above, is a key concept for handling all sorts of problems with translating programs into the VIM style. Essentially, if one can envision a program as a flowchart, such that the operation in each box takes a bounded amount of time and never needs to obtain access to additional data after performing a side effect (*i.e.*, such that the operation in each box conforms to the VIM rules for a legal event execution), then the translation into VIM is simple. A different continuation object can be generated to implement the operation in each box, and then only two details remain to be settled. One is the stringing together of the continuations, using object references, so that every continuation has a reference to each successor to which it might need to transfer control; the other is the provision of a suitable execution environment for each continuation, which may be done either using the VIM closure mechanism or by explicitly passing the environment to the continuation whenever it is invoked.

Using the flowchart analogy, it is easy to see how programs containing iteration, conditionals, sequencing of statements, and other control structures can be constructed out of the simple boxes that VIM allows. There is still a great deal of room for choice in determining exactly what functionality to pack into each box, as well as just how to organize collections of objects to represent particular data, and *how to implement the communication among continuations in a program*. These questions, whose answers can have significant performance ramifications, are considered at length by Strovink[34,35]. Whatever optimization decisions may be made in the course of a translation, though, it is clear that VIM is sufficiently powerful to serve as the basis for implementation of real programming languages, not only in the Turing-completeness sense, but in a very practical sense as well.

### 2.6: Changes of Object Format

Given the fact that VIM objects may persist for long periods of time, and come with all kinds of different lengths and sets of slot identifiers, it is convenient, though not logically necessary, to be able to change the format of an object. Such a facility in fact exists on the MuNet, and is in fact useful there. Changing the identifier of one or more slots in a blackboard-model object introduces no new complications — it can be treated simply as a side effect to that object and accordingly handled in the routine NEXT-INSTRUCTION. Changing the length of an object, however, leads to some subtle ramifications that are worthy of discussion.

Shortening of an object text (*i.e.*, moving its end marker to the left) can once again be viewed as a simple side effect to the object and handled by existing mechanism in NEXT-INSTRUCTION. If an object is to be lengthened, though, not only a side effect but also a resource request (for more space for the object text) are potentially involved. Unfortunately, a resource request cannot occur after a side effect has been performed (for then it would be impossible to abort the requesting event). Thus an object expansion would always have to be the first side effect in the execution of an event (the resource request would have to occur before the actual side effect). This rules out expanding two objects in one event execution, and is generally an annoying constraint on the construction of VIM programs.

A better solution is to split the object expansion operation into its two components: the resource request and the side effect. An event execution could then request additional space for all those objects whose expansion was contemplated, and only then perform the actual side effects. Adoption of this strategy requires a modification to the blackboard interpreter so that it is capable of representing an object for which additional space has been granted, but which has not necessarily yet been expanded to use that space. Consequently, the concept of the *authorized* end of an object is introduced, as illustrated in Figure 2.11. The total number of slots in an object, from its beginning to its authorized end, is referred to as the object's *authorized length*.

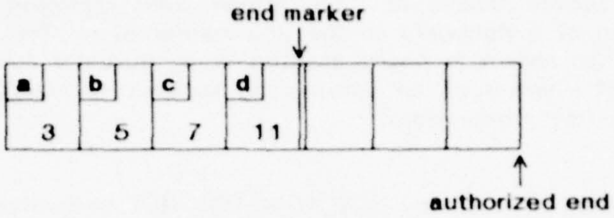


Figure 2.11: *Object with authorized end after end marker*

Only the slots to the left of an object's end marker are considered semantically part of the object; slots between the *end marker* and the *authorized* end are inaccessible and have unspecified contents. It is assumed that when the side effect of actually moving the end marker rightward is performed, identifiers and initial contents for the slots moved past will be specified. Conversely, when an end marker is moved leftward, the slots moved past become inaccessible and their contents become unspecified (but the authorized end of the object does not move, and remains at least as far right as the original position of the end marker).

There is no point in asking for additional slots to be authorized for an object unless moving the end marker is actually intended. This, in turn, is a side effect, for which **locktext**-type access is required. Thus it makes sense for the primitive (which we call **objxpnd**) authorizing additional slots for an object to also perform a **locktext** request for the object. This decision serves another purpose as well. If any object that has been **objxpnd**ed but possibly has not yet had its end marker moved is pointed to by a **locktext**-type access privilege, then the absence of such a privilege pointing to an object may be taken as an indication that the extra slots between the end marker and the authorized end are no longer needed and may, if desired, be reclaimed by the storage system. This consideration leads us to add one more basic rule to the blackboard interpreter:

**Rule R8:** If the object *o* has no **locktext**- or **newobj**-type access privileges (solid arrows with serifed double or triple arrowheads) impinging on it, move the authorized end of *o* rightward to the end marker of *o*. This rule states that, except when an object is newly created or is currently susceptible to being written into (by virtue of an outstanding **locktext** privilege), its authorized length need not be preserved.

Additionally, we must add a clause to NEXT-INSTRUCTION to handle **objxpnd** calls:

**objxpnd**(*object*,*size*): If SIDE-EFFECT-PERFORMED(*event*) then ERROR; else if HAS-LOCKTEXT(*event*,*object*) is false, then LOCKTEXT(*event*,*object*);[*] else advance $\pi$ in the activation record for *event* to the next instruction, and increase the authorized length of *object*, if necessary, to be at least as large as *size*.

Beyond these changes, NEXT-INSTRUCTION need only be updated to stipulate that moving an object's end marker is considered a write into the text of that object, and to incorporate the observations made above concerning the disposition of slots added to or removed from the accessible portion of an object text by moving its end marker.

[*] This is a "hack" to make sure that *object* has been **locktexted** before its authorized length is changed. The intent is that if HAS-LOCKTEXT returns false, **locktext** permission will be established and then the **objxpnd** request re-executed. This is why the program counter $\pi$ is not incremented until after **locktext** permission has been established. The inelegance of this definition does not correspond to any obscurity in the concept, nor does it seem to have foreboded any difficulties in implementation.

## 2.7: Summary

VIM recognizes two basic kinds of entities: objects and events. Objects may contain arbitrary binary data, and may also freely reference other objects. An event is a request to execute machine code found in the type object of the event's target object. An event may call upon VIM to create new events or objects, or gain access to object texts in either read-only or read/write mode. However, any event execution (1) must be able to complete in finite time after having performed its first side effect, and (2) may be aborted at any time prior to performing its first side effect. No event execution may request any resource after having performed a side effect, as defined by the blackboard interpreter. Thus execution of an event consists of two phases: an initial phase, when resources are accumulated, texts may be read but not written, and new objects and events may be created, followed by a final phase, which begins with the first side effect to the text of a previously existing object. During the final phase, no additional resources may be requested.* When an event execution terminates successfully, all events and objects created by it are formally installed, and all access privileges accumulated by it are terminated.

The attractiveness of VIM as a virtual machine for parallel processing derives from several desirable features. None of these capabilities is all-important, but each contributes to the appropriateness of the virtual machine for distributed computing, and VIM provides a nice package for them all. First, the **gtext-locktext** style of object access not only lends itself naturally to the solution of synchronization problems such as that handled by the continuation object in the Fibonacci example of Section 2.5.3, but gives an early signal of which resources will be

*The final phase need not occur; it is entirely reasonable for an event execution to perform no side effects.

required for a particular event execution, allowing the system to assemble them in a suitable location. It becomes unnecessary, for example, to give every processor in the network direct access to every memory, because such memory accesses as may be necessary during execution of an event can all be arranged to happen locally. This, in turn, allows reasonable performance to be exhibited even by a system using simple and low-cost technology in its communication hardware.

A consequence of this VIM object access style is that a program must contain checkpoints to which execution can be rolled back if conflicts, or deadlocks, develop. Such checkpoints are also useful for indicating points at which moving a task from one processor to another, or performing a housekeeping chore such as garbage collection, will be especially easy. Implementations of VIM are simplified by the legitimacy of rolling tasks back to their most recent checkpoints when such needs develop.

The VIM concept of "event" (in the concrete sense of "event" as a request to perform a computation) is a natural manifestation of such checkpoints. An event is also a compact and concise package, highly suitable for shipment between processors, containing all the information necessary for furthering the computation it belongs to. Finally, the use of events in the style allowed by VIM not only makes it very easy to express large degrees of parallelism, but also permits the construction of highly flexible and sophisticated control structures, as described by Hewitt[18].

The VIM object structure similarly allows the construction of flexible and sophisticated data structures, of the sort made popular by users of LISP[25]. Furthermore, it permits data to be represented as an interrelationship of small modules, enabling operations to proceed even in the presence of only parts of the data. This capability is important where a data structure may be flung across several processors in a system.

The synergy of these various features makes VIM a powerful, simple, and potentially efficient virtual machine particularly suitable for implementing modern object-oriented languages such as LISP[25] or CLU[24]. Services, such as garbage collection, required by these languages are implicitly part of every VIM implementation, simplifying the job of generating run-time systems. VIM objects are flexible enough to be used directly to represent objects in these languages without undue wastage or inefficiency. Finally, VIM events are quite capable of representing the computations that must be performed, including any desired use of parallelism.

# Chapter 3: Extensions to the Virtual Machine

The foregoing discussion of VIM describes an environment capable of supporting message-passing computation in a fairly elegant style; however, its usefulness in the broader context of a computer utility can be enhanced by the addition of a few extra features. The purpose of this chapter is to outline extensions that create a more hospitable environment for operating systems running under VIM;* readers more interested in other topics may skip this chapter with no loss of continuity. Elaborations of some of these extensions are also given in Appendix A. A reference tree network could be useful in practice even without these extensions; the purpose of the following discussion is to demonstrate the power and flexibility of the VIM approach, in that desirable system features can be incorporated without doing violence to the essential concepts of VIM.

In the context of VIM, an "operating system" might serve several functions. It might provide for long-term storage of user data, allow for user control and tracing of program execution, or mediate allocation and usage of various kinds of resources. For maximum flexibility, it is desirable to avoid building such an operating system into VIM. A preferable approach is to supply only the basic "hooks" necessary to enable a user to implement his own operating system services. The architecture of an operating system built around this philosophy is described by Gula[13,14]; our purpose here is to concentrate on the "hooks" in VIM necessary for support. To begin with, we concentrate particularly on a feature which facilitates user augmentation of the basic VIM environment, and also provides additional control over execution of user programs.

*For a discussion of such systems, see [13,14].

### 3.1: Event Tracing

In the primitive VIM environment, once a program has started running (by creation of some initial event) the only control that can be exercised over that program's execution is that built into the program itself. If the program goes into an infinite loop, there is no way to force its execution to be terminated. It is possible, within the primitive VIM context, to adopt a convention for creating events wherein a specific "trace actor" gains control before each new event execution. The trace actor will then determine whether to proceed normally with the event execution or whether, for example, to terminate that chain of events or invoke some trace routine before continuing. Unfortunately, if a user fails, either accidentally or on purpose, to obey this convention, execution of his program will not be controllable by this means. Another objection to the proposal is that it introduces substantial extra overhead to make a decision which in the vast majority of cases will be simply "go ahead."

A better solution is to associate with each event a *process object* which as part of its text would have a reference to such a trace actor. In the normal case, this could be a reference to some distinguished object meaning, simply, "proceed." In the context of the blackboard interpreter, every event could be considered to have a slot labeled with the reserved identifier ↓, containing a reference to the process object (see Figure 3.1). Before executing an event, VIM would examine the reference to that event's trace actor (found in slot ↓ of the event's process object). If that reference were found to say "proceed," normal event execution would ensue. Otherwise, the trace actor would be invoked, supplanting the normal invocation of the event's target object. The trace actor could then determine the proper course of action, and if execution of the event were called for, eventually transfer control to the event's target object.

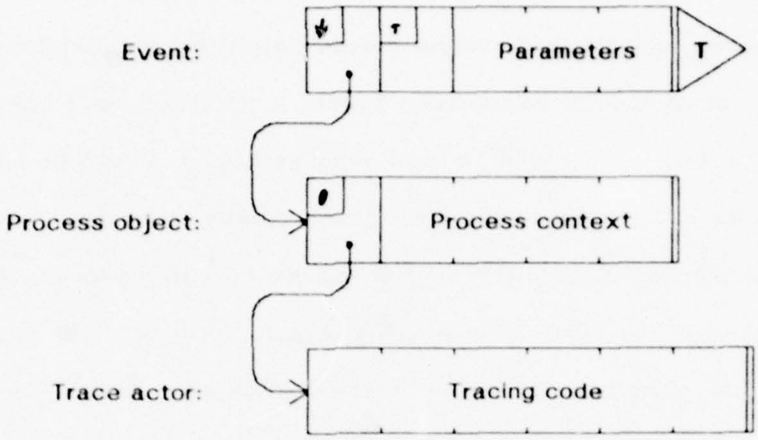


Figure 3.1: Process object and trace actor

By default, the process object would be passed down from an event to its successors (new events created by it), thus assuring continuity of the tracing mechanism through many generations of events. Due to its automatic inheritance by newly created events, the process object is the logical repository for all kinds of context information (such as user identification, privileges, quotas, etc.) normally associated with a *process* in a more traditional operating system.* When an attribute is stored in an event's process object, it will automatically apply, unless subsequently changed, to all events descended from the original event — that is, to the entire computation initiated by the first event. This notion of process object is even more flexible than the usual concept of process, since a single process object can even be shared by several concurrently executing events — but presumably only if these

*In order to maintain the integrity of this information, it would probably be desirable for VIM to put some restrictions on the ability to change an event's process object.

events are co-operating toward a single goal identified with that process object.

A method for structuring and using process objects is discussed by Gula[14]. For our purposes, only relatively simple details surrounding the creation of a semantic task (a set of events co-operating toward a particular goal) are of interest. When such a task is initiated, a new process object should be created, and assigned to the initial event of the task. If the creator of the task retains a reference to the process object, the creator will be able to manipulate the task by changing values in the text of the process object. For example, if it is desired to cancel the task, a trace actor can be inserted which will terminate any event that invokes it. The ability to easily perform this kind of manipulation is the reason for introducing the extra level of indirection of placing the reference to the trace actor within the text of a process object, rather than having it be directly an attribute of an event. Changing a particular process object's trace actor reference will suffice to influence *all* events using that process object, avoiding any need to explicitly know the identities of the affected events.

This trace actor mechanism is perhaps not yet powerful enough to meet all reasonable needs. Halting a particular line of computation is made simple enough, but gathering execution statistics or performing an event-by-event execution trace is still cumbersome. This is because the trace actor will have to explicitly transfer control to the event's target object when done with its own tracing operations. The target object will presumably want to access some object texts, or otherwise make requests that might result in the event's being aborted. By our earlier rules, such requests would be illegal after any side effect had been performed on any object text. Therefore, the trace actor must refrain from performing any side effect, such as incrementing a count of successful event executions, if it is subsequently going to transfer control to the event's target object. The only way the trace actor can leave behind a record of a successful execution, without violating any rules, is to create an event recording the occurrence. That event will only be activated if the subsequent execution of the target actor completes successfully. Assuming the event is indeed activated, it can then trigger an arbitrarily complex series of computations, including side effects, to update the relevant statistics.

Thus statistic-gathering, although cumbersome, is not impossible. The operation which remains intractable is interactive tracing, event by event, of a program execution. Here, it may be desired to construct a trace actor which pauses before executing an event, waiting for user input before continuing. This much can be achieved using the trace actor mechanism thus far described. What is difficult is installing a new trace actor which will allow the current event to continue, *but pause before executing any descendant event* created by the current one.

This is similar to the problem of correctly implementing a "trace bit," specified to cause a trace interrupt before every instruction execution, on a processor of traditional architecture. The solution is also similar. An event must have two states: not-yet-traced and already-traced. These states are comparable to the active/inactive status indicated by an event's active flag. When a not-yet-traced event is scheduled for execution, its trace actor, if any, is invoked instead of its target object. After a successful trace actor invocation, the event is changed to the already-traced state. In this state, the target object is always invoked.* New events would ordinarily be created in the not-yet-traced state, for proper operation of the tracing mechanism. However, a special facility for creating already-traced

*It may be desirable to allow the interposition of a short, side-effect-free "aftertrace actor" here to permit, for example, immediate killing of events even if they have reached the already-traced state.

events would be useful to debugging software.

In summary, several advantages can be gained by associating with every event a *process object*. The process object is useful in implicitly passing context information down through a whole chain of successive events. It is also useful, in conjunction with the concept of the *trace actor*, for retaining control of a computation that has been started, and in tracing and gathering statistics. Its usefulness in this regard may be enhanced by distinguishing between already-traced events and not-yet-traced events.

### 3.2: Object Tracing

Another service which an operating system might want to use would be a facility for monitoring patterns of access to object texts. In a sense, this kind of capability is the dual of the event-tracing function described above, but some of the difficulties are greater. In any case, one can distinguish an analogous set of useful operations: gathering statistics regarding accesses to an object, intercepting such accesses, or suspending them to allow for some period of computation or user interaction. Thus, by analogy, one might imagine associating a "trace actor" with each object. Code associated with the trace actor would be invoked upon any attempt (via **gtext** or **locktext**) to access the text of the object. As before, some default value would mean "no tracing," whereupon access to the text of the object would follow the normal procedure outlined previously.

Gathering statistics regarding accesses to an object would be possible, subject to restrictions similar to those discussed in our first proposal for event tracing: no side effects could be directly performed by an object's trace actor, since the currently executing event might attempt additional resource requests after return from the trace actor. The trace actor could, however, create a new event to record the fact of the object's having been accessed. It is true that an attempt to create such an event is a resource request which might result in abortion of the current event, but this will never cause a problem, since the trace actor is only invoked in the course of processing a request which might in itself cause the current event to be aborted.

Another possible use of a trace actor for an object would be in validating requests to access the object. Under VIM, the main control on access to an object is possession of a reference to the object, which is admittedly quite a coarse restraint. A trace actor could examine the process object of the current event to determine the identity of its owner. On the basis of this information it could decide whether to allow access to the object in the requested mode. If access were to be denied, the trace actor could terminate execution of the current event by signalling an error to VIM. This is a rather clumsy protection mechanism, though; further discussion of protection issues will appear below.

A third use for trace actors would be in conjunction with an incremental compilation or dynamic linking scheme. Uncompiled or "unlinked" objects could be given a special trace actor which would, upon first access, initiate a compilation or linking step to fill in the correct text for the object. The trace actor could suspend and record all requests for access to the object while being made ready, then re-create all the suspended events.

Perhaps the most important application of object trace actors, however, is in restoring a certain universality of message-passing systems that is compromised to some extent (in return for a potential gain in run-time efficiency) in the design of the basic VIM machine. In a pure message-passing system, all interaction between entities in the system is by exchange of messages; thus any module in the system can be fully specified by detailing its response to various different message protocols. The fact that the only way to communicate with a module is by passing it a message makes it simple to transparently splice additional processing into the message path ahead of the module. This could be done either for the purpose of augmenting or altering the semantics of the original module by modifying its response to certain messages, for the purpose of gathering usage statistics, or for the purpose of creating a virtual entity within the system. An example of this last application is the concept of a *future*[3], in which a dummy actor is supplied as a place-holder to collect and save messages received while the real actor that is to receive those messages is in preparation. Another example could occur in the context of *virtual memory management*: an actor could be "swapped out" and a dummy substituted for it to initiate a "swap in" operation upon receiving a message. At any rate, an important fact about the exclusive use of message-passing protocols is that they allow an arbitrary computation to be triggered by receipt of a message. This is the universality property alluded to above.

In VIM, however, there are two ways of interacting with an object. One is by sending it messages; the other is by accessing it directly (via **gtext** or **locktext**). The message-passing mode of interaction has the nice properties discussed above, but the accessing mode does not. The addition of object tracing to VIM restores these universality properties. By specifying an appropriate trace actor for an object, it is possible to trigger an arbitrary computation upon any access to the object. Thus any of the virtual memory or other mechanisms mentioned in the previous paragraph can be implemented. This general capability, in fact, is at the heart of the applicability of object tracing to all the uses discussed earlier in this section.

### 3.3: Protection

In the basic VIM environment, possession of a reference to an object is a grant of unlimited access to that object and all objects reachable by chains of references starting with that object. There are many situations in which it is desirable to give only partial access to an object, for example, access to invoke the object as a target object, but not to read or write its text. One might wish to forbid writing into a text to prevent some unauthorized agent from damaging the object; one might wish to prohibit reading from a text to avoid giving out references contained within it.

The trace actor mechanism outlined in the previous section may be used to discriminate among the access rights granted to different holders of a reference, provided that it is impossible for an unprivileged holder of a particular reference to masquerade as a privileged one. It can also be used to enforce "execute-only" access to an object either across the board or for certain users. Under execute-only access, the object's text can be accessed only during its own execution, *i.e.*, if it is the target object of the current event. A user distributing a reference to an execute-only object need not fear compromising the secrecy of the references contained therein, for they can only be accessed by the machine code associated with the execute-only object itself, which can apply arbitrarily complex security criteria before divulging any information.

Similarly, the trace actor mechanism can be used to enforce read-only access, wherein the text of an object may be read but not changed. In fact, across-the-board imposition of *read-only* or *execute-only* access (or both!) might be a common enough operation to warrant adding special bits (for efficiency) to object texts for specifying either kind of access restriction.

Combined with the ability to generate unique references by creating new objects, and use these references later as keys,* these admittedly *ad hoc* protection mechanisms might actually serve as the basis for a reasonably comprehensive security system. However, research on capability systems has led to much more elegant solutions to various protection problems[28,41]; it remains to be seen exactly how best to use these ideas in the VIM environment.

### 3.4: File System Support

Another area of operating system interaction is in helping the user manage his data. Since VIM supports a universe of objects with unlimited ability to reference each other, it would seem natural to use these as a basis for building any desired filing system. The user could represent his data as objects, and references to any such objects could be entered into a directory system, also composed of objects. Unfortunately, the storage of reasonable amounts of data in this fashion imposes a requirement on the network to maintain a large virtual memory space of objects, larger, for example, than could fit in the collective primary memories of the processors in the network. Thus low-level network software might be required to page objects in and out from secondary storage attached to the network at one or more nodes. Such a scheme has the advantage of being completely transparent to all programs running on the network, but the possible disadvantages of being harder to control explicitly where such control is desired, and of further enlarging and complicating the lowest-level network software.

An alternative approach is to leave the management of secondary storage to higher levels of "operating system" software, possibly supported by some special-purpose facilities implemented at lower levels. If the appropriate facilities are pro-

*The term "keys" here is as used by Henderson[17].

vided, *such a scheme* can remain reasonably transparent to user-level software (see Gula[14]).

A problem that must be faced by any scheme for data management on a large scale is that of reliability. Especially if viewed as a multiprocessor architecture, rather than a distributed computing network, it may be reasonably tolerable for a reference tree network to "crash" occasionally due to some failure, and it may be convenient to bring down the network periodically, for reconfiguration, maintenance, or whatever. Such a "crash" may entail the interruption, even the loss, of all computing then occurring on the network, but it is not tolerable for it also to entail the loss of all data stored in the system. Thus whatever *large-scale data storage* scheme is used must be "stable" in the sense of being able to survive virtually any conceivable incident on the system substantially intact. This implies that updates to data in "stable storage" must be recorded in a non-volatile fashion within a reasonable length of time. It also implies that when the system, or some part of it, is restarted, the *non-volatile record must be available and understandable*. These challenges must be faced by any software that implements the large-scale data storage mechanism, whether at the lowest level or at higher levels.

One possible way of managing stable data storage without building it into VIM is to use the object tracing mechanism. One "object manager" might be associated *with each mass-storage* device in the system. Inactive objects could be paged out to mass-storage devices and "aliases" understandable to the object managers substituted for them; every such alias could name as its trace actor the relevant object manager. When an object's text is again needed, a "trap" to the object manager could cause it to be retrieved from mass storage.

Unfortunately, accessing efficiency is not the only casualty when aliases are introduced; the network garbage-collection mechanism also suffers. When an object is converted into an alias, its object manager must record all references contained in the object's text, for use if the object ever needs to be reconstituted out of the alias. Since these objects are always referenced from the object manager, they cannot be deleted unless the object manager itself becomes inaccessible. This in turn cannot happen unless *every* object managed by the manager becomes inaccessible. This is a realistic possibility for some well-chosen assignments of objects to managers, but is by no means likely in the general case.

### 3.5: Summary

This chapter described possible modifications to the VIM virtual machine designed to facilitate controlling and gathering statistics about event executions and object accesses, handling protection problems, and providing other operating system services. A reference tree network could be quite useful, simply as a computing engine, even without any of these facilities; however, the incorporation of features such as those described would contribute to making a network a more well-rounded total computer system. For the purposes of this thesis, perhaps the real significance of the material discussed in this chapter lies elsewhere yet — it shows some of the flexibility of the VIM programming style in adapting to the additional needs considered in this chapter. This flexibility is an important reason for being enthusiastic about VIM as a good interface between programs, operating systems, and multiprocessor networks.

# Chapter 4: Architecture of Reference Tree Networks

## 4.1: The Physical Machine: Network Topology

As much as possible, it is the intent of this research to avoid becoming committed to the narrow technological characteristics of any one type of network. However, for concreteness, it is helpful to make certain assumptions. Furthermore, the algorithms presented depend on a certain *logical* organization of processors which may be more easily achievable with some physical organizations than with others.

The basic logical structure assumed by our implementation is a collection of processors, each with a private local memory (*i.e.*, no sharing of memory between processors is required), each connected to a small number of other processors which are its *neighbors*. The connections are bidirectional and symmetrical; thus if **A** is a neighbor of **B**, then **B** is a neighbor of **A**. Such an organization of processors is equivalent to an undirected graph (which may or may not contain cycles) in *which only a few arcs emanate from each node*. "Few" here is a relative term; its use is due to the fact that the overhead incurred by a processor will increase if that processor accumulates more neighbors. Thus it might well be appropriate for higher-capacity machines or machines with more of a commitment to serve the network (rather than, say, their owners) to have greater numbers of neighbors. A variety of topologies are consistent with these general restrictions (some are shown in Figure 4.1).

In addition to the specifications given above, each processor is required to have a processor ID unique to the entire system. (This ID is used in generating unique names and unique time stamps. If unique names and time stamps can be dispensed with, or generated in some other fashion, processor ID's are not necessary.) It is not necessary that all processors be identical, as long as all interpret

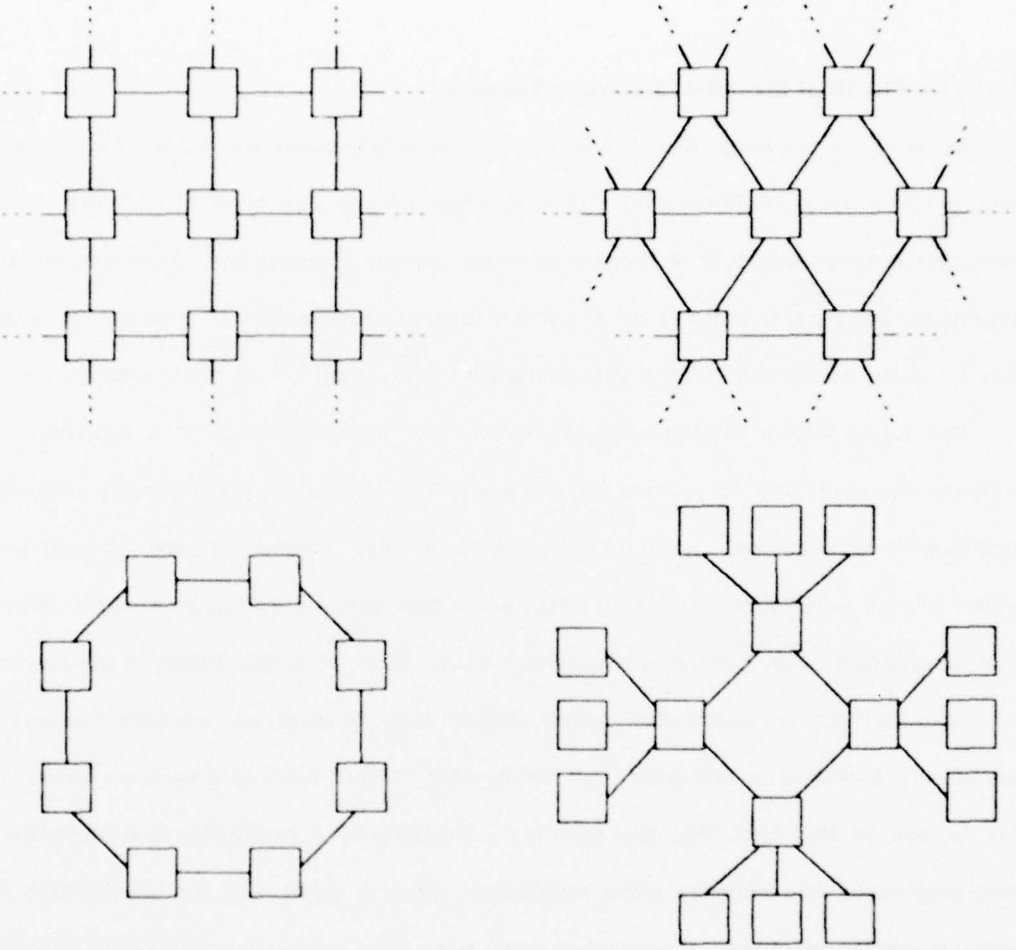


Figure 4.1: Some possible network topologies

substantially the same machine language.

The physical topology of the system need not follow the logical topology described above. The logical topology in effect constrains the paths over which information may travel; any physical topology which permits information flow over these channels may form the basis for an acceptable implementation. For example, an Ethernet (on which every processor can communicate directly with every other processor) could be made to support th϶ logical topology descril ed above either by declaring every processor to be a neighbor of every other (which might however impose a large amount of overhead on each), or by choosing for each processor a set of logical neighbors, and constraining the communication patterns on the net so that no processor ever sends a message to another processor that is not its neighbor. However, this is far from the ideal way of using an Ethernet.

In any case, the scheme presented here was certainly not designed for Ethernets, but rather for physical networks with properties closely matched to those of the logical network. Such networks have several advantages:

- expansibility. The network can be expanded to a very large size at little marginal cost. The space allocated for unique processor ID's does grow, but only logarithmically. Otherwise, expansion presumably involves simply hooking up new processors to the edges of the network, and has only a very local impact.

- bandwidth. For many topologies, there are potentially *a large* number of communication paths between any two points in the network; no central Ether or other medium serves as a bottleneck.

- reliability. Even a catastrophic hardware failure at some node is likely to affect only a limited number of other nodes (its neighbors). In systems with a central medium, there are components whose failure will stop all communication on the network. Of course, reliability is also strongly influenced by the software system's ability to carry on in the face of failures; admittedly, this thesis does not address this problem very thoroughly.

- flexibility. Many different processor and link technologies can in principle coexist in the system, allowing considerable freedom in picking the lowest-cost option for the performance desired at each point.

There are disadvantages to this kind of organization also. Chief among these is the need for extra processors to become involved in transactions between processors which are not neighbors, with the attendant overhead and delay. It is hoped that the scheduling strategies proposed in Chapter 7 will tend to minimize the need for this kind of transaction.

The topologies depicted in Figure 4.1 consist of units which are private processor-memory pairs, connected by communication lines. An attractive alternative might be composed of multiport processor and memory elements, as shown in Figure 4.2. Although each processor might have its own private storage for read-only and temporary data, all objects would be stored in the multiport memories. Assuming a nominal amount of local program or microcode memory at each processor, it should be possible to connect as many as three or four processors to each memory without serious degradation of performance due to access conflicts. An architecture such as this has some flexibility advantages — a processor can attach itself to any adjacent memory that contains data to be operated on, or even simultaneously operate on data stored in several different memories. Additionally, each processor can serve as an active communication link between any of its adjacent memories, moving data from one to another (or even performing more sophisticated operations) at high speed. For a tightly coupled local system, this multiport architecture is very attractive.

The reference tree network implementations we will discuss are described in terms of the kinds of topologies shown in Figure 4.1. They can be adapted to mul-

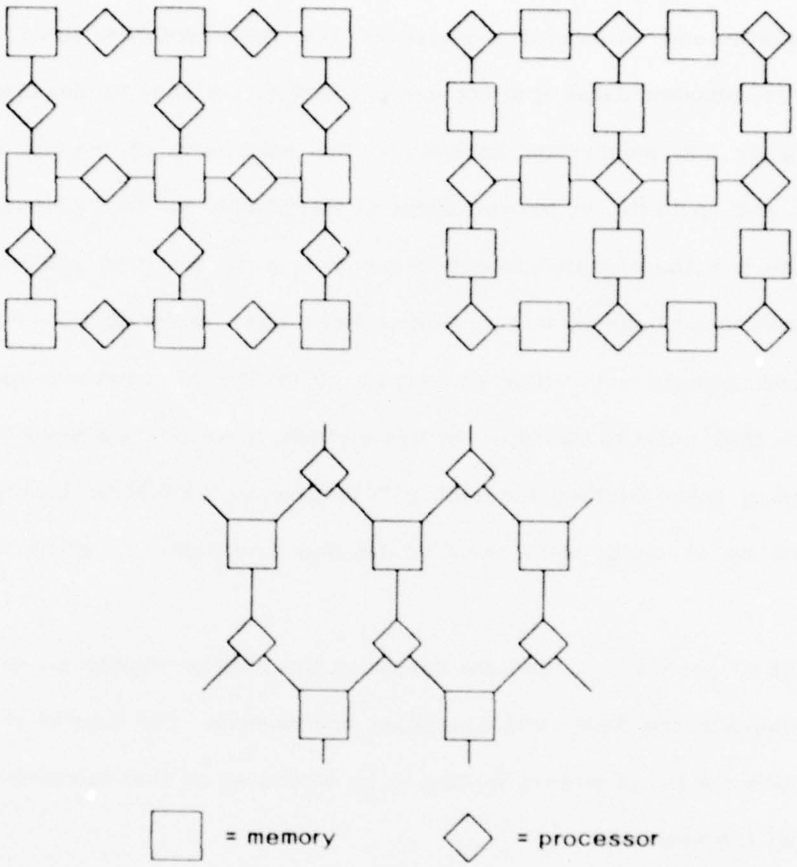


Figure 4.2: Multiport processor-memory networks

tiport networks such as those in Figure 4.2 by considering each memory to be a node, and drawing links between every pair of memories that share a common processor. Some modifications to the reference tree algorithms would undoubtedly help make the best use of the capabilities of multiport networks, but the basic concepts are quite applicable to either kind of network.

### 4.2: Dynamics of the Network

The static and structural aspects of our implementation have now been described in sufficient detail that we can proceed to consider its dynamics, that is, the motivation and mechanism surrounding the scheduling of events, sending of messages, and so forth. In the remainder of this thesis, we shall generally be concerned with a "steady-state" situation in which, as a result of some unspecified history, several processors in the system have been assigned things to do, and need to communicate with other processors containing, for whatever reason, data upon which they need to operate. In this section, therefore, we shall concentrate on developing some intuition for such a "steady-state" situation, briefly exploring mechanisms by which it might come to be and strategies by which it might be managed.

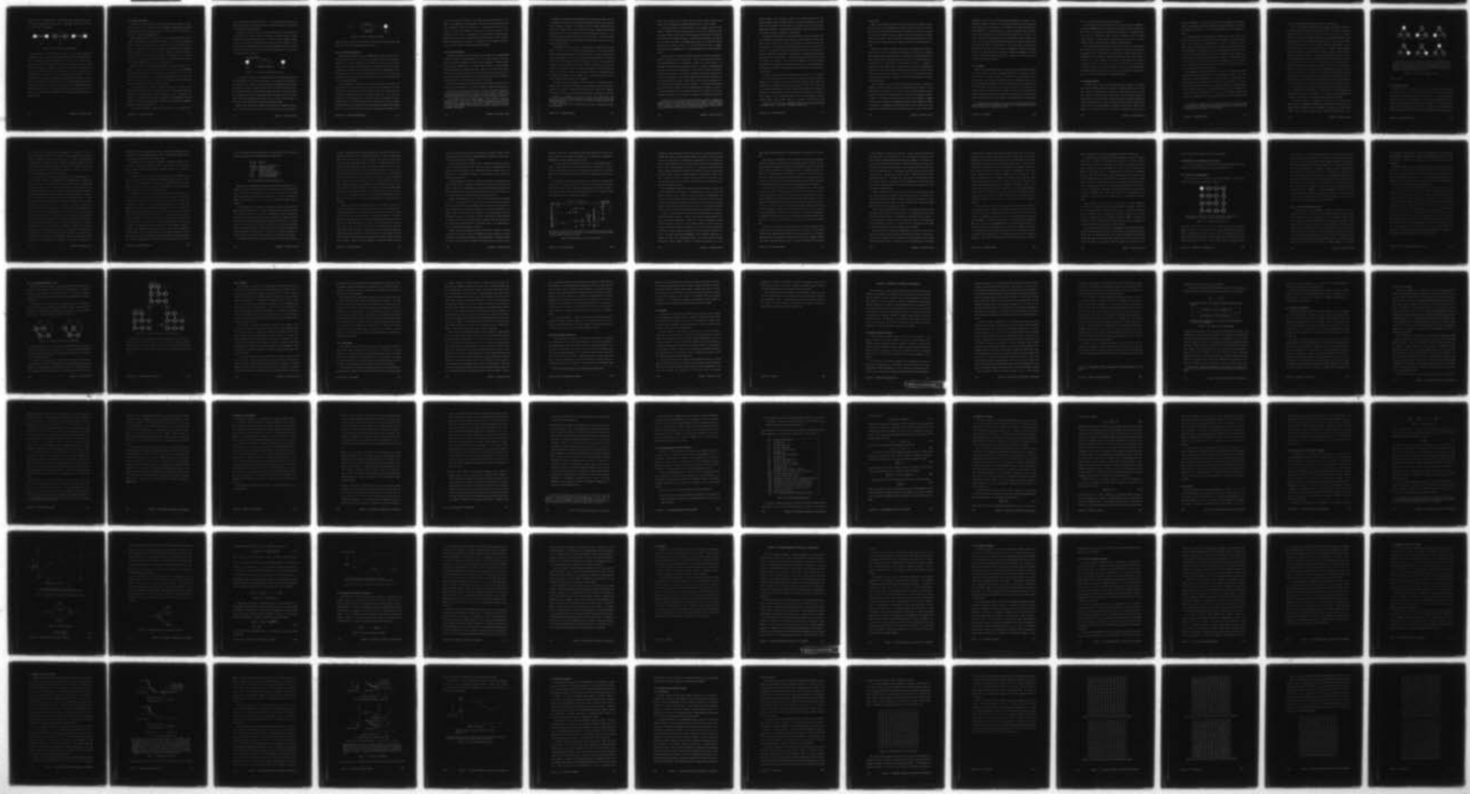
A piece of software or firmware known as the *monitor* resides on each processor and supports the basic VIM execution environment. The monitor maintains on each processor a list of events waiting to be executed on that processor. This list is known as the *event list*.

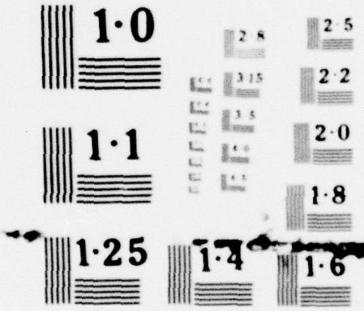
Let us assume that initially, by some means such as an operator typing at a keyboard, one processor somewhere in the network has been given an event to process. This processor will now have one event on its event list. The goal of a processor is to empty its event list, so our processor will take the new event off its list and see what to do with it. Most likely, the event will cause some computation to occur and then result in a new event's being added to the event list, whereupon the whole cycle will repeat. As long as this situation persists, and each event causes exactly one new one to take its place, there is little opportunity for other processors to get into the act.

Let us suppose, therefore, that at some point an event causes two or more

AD-A076 570   MASSACHUSETTS INST OF TECH  CAMBRIDGE LAB FOR COMPUTE--ETC  F/G 5/8
REFERENCE TREE NETWORKS: VIRTUAL MACHINE AND IMPLEMENTATION.(U)
JUL 79  R H HALSTEAD   N00014-75-C-0661
UNCLASSIFIED  MIT/LCS/TR-222   NL

2 OF 3
AD-
A076570





NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

events to be added to the event list. Henceforth there will be several events vying for our processor's attention at the same time. The processor might continue to operate on all the events itself by always taking the oldest event from the event list, producing its consequent events, and placing these at the end of the event list. Things could be speeded up, though, by taking advantage of the other processors in the system. If there are enough events on the event list, the overhead of sending some of them to a neighbor should be worth the increased processing power thus brought to bear on the problem.

The process by which it is decided what events to execute where is the subject of subsequent chapters; before taking that up it is a good idea to have a short look at the mechanics of moving events (and, as a consequence, objects) around. Sending an event to another processor is simple enough — all that is required is to send a list of references to the objects participating in the event. This, however, represents only a small part of the overhead required to really bring that processor into the action. Before the new processor can make any sense out of the event, it will need a copy of the text of the target object of the event. If it does not happen to already have this text, it will have to send an inquiry for it to some processor which does.* While this inquiry is being sent and replied to, the event cannot be processed. Depending on the nature of the target object, further inquiry-response cycles may be required to gather enough information to process the event. Finally, when the event is processed, a new event or events will most likely be generated, containing references related to those present in the original.

*This will probably be the original sender of the event; however, it may be that no neighbor of the inquirer has a copy of the text which enables it to reply immediately to the inquiry. In this case, the inquiry must be forwarded until it reaches a processor capable of replying, whereupon the reply must be forwarded back to the original inquirer.

Frequently, the texts of these objects will not be available locally either, so additional inquiry-response delays may ensue as subsequent related events are processed. Thus the true overhead associated with sending an event to another processor is the overhead required to establish a "working set" for that event and its consequents on that processor.

Having exposed the drawbacks of sending an event to another processor, it is worth mentioning some mitigating factors. First, if the event really represents the beginning of a computation that will proceed for a time without requiring too much communication with other activities, using another processor can still come very close to doubling the effective computing speed of the system. Second, the efficiency of the system can probably be improved greatly through various forms of tuning. For example, the sender of an event might also send a selection of object texts likely to be asked for by the receiver in the process of setting up its "working set." Similarly, the reply to an inquiry might not be just one object text, but a collection of related texts including the one asked for. These strategies, which are analogous to virtual memory strategies that attempt to predict which pages should be kept in core, might not do much to reduce the number of bytes sent over communication lines, but would reduce the number of wasteful delays between sending of inquiries and receipt of responses.

Fortunately, the working set that must be accumulated by an event running on a new processor can often be reasonably compact. Suppose, for instance, that the event is a function receiving arguments and a continuation. A working set that will allow this computation to proceed for a while would include the body of the function (or at least those parts of the body which will be exercised by the arguments given) and the texts of the arguments. If the arguments are numbers, no extra information is required. If the arguments are highly structured, it is probable (depending on the nature of the function) that only a portion of the texts accessible from the argument references will ever be accessed. Note that the text of the continuation is not needed at all — at least not until the function returns a value, which might be a natural point to transfer responsibility back to the first processor, where the text of the *continuation* is presumably located.

To summarize, then, we can imagine the system beginning to operate with one event at a particular processor. Activity will spread to other processors as the parallelism of the program increases and overloaded processors send events to their less-loaded neighbors. Perhaps at some later point the various threads of the computation will either die out or begin to come together again. Thus the number of active processors might shrink until there is again only one active processor with one event to process — perhaps "print out the answer." Of course, another possibility is that the system is constantly receiving new events to process, as users come up with new tasks for the system to perform. This will lead to more of a steady-state situation where, at any time, some tasks are just beginning, some just ending, and others in various growing or shrinking phases. In either case, the system will suffer from some amount of overhead and inefficiency as processors build up working sets for newly acquired events, but may gain even more from the application of extra processing power to its business.

## Chapter 5: Reference Trees

We now consider the details of an implementation capable of supporting the VIM object structure. These details fall into two categories: those involving the bookkeeping that keeps track of the state of objects and their whereabouts, and those pertaining to the garbage-collection of objects.

Keeping track of an object in a reference tree network is accomplished by maintaining the processors which contain references to the object in a *connected, acyclic* graph called the *reference tree* for that object. Each reference tree consists of some subset of the nodes and arcs (processors and inter-neighbor links) of the network. The nodes which belong to the reference tree are chosen to include all the processors in the network which contain references to the object,* and the arcs are chosen in such a way that (1) the reference tree is *connected*, *i.e.*, it is possible to reach any node in the tree from any other node, traveling only over arcs that are in the tree, and (2) the tree is *acyclic* in that the arcs in the tree form no closed loops. (Note that the arcs in the reference tree are undirected, hence requirement (2) means that there should be no undirected cycles.) Put another way, there should be a unique path (using only arcs in the tree) from any node in a reference tree to any other. Additionally, every arc in a reference tree must go between two nodes that are in the tree. Reference trees are so named because they form unrooted trees embedded in the network (see Figure 5.1).

It is important to note that, in general, the reference trees for different objects need bear no relation to each other. In particular, it is not the case that

*Keeping a reference tree for an object connected may require it to include certain processors which do not contain any references to the object and would not otherwise need to be in the reference tree. We shall return to this problem later.

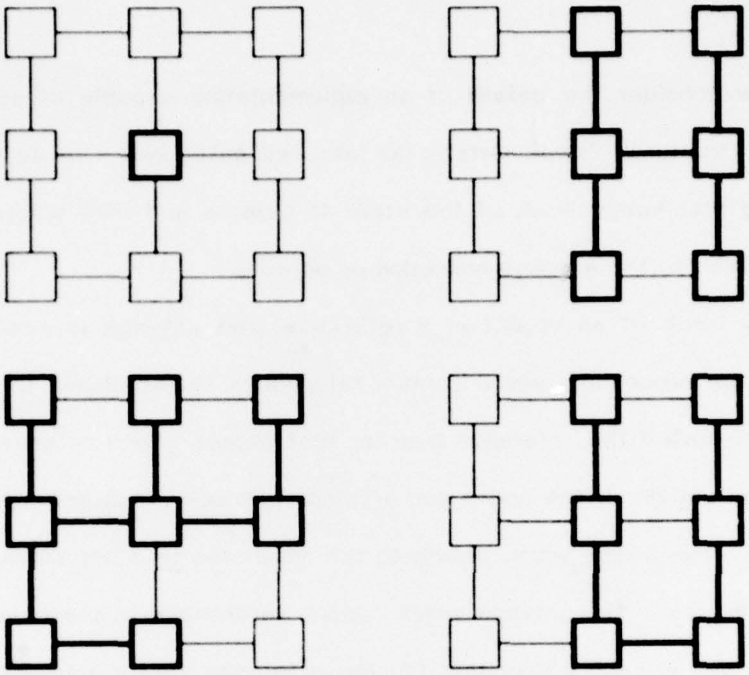


Figure 5.1: Examples of reference trees (in heavy lines)

there is one "reference tree" in the network, used for all objects. Central to the concept of reference trees is that they are free to grow and shrink dynamically, following changes in the roles of the corresponding objects in the operation of the system.

Also significant is the fact that reference trees can be maintained by a completely distributed mechanism in which each processor in a tree remembers only the state of its immediately adjacent links (*i.e.*, whether each link is in the tree or not). Processors not in the tree for an object, of course, have no references to the object, and need remember no information about it. Even the cycle-free nature of the tree can be preserved on the same strictly distributed basis — no central clearinghouse is needed to determine whether a cycle is being formed.

### 5.1: Object Text Management

Some subset of the processors having references to an object will also be *custodians* of a copy of the text of the object. The management of object texts has three basic goals: (1) to ensure that no object text is "lost" (*i.e.*, to ensure that at least one processor has custody of an object's text at all times), (2) to keep each processor in the reference tree for an object apprised of the directions in which it may send inquiries requesting a copy of the text, and (3) to provide a mechanism for performing side effects on object texts. The primary reason for requiring reference trees to be connected is so that any processor with a reference to an object can always communicate with all custodians for that object simply by following links that are part of the object's reference tree. Indeed, we will require that *all communication concerning an object travel strictly over links that are in the reference tree for that object.*

The acyclic nature of reference trees guarantees that, once a message has been sent across a particular link, it can never "loop back" to its sender unless either (1) it leaves the reference tree, or (2) some processor returns the message back along the same link over which it was received. Thus when a processor wants to send a message to a custodian of a text, all it needs to know locally is "which way" to send it initially; *i.e.*, which of its neighbors in the reference tree lies along the unique path from it to the desired custodian. *It does not even need to know the identity of the custodian*, since the message can be forwarded from node to node using only the local information telling which way the desired custodian lies. *Consequently, the only information processors need to record concerning the location of custodians is which ways (through the reference tree) they may be found.* This in turn means that local changes in custody need only cause data base updates in the processors directly involved — the remainder of the reference tree will not be affected.

**gtext** requests may be handled using an inquiry-response protocol in which a processor containing an event which has performed a **gtext** request for an object may send an inquiry through that object's reference tree to a custodian. The custodian may then reply by sending a copy of the text of the object back to the original inquirer.

Since a **gtext** request is for read-only access, multiple copies of an object text may simultaneously serve **gtext** requests in different locations. The handling of **locktext** requests is more complicated. The basic strategy for satisfying a **locktext** request is to eliminate all redundant copies of the text in question, leaving only one copy at the site where the **locktext** request was made. Once this has been accomplished, the requestor may perform any desired side effects on the text, without fear that some other copy of the text might escape the update. Unfortunately, this strategy leads to new possibilities for deadlock: if two processors simultaneously try to perform **locktext** requests for the same object (each requesting to have the unique remaining copy of the object's text), some means must exist for inducing all the processors in the object's reference tree to co-operate in satisfying first one of the requests and then the other. The way this is accomplished in a reference tree network is by associating with each **locktext** request a unique priority chosen from some totally ordered set. Then at every processor which must co-operate in satisfying a **locktext** request, priorities of conflicting requests can be compared and the highest-priority request honored. Furthermore, since **gtext** requests can conflict with **locktext** requests, **gtext**-type inquiries must also carry priorities.

A minor augmentation of the scheme suggested in the preceding paragraph is necessary to avoid the problem of *cyclic restart*[32]. An example of cyclic restart may occur in the presence of two events, *A* and *B*, and two objects, *X* and *Y*, where execution of *A* first requests **locktext** access to *X* and then *Y*, and execution of *B* requests **locktext** access to *Y* and then *X*. Suppose *A* is executing and has acquired access to *X*. Then *B* begins to execute, acquiring access to *Y*. When *A* asks for access to *Y*, its execution may be aborted and restarted, once again acquiring access to *X*. When *B* requests access to *X*, *B* may be aborted and restarted, leading to a repetition of the above cycle. This kind of "dynamic deadlock" may be avoided by establishing a total priority order on *events*, rather than just individual requests. Then if event *A* has a higher priority than event *B*, any access privileges ever requested during execution of *A*, *even during executions that were subsequently aborted*, could be made unavailable to *B* until successful completion of the execution of *A*. A simple measure of an event's priority, and one which is consistent with fair scheduling, is its "age" (suitably qualified, as with a processor ID, to make it unique). Using this measure, there will always be an "oldest" event which will be able to accumulate all the resources it needs in order to complete. Then the second oldest event will be able to complete, and so on, avoiding cyclic restart.

Consequently, we use *time stamps* as priorities, and associate higher priority with smaller (older) time stamps. It is not necessary that all processors have access to a common clock in order to generate these time stamps; a method by which these time stamps can be obtained, and their uniqueness guaranteed, is discussed below in section 5.1.4.

We now discuss in more detail the management of object texts. For each outgoing link of each reference tree it belongs to, a processor maintains the information shown in Table 5.2.*

- "text-this-way" bit
- "request-received" field (values are NULL, INQUIRY, and LOCK)
- request time stamp

Table 5.2: Text management information

For each reference tree it belongs to, a processor also records whether it has a copy of the text of the corresponding object (and, if so, the memory address of the copy!). Additionally, it keeps track of all events on that processor currently waiting for or using the text, and the timestamps of those events.

For reasons discussed below, every reference tree link has an asymmetry, the processors at its ends bearing a master-slave relationship to each other. In a stable condition, one end of a link (for a particular reference tree) is always master, and the other, *slave*. *Transiently*, both may be slaves, but both ends of a link can never simultaneously be masters of that link. The way in which the master-slave relationship is recorded and manipulated is described in a subsequent section. For now, suffice it to say that it is possible for a processor to determine whether it is master of a particular link of a particular reference tree; that a processor can relinquish its mastery of a link to the processor at the other end by sending a suitable message; and that a processor that is not master can send a "mastery-request" message which will induce the master to relinquish its mastery of the link. The pieces of information enumerated in Table 5.2, plus mastery infor-

*Various encoding schemes can be used to reduce the average amount of memory needed to store this information. For example, the contents of the request time stamp will only be relevant if the "request-received" field is not NULL, presumably an infrequent occurrence, on the average, for any particular object.

mation, constitute all the data used in manipulating object texts.

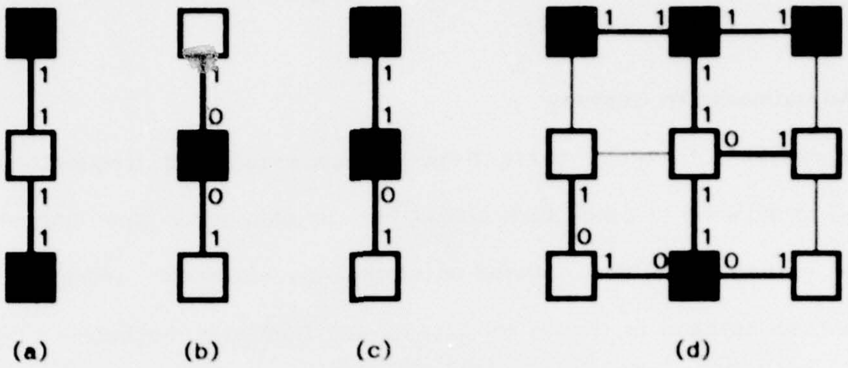
### 5.1.1: Adjustments to Custody

Transmission of object texts between processors will frequently occur in response to inquiries or other such stimuli, but can also occur "spontaneously" as a result of garbage-collection activity or storage-load balancing among processors. The spontaneous case is simpler to discuss, and illustrates mechanisms common to all communications of texts from one processor to another. We deal only with movement of copies of an object's text among processors already in the reference tree for the object; expansion and contraction of reference trees will be discussed later.

The representation of texts sent over communication links cannot be discussed without reference to concepts not yet introduced; for now, we note simply that a suitable representation exists.

Explaining object text management involves discussing the meaning of each text management datum in Table 5.2, and the mechanisms by which these data are used and updated. The "text-this-way" bit for a link indicates whether a custodian of the text can be reached, through the reference tree, starting with the link (see Figure 5.3). The "text-this-way" bit need be updated only when custody of a text changes.

The simplest kind of custody change occurs when a processor ceases being a custodian by simply deleting its copy of a text. For example, in Figure 5.3(a), two processors have copies of the object text. It should be possible for one of them to delete its copy without any damage being done. The problem is that processors in a reference tree must somehow agree on which copy of the text will be kept, a negotiation possibly global in scope. Simply knowing that another custodian exists



*Heavy lines indicate reference tree links; shaded boxes indicate processors with copies of the text. "Text-this-way" bit is shown next to each reference tree link emanating from a processor.*

Figure 5.3: "Text-this-way" bits

is not enough to allow a processor to delete its copy of a text, for the other custodian could then, symmetrically, make the same decision, and all copies of the object text would be lost.

Negotiation can be avoided if every processor, just before deleting its copy of a text, sends the text to another processor, thus ensuring that the information will not be lost. If the text message arrives at another processor already in possession of a copy of the text, the message can be discarded. Otherwise, the recipient processor must become a custodian for the text (although it always has the option of passing it on to yet another processor). If the two custodians in Figure 5.3(a) both transfer their custody to the center processor, one text message will be received and the other discarded, leading to the situation in Figure 5.3(b).

Note that every reference tree has embedded in it a (possibly null) subtree which includes all the processors that can reach a text in more than one direction, *i.e.*, that have at least two "text-this-way" bits set. In Figure 5.3(d), this subtree consists of the top center and middle center processors. Whenever such a subtree exists, it is bounded by two or more custodians (upper left, upper right, and bottom center, in Figure 5.3(d)) which each have only one "text-this-way" bit set. As long as each of these custodians obeys the text-preservation discipline set forth above, at least one copy of the text is guaranteed to be preserved no matter what the processors in the interior subtree do with any copies they might have. Therefore, we may exempt these interior processors from the requirement of sending out a copy of the text before deleting it. The rule used by the processors is *if at least two "text-this-way" bits for an object are set, a local copy of the object's text may be deleted without sending any messages.* Note that such a deletion (as by the top center processor in Figure 5.3(d)) does not require adjusting any "text-this-way" bits.

In general, sending a text from one processor to another requires adjusting some "text-this-way" bits. However, *it only requires adjusting the bits pertaining to the link over which the text was sent.* The sending processor always sets its "text-this-way" bit for the link, recording the fact that the text is out in that direction, and will continue to be, at least until a copy of the text comes back over the same link. The receiving processor, however, must be told by the sender how to set its "text-this-way" bit for the link. Presumably the bit is a 1 before the message is received (else the receiver would be very surprised to see a text coming from that direction!). The question, then, is whether there will continue to be a text in that direction after the message. If the sender keeps its copy of the text, the answer is yes. If there is another custodian in the same direction from the receiver as the sender ("behind" the sender, as it were) then the answer is yes. If there is no such custodian and the sender is going to delete its copy of the text upon sending the message, then the answer is no. Corresponding to these alternatives are two kinds of text messages, T+ *if the receiver should set his "text-this-*

way" bit, and T- if he should clear it. For example, going from Figure 5.3(a) or (c) to (b), the center processor would receive T- messages; going from (b) to (a) or (c), it would send T+ messages.

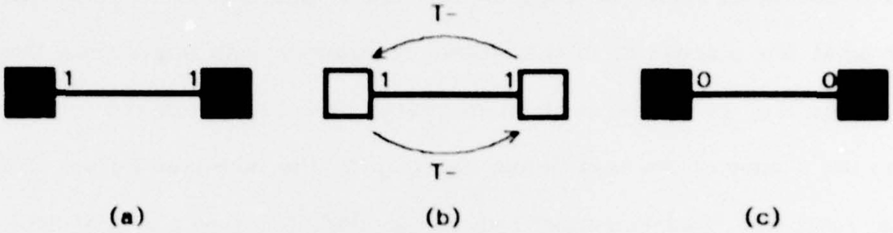


Figure 5.4: Race condition in sending texts

Unfortunately, this updating scheme can lead to an inconsistency among the "text-this-way" bits. Figure 5.4 shows what would happen if two neighboring processors simultaneously decided to delete their copies of a text. Each would be required to send a T- message to the other just before deleting its copy. Figure 5.4(b) shows the two T- messages in transit. Upon receiving its message, each processor would obediently clear its "text-this-way" bit and once again become a custodian of the text, leading to the inconsistent state in Figure 5.4(c). This can be prevented if a processor is prohibited from sending a text message over a link that it is not master of. Since both ends of a link cannot simultaneously be master, the situation depicted in Figure 5.4 can never arise. No unsolvable problems are created by this restriction — a processor desiring to send a text over a link of which it is not the master must simply send a "mastery-request" message and await the reply.

### 5.1.2: Inquiry Processing

We now turn to the handling of inquiries from processors wanting to obtain copies of a text. Such inquiries will result from **gtext** or **locktext** requests made by events executing on those processors. For the time being, we restrict our attention to inquiries of the **gtext** variety.

When a **gtext** request is issued for a text not available locally, the processor on which the request is issued has two alternatives: it can send the requesting event to another processor where its chances of satisfying the request will be better, or it can send out an inquiry to bring in the requested text. Assuming it chooses the latter, the processor must select a neighbor for whom the "text-this-way" bit is set and send an inquiry message to that neighbor. (If the inquiring processor is currently master of the link, it would do well to first relinquish mastery to its neighbor, facilitating the return of the desired text.)

When a processor receives an inquiry message, it responds immediately if it is able (*i.e.*, it has a copy of the text, is master of the link, and no pending **locktext** request with priority prevents it from sending out the requested text). If it is not able to reply immediately, it sets the "request-received" field for the link over which the inquiry was received to have the value INQUIRY and records the time stamp of the inquiry in the "request time stamp" field.

If an inquiry cannot be replied to because the replier is not master of the relevant link, a "mastery-request" message must be sent. When mastery has been obtained, the reply may then be sent (assuming no other circumstance then prevents it). If an immediate reply cannot be made due to a **locktext** request with priority, no further action need be taken. When the event requesting the **locktext** finishes, the inquiry will be honored.

If a processor receives an inquiry for a text which it does not have, then it will in general have to forward the inquiry. The "request-received" field set upon receiving the inquiry records the direction in which the desired text should be returned when it is finally obtained.

Forwarding of inquiries contains a couple of pitfalls to avoid. Generally speaking, forwarding of an inquiry received is analogous to sending of an inquiry needed to satisfy some internally generated need. In either case, some neighbor in whose direction a text exists is selected as the target of the inquiry. However, a neighbor selected for forwarding should not be the same neighbor from whom the original inquiry was received, or else an infinite loop of forwarding inquiries could occur in the situation shown in Figure 5.5.

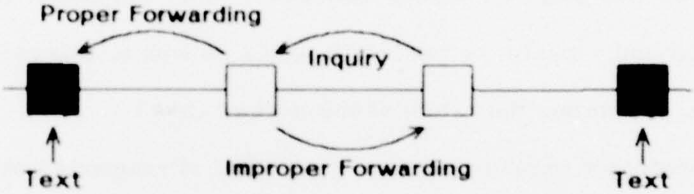


Figure 5.5: Inquiry received from direction of text

It is possible for there to be no legitimate direction for forwarding an inquiry (if the only direction in which a text can be found is in the direction of the sender of the original inquiry). However, the sender must have *thought* there was a text in the direction it chose for sending the original inquiry. Therefore, unless the reference tree data base has broken down, the inquiry must have "crossed in the mail" a T- message to the inquirer containing the desired text (see Figure 5.6). In other words, the inquiry *had already been replied to*, even before it was received. In such a case, the incoming inquiry message can simply be ignored.

When a processor that has "request-received" fields set to INQUIRY receives a copy of a desired text, it should send a copy of the text over every link over

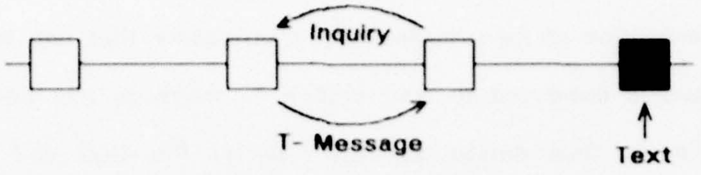


Figure 5.6: Inquiry crossing text message

which an inquiry for that text was received, clearing each "request-received" field to NULL as the corresponding text message is sent.

### 5.1.3: Side Effect Management

When an executing event issues a **locktext** request, and it is decided to continue execution of the event on the same processor, steps must be taken to ensure that a copy of the relevant text is brought to that processor, and that all other copies are deleted. These conditions are satisfied if and only if the processor has a copy of the object's text and all the processor's "text-this-way" bits for the object are zero. If some of the "text-this-way" bits are nonzero, then LOCK messages must be sent out over those links connected to the processor that have nonzero "text-this-way" bits.

A LOCK message is a request for the processor receiving it, and all other processors in its part of the reference tree (all processors reachable from it through the reference tree without using the link over which the LOCK message was received), to delete their copies of the relevant object text. Thus a processor receiving a LOCK message will, in general, set the corresponding "request-received" field to LOCK and forward the LOCK message over every link whose "text-this-way" bit is set, except for the link over which the original LOCK message was received. A processor that receives a LOCK message but has nowhere to forward it to is expected to delete its copy of the text and respond with a T- message as an indication of its compliance.[‡] A processor that has forwarded some LOCK messages is expected to wait until a T- message has been received in response to each, then delete its own copy of the text and make its own response, in the form of a T- message sent over the link whose "request-received" field had been set to LOCK. Finally, when the original processor on which the **locktext** was requested has received a T- message from each neighbor that had a text, it can proceed with execution of the requesting event.

### 5.1.4: Conflict Resolution

What has just been described is the simple case, where only a single **locktext** request is pending in the whole reference tree. In an actual situation, a processor may receive LOCK messages for the same object, originating on different processors, over several different links. To further complicate matters, it could simultaneously receive inquiry messages for the same object over yet other links. Finally, the processor could have locally executing events also requesting various kinds of access to the object. A processor in such a situation must serve as an arbiter and determine which of a set of conflicting requests to honor first.

To handle these conflicts, LOCK messages, inquiry messages, and events themselves, carry time stamps.[*] Every event is assigned a *unique* time stamp when it

[‡] It is possible to devise protocols that do not require actual text messages to be sent back from all the leaves in response to LOCK messages. A short "LOCK acknowledge" message can be sent instead. Unfortunately, such a scheme complicates the handling of inquiries by invalidating the premise illustrated in Figure 5.6.

[*] On the MuNet, inquiries are not time-stamped. This is adequate in every case except that of an object which is named in **gtext** requests and also continually in **locktext** requests. Under these circumstances, the **locktext** requests will take precedence and the **gtext** requests may never be satisfied. This has not occurred to any observable degree in normal operation but must be regarded as a "hole" in the MuNet implementation.

is created.[†] In principle, the precise algorithm for assigning time stamps to events is unimportant, so long as it never assigns the same time stamp to more than one event. However, in practice, it is desirable if the time stamps assigned increase with time, so that priority, which accrues to the events bearing the lowest time stamps, will accrue to the events that have been waiting the longest. It is not necessary to use a globally accessible clock to achieve this; a set of local clocks will do, provided that they are kept sufficiently synchronized that fairness criteria are not grossly violated.

On the MuNet, each local clock is a counter which is incremented once before each use. This provision is sufficient to guarantee that no single processor will ever give the same time stamp to more than one event. To assure that processors acting independently will not give out the same time stamps, each processor appends its unique processor ID to the value read out from its counter. Whenever a request or event message bearing a time stamp is received at a processor, the processor's time stamp counter is increased, if necessary, to match the time stamp value in the message. To make sure that rough synchrony is maintained even in the absence of request or event traffic, each processor periodically sends its current time stamp value to each of its neighbors.[‡]

The need to have time stamps for conflict resolution is unfortunate, since it imposes a space requirement of unknown size, depending on how long the system is expected to operate. Perhaps some scheme can be devised for "garbage-collecting" and recycling time stamps that have fallen into disuse, but any such

[†] Actually, on the MuNet, an event only receives a time stamp when it first needs one for purposes of these protocols. This policy reduces the speed with which time stamps are used up, but does not introduce any indeterminacies into the processing of requests.

[‡] For a further discussion of synchronization of clocks in distributed systems, see Lamport[22].

scheme must preserve the ordering among all entities currently bearing time stamps. No such scheme was considered during the course of this research.

A processor needing to arbitrate among a set of requests uses their time stamps to establish a priority order among them. It then acts to satisfy as many as possible, subject to the restriction that no request be satisfied at the expense of a request with an earlier time stamp. In practice, this means that if the request with the earliest time stamp is a **locktext**-type request, it is satisfied first and all other requests are held in abeyance. If the request that has priority is a **gtext**-type inquiry, it and all **gtexts** up to the earliest **locktext** can be honored simultaneously.*

By these means, a processor can determine which of the various requests of which it is aware should be satisfied first. It does not follow that the processor will singlehandedly have the *wherewithal to satisfy the selected request*. For example, if the highest-priority request is an inquiry, it cannot be satisfied without possession of a copy of the relevant text. Thus, in order to satisfy the request it picks as the most urgent, the processor may need to forward this request to one or more of its neighbors. Each of those neighbors will then compare the priority of the forwarded request to those of all the others it has received, and again act on the most urgent one, holding the others in abeyance. The only external information a processor needs in order to make this determination is the type (*i.e.*, inquiry or LOCK) and time stamp of the most urgent request which is known to each of its neighbors *and which it must co-operate in satisfying*. Thus whenever a processor

*If satisfaction of the request that has priority also incidentally satisfies some request made by an event with a much later time stamp, it is permissible to attempt to execute this latter event while the former request is still active. What is not permissible is to take actions to satisfy the latter request which would be inconsistent with the treatment required to satisfy the former.

sends an inquiry or LOCK message to a neighbor, it is in effect saying, "This is now my most urgent request involving this object. Its urgency is indicated by the time stamp in this message. You may forget any previous requests I have sent you regarding this object, for I will resend any of them if and when it becomes my most urgent request and your co-operation is still needed."

A strategy that reconciles our previous discussion of text management with this philosophy is as follows: when an inquiry or LOCK message arrives at a processor, set the "request-received" field accordingly and record the time stamp from the message in the "request time stamp" field for that object for that link (see Table 5.2). 'f some other request for the same object active on the same processor has a lower time stamp, take no further action — the newly received request must wait until it has top priority.† If the newly received request has the lowest time stamp of all, however, it must be honored immediately, using the algorithms described above for handling inquiries and LOCK messages. If those algorithms call for forwarding of requests, the forwarded requests must bear the same time stamp as the original.

An *inquiry* can be considered satisfied when a text (either T+ or T-) message is sent back over the same link from which the inquiry was received. Therefore, when a T+ or T- message is sent over a link whose "request-received" field is set to INQUIRY, the "request-received" field can be reset to NULL. This can be done even without knowing whether the text message was intended as a response to the inquiry or just happened to be sent while the inquiry was active. Likewise, a LOCK request can be considered satisfied, and the corresponding "request-received" field cleared to NULL, when a T- message is sent back over the

†Except that if the request is a **gtext**-style inquiry, and all requests with lower time stamps are also, then all may be satisfied simultaneously.

requesting link.

When a text message has been sent over a link, and the "request-received" field cleared, there is instantaneously no request active over the link. This situation will continue at least until the processor at the other end of the link has itself disposed of the request, bringing top priority (for its local operations) to another request for the object. Then if the processor which had sent the text must once again co-operate in satisfying the new request, it will receive another request message.

Note that no request message is ever sent to a neighbor whose "text-this-way" bit is not set. Thus processors at the fringes of a reference tree will only send requests inward, toward the areas of the reference tree where texts exist. Requests will never be sent outward past the last text in any particular branch of a reference tree. Consequently, if a processor has sent a T- message, in response to a LOCK request, say, it will thenceforth receive only inward-directed requests, but, until the processor once again has a copy of the text, none from the center of the reference tree (defining the "center" as that subtree of the reference tree which contains all custodians in the reference tree and has a custodian at every leaf node).

Discussion thus far has centered on the treatment of requests received from neighbors. Requests arising from locally executing events are treated in a very similar fashion. If processing such a request involves sending messages to neighbors, the time stamp of the requesting event is used in those messages. When the conditions of the request are met (a copy of the text available, for **gtext**; exclusive access to the text, for **locktext**), execution of the requesting event may be resumed. This resumption may conceivably lead to other requests, for other objects, but all requests made by an event remain active (*i.e.*, must continue to be satisfied) until execution of the event has been completed successfully, or until the event is aborted. Therefore, if a request from another event with a lower time stamp, either on the same processor or on a different one, conflicts with the continued satisfaction of a request made by a currently suspended event, this latter event must be aborted so that the higher-priority request can be honored. This does not cancel the requests made by the aborted event; they continue to be active, and to take priority except where temporarily superseded by requests with even lower time stamps. But satisfying one of the superseding requests, without aborting an event whose request has been superseded, violates the guarantees about access rights made by the definitions of **gtext** and **locktext**. Therefore, such an event must be aborted and held for re-execution at a time when all its requests can be given top priority.*

### 5.1.5: Summary

Within the framework of reference trees, an object text management protocol can be devised which is capable of handling multiple copies of object texts, reducing the number of copies to one when a side effect is to be performed, and arbitrating between asynchronously generated requests of differing priorities. Locations of texts within a reference tree are recorded by means of a distributed set of "text-this-way" bits kept at the various processors in the reference tree. These bits do not indicate the exact locations of texts, but only the directions, traveling through the reference tree, in which they may be found. As a result, local motions of texts require only local updates of "text-this-way" bits; other proces-

*On the MuNet, some complication is avoided by instantly aborting any event whose request cannot be immediately satisfied. Thus no "suspended" events ever exist, only aborted events awaiting re-execution.

sors in the reference tree do not need to be informed.

The object-custody protocol allows processors to request read-only (**gtext**) or read/write (**locktext**) copies of object texts on behalf of events executing on them. Each such request must be labeled with the priority of the requesting event, in the form of a unique time stamp. These time stamps are used to resolve conflicts between requests, and the "text-this-way" bits are used to forward requests to all processors that must co-operate in satisfying them.

By recording only partial information about text locations (only "text-this-way" bits instead of the actual identities of custodians of texts), this object text management protocol throws away the opportunity to perform certain kinds of optimizations, such as sending a request to the nearest processor with a copy of a desired text. In return, updating the information is very simple and economical, encouraging a fluid situation in which texts may be moved freely to balance loads or adapt to changing access patterns. Moreover, all *essential* operations (determining whether there is more than one copy of a text, reaching all custodians of a text, *etc.*) can still be performed in a straightforward fashion.

### 5.2: Garbage Collection

A reference tree network includes garbage-collected storage as a standard part of the programming environment it supports. Garbage collection on such a network entails the *identification* and *disposal* of objects that will never be used again. When an object becomes inaccessible, it may have become known on several processors. None of these processors can take the initiative to delete the object outright because, in general, none knows whether accessible references to the object exist on other processors. Therefore, it seems that it might be very difficult to ever reclaim the object. If the object is only known on one processor, the story is different. In this case, it is obvious that no references to the object exist on other processors (else the reference tree would be larger) and therefore the object can be deleted if it is not accessible on the one processor where it is known.

Our garbage-collection scheme works by shrinking the reference tree of an object to be collected until only one processor knows about it, at which point the object can be collected by traditional means. In order for a reference tree to shrink, nodes must remove themselves from it. Clearly, any node which has more than one neighbor in the reference tree cannot unilaterally remove itself — if it did, the tree would become partitioned, since those nodes which were originally connected by the removed node would now have no means of communicating. Thus only "leaf" nodes — nodes which have exactly one neighbor also in the reference tree — may disconnect themselves from it.* Fortunately, since reference trees are acyclic, every reference tree has leaf nodes.

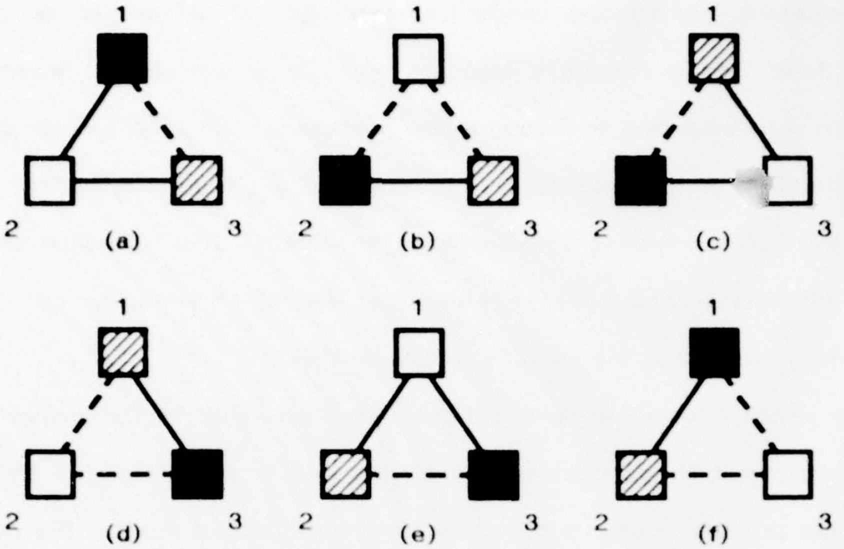
This garbage-collection scheme depends on the fact that a garbage-collectable object will not be used anywhere once it becomes garbage-collectable. Thus, after some interval, processors with references to such an object may guess that it can be collected by the fact that they have not seen it used recently. Even if an object is still potentially accessible, it is inefficient to keep it on processors where it is not needed. Therefore, it is economical for a processor in the reference tree for such an object to remove itself if it can. If that strategy is applied consistently, the reference tree of any garbage-collectable object should slowly shrink

*An additional consideration is that no leaf node which is a custodian of a text for an object may remove itself from that object's reference tree without first preserving the text by passing it on to its neighbor.

to a point (a single node), whereupon the object can be disposed of.

There is one unfortunate problem with this scheme, involving the collection of objects which are part of certain cyclic data structures. For example, consider an object **A** whose text contains a reference to **B**, whose text in turn contains a reference to **A**. Assume further that the structure is garbage-collectable — that neither **A** nor **B** can be reached from any active event. Then by the argument given above, the reference trees for both **A** and **B** should slowly shrink to a point. If both converge to the same point, there is no problem: ordinary garbage-collection techniques can easily handle the situation. However, another scenario is possible, as outlined in Figure 5.7. Here the two objects may spend forever chasing each others' tails, and it may be that *neither* reference tree will ever shrink to a point. The reason this can happen is that when a text is moved from one processor to another it draws with it the reference trees for all objects referenced in that text. Thus when the reference tree for object **A** shrinks and the text of **A** moves from processor **1** to processor **2**, the reference tree for **B** will be extended by the addition of a link from processor **1** to processor **2**. If the reference tree for **B** attempts to contract next by the removal of processor **3**, the text of **B** will have to be sent from **3** to **1**, re-extending the reference tree of **A** to include processor **1**. Of course, there are many other sequences of events, even starting from one of the configurations shown in Figure 5.7, which will result in both reference trees converging on the same point; however, it is possible to have an infinitely long sequence of events which never results in either object being collected.

This problem is similar to the problem of *cyclic restart* in some transaction-based data base management systems[32]; perhaps there are solutions to this garbage-collection problem which are analogous to solutions to the cyclic restart problem. There is reason to believe, however, that the problem will rarely occur in



*Solid lines denote links in the reference tree of object* **A**, *dashed lines the tree for object* **B**. *A solid box represents a processor with a text for* **A**, *a shaded box a processor with a text for* **B**. *Successive reference tree contractions, alternating between the reference trees of* **A** *and* **B**, *can lead to the sequence of situations shown above as (a) through (f), whereupon a final contraction involving* **B** *will restore situation (a).*

Figure 5.7: Cyclic restart in garbage collection

actual operation.

### 5.3: The State Protocol

Reference trees are maintained by means of an interprocessor communication protocol which may be used to grow and shrink reference trees while preserving the required connectedness and freedom from cycles. In order to participate in this protocol, each processor maintains, for each object it knows about, a *link state* for each neighbor (there are thirteen different link states). Various flavors of messages can be sent pertaining to an object's reference tree, and a simple state transition table dictates the responses so as to keep the various link states mutually consistent. By sending appropriate messages, it is possible for a node to delete itself from a reference tree (an option only available to leaf nodes) or extend a reference tree to include a new processor. Typically, a node will attempt to delete itself if it discovers, in the course of a garbage collection, that it no longer has any accessible references to an object. The reference tree for an object will be extended if a text referencing the object is sent to a processor not previously a member of the object's reference tree.

The protocol for accomplishing these things we shall call the *membership protocol* (because it keeps track of membership in reference trees) to distinguish it from other protocols such as that used for moving object texts. The membership protocol, a further evolution of an earlier protocol[15], is more intricate than one might at first imagine necessary, but simpler protocols failed because of deadlocks or inconsistent states reached after inopportune sequences of events. The protocol is described below in what some may find to be daunting detail; the reader who finds the going tedious can at any point skip the remainder of the section and proceed to the next without any loss of continuity. The reader who perseveres, however, will hopefully be rewarded with more than merely an understanding of our particular reference tree management protocol. There is not really *a* reference tree protocol, rather there is a whole *family* of such protocols, a representative member of which is described here. The description below includes an outline of the design considerations which made the current protocol what it is. After completing this section, the reader should be equipped to design his own reference tree protocols to fit his own particular needs, or even, perhaps, improve on the one presented here. The important thing about this description is not its detail, but its illustration of the concept of using link states to maintain a global structure while keeping only local information. The material in this section is presented in a relatively informal style; Appendix B contains a more rigorous argument for the correctness of the membership protocol — that it indeed keeps reference trees connected and prevents cycles from forming in them.

The membership protocol makes no attempt to recover from damaged or lost messages, or messages arriving out of order. These problems can be solved by various well-known means[26] which may be assumed to provide an underlying protocol on each link.

The membership protocol requires that each object have a globally unique name. This name is needed so that when an attempt is made to extend a reference tree to a new processor, that processor can determine whether it already knows of the object via some other route. This information in turn is necessary to detect attempts to form cycles in the reference tree.

The membership protocol involves seven basic kinds of messages, whose meaning and format are as given in Table 5.8. Each message is specialized by identification of the object (and hence reference tree) to which it pertains. This specialization is effected in one of two ways, depending on the message type. Messages which establish new communication paths for an object (commonly by extending the object's reference tree to include another processor) include the object's global name (shown as *GN* in Table 5.8). These messages also include a shorter *local name* (*LN*) which the sender of the message will use in the future for sending references to the object over the same communication link. The other messages in the membership protocol (as well as *all* other messages, including, for example, text management messages) use only the local name to denote the intended object. The use of local names not only shortens messages but speeds their processing. Since the space of local names is smaller and presumably reasonably compact, conversion of an incoming local name to an internal object reference can be accomplished economically by means of a direct table look-up, rather than the more expensive scheme required to look up a global name.

| *Message* | *Meaning* |
|---|---|
| R+ *GN LN* | request to add link to tree |
| L+ *GN LN* | agreement to add link |
| L- *GN LN* | refusal to add link |
| + *LN* | transfer mastery of link |
| - *LN* | request to drop link from tree |
| A+ *LN* | positive acknowledgment |
| A- *LN* | negative acknowledgment |

Table 5.8: Membership protocol message types

Messages in the membership protocol may be sent either spontaneously (*i.e.*, in *response to some external* stimulus, such as the need to have a local name for an object so that a text referencing it can be sent) or in response to an incoming message requesting some change in a reference tree. R+, +, and - messages are always sent spontaneously; *L+, L-, A+, and A- messages* are always sent as responses.

The membership protocol operates by associating with each end of each link a *state* for each possible object. It is these states which actually define the extent of an object's reference tree. In terms of implementation, each processor must maintain a data base for each object it has a reference to, indicating the state (with respect to that object) of each link adjacent to the processor. It is important to realize that the two processors at the ends of a link may have different ideas of the state of the link; this may be as the result of some intentionally introduced asymmetries discussed below, or it may occur if messages regarding the object have been sent at one end of the link but not yet received at the other.

The possible states may be grossly characterized as being either *stable* or *transient*. Stable states are states which might be expected to persist over a relatively long period of time. Transient states are those in which a message has been sent across the link and a reply is expected; the reply will cause a transition to some other state, either stable or transient. Transient states exist to provide the proper sequencing so that the next pair of stable states to be established is consistent and does not result in partitioning the tree or closing a cycle. For purposes of discussion, the states have been given one- to three-character mnemonic names (listed in Table 5.9, below). For purposes of actually manipulating object references passing over links, the most important attribute of each link state is whether a processor in that state is directly prepared to send or receive a local name for the object on that link. Processors in all states but **X**, **N**, and **N?1** are directly able to send local names (*i.e.*, such names have already been declared by R+, L+, or L- messages); processors in all states but **X**, **N**, and **M?** are directly able to receive them (*i.e.*, such names have already been declared to them and recorded).

In addition to the various link states, each processor maintains for each object a *processor state*, either "in the reference tree" or "not in the reference tree." Certain link states are only consistent with a particular processor state. Rather than show the pair (processor state, link state) that governs a processor's response to messages arriving on a link, we encode the processor state information into the link state, adopting the convention of using state names containing the letter "X" to imply a processor state of "not in the reference tree" and other state names to imply the opposite. Thus either all of a processor's link states for a given object will contain X's, or none will. When a link state changes between these categories, other link states in the processor must also change to preserve this consistency. Special transitions (between X and **N**, X? and **N?**), requiring neither the receipt nor the sending of messages, are provided to fulfill this need. In general, each **X** link state has an analogous **N** state, differing only in the processor state of the processor in question.

We now describe the five stable states. Perhaps the state most likely to occur is **X**, which indicates that not only is the link not considered part of the reference tree, the processor is not considered part of the reference tree. If a processor has no knowledge of an object, it acts as though it were in state **X** for that object on every link.

The state analogous to **X** is **N**. In state **N**, the processor is considered part of the reference tree, but still does not believe the link in question to be part of the object's reference tree. State **N** may come about either because the processor is the only processor to contain any references to the object, or because the processor is connected to the reference tree by some other link or links.

Another closely related state is **L**. State **L** is like state **N** in that the relevant link is not considered part of the reference tree, but indicates that the processor at the other end of the link does know about the object, and that local names have been established for communicating references to the object over that link. In a stable condition, the processor at the other end of the link will also be in state **L** with respect to the link. The reason for state **L** is to enable the communication of an object reference over links that cannot be allowed to join the object's reference tree because adding those links would close cycles. Such communication is not only desirable, but sometimes is necessary.

Another stable state is **M**, which indicates that the link in question *is* believed to be part of the reference tree, and furthermore that this processor is currently the master of that link (for transactions involving that object). The master of a link is the only one that can effect changes in the status of the link or send a text of the object over the link. This asymmetry seems to be necessary to prevent confusion resulting from, for instance, both ends of a link simultaneously attempting to terminate their connection with the reference tree.

In a stable condition, the state at the other end of a link from **M** will be **S**, for "slave." A processor in state **S** cannot directly cause a change in the status of the link; it may however (by means of a message not discussed here) request the master to commence a change, and it may respond to messages sent by the master.

The transient states will not be described to the same level of detail as the stable states. For the most part, they acquire their meaning from their relationships with the stable states. Instead of attempting to describe the meaning of these states, we present a state-transition table (Table 5.9), and summarize below the normal sequences for handling several situations.

| | *Transition Upon Receiving* | | | | | | | | *Spontaneous* | |
|---|---|---|---|---|---|---|---|---|---|---|
| *State* | R+ | L+ | L- | + | - | A+ | A- | *LN* | *Transitions* | |
| **X** | L+:**S** | | | | | | | | :**N** | |
| **N** | L-:**L** | | | | | | | | R+:**M?** | :**X** |
| **L** | | | | | A-:**N?1** | A+:**M** | | :**L** | -:**L?** | |
| **M** | | | | | | | | :**M** | -:**X?** | +:**S** |
| **S** | | | | :**M** | A-:**N?1** | | | :**S** | | |
| **X?** | | | | | | | A-:**X** | :**N!** | :**N?** | |
| **N!** | | | | | | | A+:**M?** | :**N!** | | |
| **N?** | | | | | | | A-:**N** | :**N?** | :**X?** | |
| **N?1** | | | | | | L+:**S** | :**N** | :**N?1** | R+:**M?1** | |
| **L?** | | | | | A-:**N?1** | A+:**M** | A-:**N** | :**L?** | | |
| **M?** | :**L** | :**M** | :**L** | | | | | | | |
| **M?1** | | | | | | A+:**S?** | :**M?** | :**M?1** | | |
| **S?** | | | | | :**S?** | :**S** | | :**S?** | | |

*The notation a:b means that under the specified circumstances, a transition to state b can occur with the emission of message a. Transitions occasioned by the receipt of a local name are shown in column* LN.

Table 5.9: Membership protocol state transition table

Although the table indicates which states may *receive* a local name and what state changes may ensue from that eventuality (shown in the column headed *LN*), it does *not show in what states a local name may spontaneously be sent*. As mentioned above, a local name may be sent from any state except **X**, **X?**, **N**, and **N?1**, and never causes a state change in the sender. If a processor in state **N** or **N?1** wishes to send a reference to the object, it must first send an R+ message, which will cause a state change to a state from which a local name may be sent. A processor in state **X** or **X?** has no business sending a reference to the object, since such a processor is not in the reference tree for the object and therefore, presumably, has no references to send.

The fundamental principle that motivates this protocol design (other than the need to maintain the link data base in a consistent state) is that a processor must always be able to send a reference over any link *without prearrangement*. For example, it is not acceptable that the sending of a reference should be the culmination of some transaction allowing the reference to be sent only upon receipt of suitable clearance from the message's target. In order to understand this requirement, the circumstances under which references may be sent must be considered.

In general, a reference will be sent as part of some text which is being communicated between processors. Sending a text involves communicating the reference to the object whose text is being sent, as well as references to other objects referred to in the text. Thus obtaining clearance to send a text may *involve simultaneously obtaining* clearance to send several object references. Unless every processor always has clearance to send any object reference, it is easy to see how the piecemeal aggregation of such clearance could lead to a deadlock on a link. This is especially true when, as is the case with this protocol, transactions involving different objects are completely *independent — no overall* master-slave relationship applies to all communication on a particular link, for example.

This need to avoid deadlock is one of the primary factors acting to complicate the protocol design, and requires that any processor always be able to send any object reference without the possibility of confusing the processor at the other side. The only exception to this requirement occurs if the sending processor is in state **X** — if a processor has no references to an object, it has none to send! In any other state, the processor must either already have a local name for the object to use over the link, *or* have the option of picking a local name, declaring it to its neighbor with an R+ message, and then immediately using it in messages.

We now turn to how and why various state changes may occur. We start with a processor in state **X**, having no references to the object in question. The only kind of message that can be received in state **X** is an R+ message from some processor attempting to extend the reference tree for the object, perhaps in order to send a text mentioning the object. Upon receipt of the R+ message, our processor returns an L+ message as an indication that the link should indeed be added to the tree, and changes to state **S** in anticipation of the sender of the R+ message entering the **M** (master) state when it receives the L+ message. Simultaneously, the states of all other links to our processor change from **X** to **N**, indicating that our processor is now part of the reference tree. Also, any links in state **X?** change to **N?**.

Now that our processor is part of the reference tree, it may attempt to further extend the tree by sending a reference along one of the links just converted to state **N**. *From state **N** an object reference must be preceded by an R+ message.* Upon sending the R+ message, the sender's state for that link changes from **N** to the transient state **M?**, awaiting a reply. The reply to R+ depends on the condition of the processor at the other end of the link. If it was in state **X**, it changes to **S** and replies with L+, as described above. Upon receiving the L+ message, our processor changes from **M?** to **M**, and the link has been established. If the other processor *is in state* **N**, *then the link cannot be added to the reference tree because it* would close a cycle (since both processors are already connected by some other route in the reference tree). Consequently, the other processor responds negatively, with an L− message, and changes to state **L**. When the sender of the R+ message receives the L− message, it also enters state **L**. As long as both processors remain in state **L**, local names have been established for communicating references to the object over the link, even though the link has been agreed not to be in the object's reference tree.

Another possible scenario is that two processors, both in state **N** (for the same link) might simultaneously attempt to add that link to the tree by sending R+ messages to each other and entering state **M?**. Under these circumstances, it is clear that the link should not be added, or a cycle will be formed. Therefore, each **M?** will react to the R+ with a transition to state **L**.

Once it has been agreed that a link *is* part of the reference tree for an object and things have settled to a quiescent state (*i.e.*, no messages are in transit), one processor (the master) will be in state **M** and the other (the slave) in state **S**. It is a simple matter to reverse the roles of master and slave, but the transaction must be initiated by the master. The master sends a + message and enters state **S**. When the slave receives the + message, it enters state **M**.

Having seen how a link may be established in the reference tree, we now come to the question of how a link may be deleted from the tree. Due to the connected, acyclic nature to the tree, every time a link is deleted, a node is also being removed from the tree. Thus the only reason for deleting a link is because a processor wants to remove itself from the reference tree. This in turn will be *caused by that processor's discovery that it has no references to the object* reachable from any active data on that processor. No node which has more than one neighbor in a reference tree can unilaterally remove itself. If it did, the tree would become partitioned, since those nodes which were originally connected by the removed node would now have no means of communicating. Only "leaf" nodes — nodes which have exactly one neighbor also in the reference tree — may disconnect themselves from it. Furthermore, no node may remove itself from a tree leaving dangling (though non-tree) links in state L. (But a link inconveniently in state L may be removed by sending a - message and changing to state L? — the reader can follow the transitions that ensue.) Thus a processor may attempt to remove itself from the tree only if all its links but one are in state **N** (or **N?**). Additionally, that one link must be in state **M**; if the processor is currently a slave on that link (for that object), it must first induce the master of the link to relinquish its mastery.

A master requests to remove itself from the tree by sending a - message to its slave and changing to state X? (simultaneously all **N** links from that processor should change to **X** and all **N?** links to X?). The slave responds with A- and changes to state **N?1**. Upon receiving the A-, the old master returns to state **X**, emitting another A-. When it receives this A-, the old slave goes to state **N** from **N?1**. The extra level of acknowledgment here is needed because a processor in state **S** may send out object references as local names, a capability it must have. The old master must be prevented from returning to state **X**, where such references will not be accepted, until it is confirmed that the old slave is no longer in state **S**. In effect, the A- message sent by the old slave serves to "flush" the channel, bringing up the rear for any local names that might have been sent.

A complication for this scheme occurs precisely when a processor in state **S** sends such a local name to an ex-master now in state **X?**. In this case, the ex-master will once again be in possession of a reference to the object, and must abort its initiative to leave the reference tree. It does this by changing to state **N!** upon receiving the local name. In state **N!**, when the expected A– acknowledgment is received from the old slave, the reply will instead be an A+ message and a transition to **M?**, indicating a desire to remain in the reference tree after all. When the old slave, now in state **N?1**, receives the A+ message, it replies with L+ and returns to state **S**. Receipt of the L+ message by the old master will then cause it to return to state **M**. An L+ message is used here rather than, say, A+, because a processor in state **N?1** does not have a local name it can use immediately to send references over the link. The L+ message serves to re-establish such a local name.

Other transitions in Table 5.9 exist to take care of other pathological occurrences. For example, the old slave, while waiting in state **N?1** for either an A+ or A– reply, may discover that it needs to send out a reference to the object. Since in state **N?1** it has no local name for so doing, it must declare one by sending an R+ message, which is accompanied by a transition to **M?1**. The reader can follow the sequence of transitions and replies triggered by this R+ message, and see some more of the entries in Table 5.9 come into play.

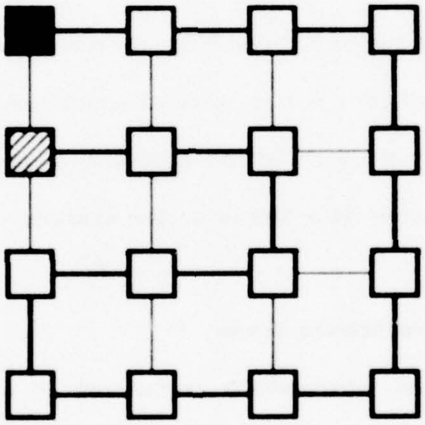
Cases like this are another source of complication in the membership protocol. Generally speaking, whenever a processor can undergo a spontaneous transition to another state, the new state must be able to respond meaningfully to any message the old state might have been expecting. When both processors at the ends of a link are susceptible to spontaneous transitions, adding one new function to the protocol may require the addition of several new transitions.

### 5.4: Modifications to the Reference Tree Concept

This section is devoted to a discussion of several potential problems with the reference tree concept, along with some possible solutions to them.

### 5.4.1: Reference Tree Management

Use of reference trees can lead to two kinds of inefficiencies. Both are suggested by the reference tree depicted in Figure 5.10.



*The solid box represents a processor with a copy of the object's text; the shaded box represents a processor inquiring for the text.*

Figure 5.10: A non-optimal reference tree

One liability is that reference trees may not always grow in the most desirable shapes. In Figure 5.10, the shaded processor's inquiry and the solid processor's response will both have to travel the entire length of the reference tree, when in fact the processors are adjacent. The other liability, which can also occur in more "stretched-out" reference trees, is the overhead involved in keeping reference trees connected. Even if the two end processors in Figure 5.10 are the only two which continue to have any interest in the object, all the intermediate processors must still stay in the object's reference tree to keep it connected. In a large system with many objects, such extended reference trees could impose significant overhead on each processor. The amount of overhead involved in membership in a reference tree is not large, but is *non-zero*. As a network grew, it would be possible, especially given an unfortunate event and object distribution strategy, for the average size of reference trees to increase to the point where each processor found itself forced to keep track of more and more objects. Conceivably the network could be slowly strangled as more and more of its storage was devoted to these reference tree "cobwebs." Even more disturbing is the prospect that a similar fate would befall a network, not because of any attempt to scale up the number of processors, but simply because of the accretion of objects over time, as users come to have more and more data stored on the system.

### 5.4.1.1: Disconnecting Reference Trees

Overhead imposed on intermediate processors in a reference tree by the requirement that the tree be kept connected might be eliminated if it were permissible to disconnect pieces of a reference tree. It is not difficult to devise a protocol by which one of the intermediate processors could cause this to happen. Since all communication involving an object travels only along links in its reference tree, however, two requirements must be met: a custodian of a copy of the text of the object must exist on either side of the break (otherwise the processors in one of the disconnected pieces would have no access to the text of the object), and the object must be immutable (otherwise an update performed in one half of the tree would never become visible in the other half). Generally speaking, it is not possible to tell whether a side effect may in the future be performed on an object, limiting the applicability of this approach. However, if such information were supplied, say in the form of a "read-only" bit, reference trees of "read-only" objects could be disconnected.

Effectively, breaking a link in the reference tree for an object creates two reference trees for the object. Each of the new trees will then behave as if it were the only reference tree for that object. Specifically, leaf nodes of either tree may then delete themselves from it, so all the intermediate processors in our example can leave the tree, one by one, resulting in the desired situation where only the two distant processors know about the object.

In fact, the only problem with disconnecting a reference tree arises if it is ever desired to re-connect the tree. The reference tree management protocol avoids cycles in reference trees by refusing to make a connection if two branches of reference tree for the same object bump into each other. This is done because it is assumed that all branches of reference tree for the same object are already connected; therefore, adding another connection would close a cycle. If, as the result of a disconnection, the two branches are not connected, the protocol will still refuse to connect them. It would be quite difficult to allow such branches to be re-connected without introducing the possibility that cycles could be formed. Thus it seems that once a reference tree is broken into two or more pieces, those pieces must continue to exist independently for as long as they continue to exist. This is not necessarily bad, though. Each piece is still free to grow, shrink, and move just as the original was, and thus each may independently be reclaimed by the garbage collection mechanism when its usefulness is ended.

### 5.4.1.2: Reorganizing Reference Trees

Figure 5.10 gives an example of a non-optimal reference tree which could be improved by being rerouted. Any approach to rerouting based on purely local knowledge of the reference tree should fit easily into our scheme, provided it preserves the essential properties of reference trees (connectedness and freedom from cycles).

One possibility is for a leaf node which is aware that one of its neighbors is connected to the tree by a different path (perhaps because of being in state L with respect to that neighbor) to break its old connection and connect instead to that neighbor. This kind of operation is depicted in Figure 5.11.

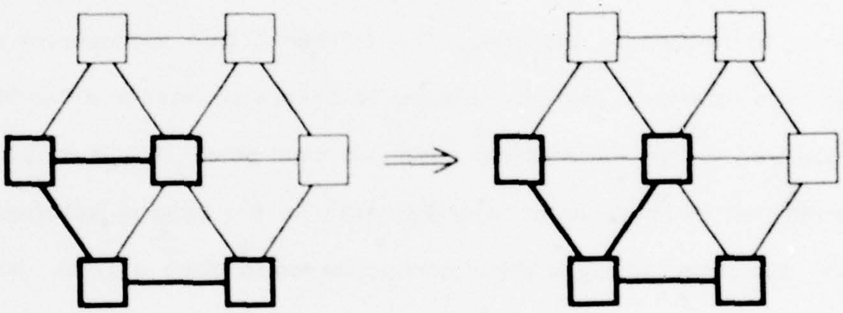


Figure 5.11: A simple reference tree reorganization

It is difficult, unfortunately, for a non-leaf node to make this kind of jump, because it will not know which of its old links to break. If the wrong choice is made, not only will the reference tree become disconnected, but one half of it will contain a cycle, as shown in Figure 5.12.

In addition to the mechanics of reorganizing the tree, there is of course a strategy question—when is reorganization wise? Once again, in simple cases the answer can be fairly obvious, but in general it may not be. If the goal for the processor changing its links is to get closer to a copy of the text of the object, then it

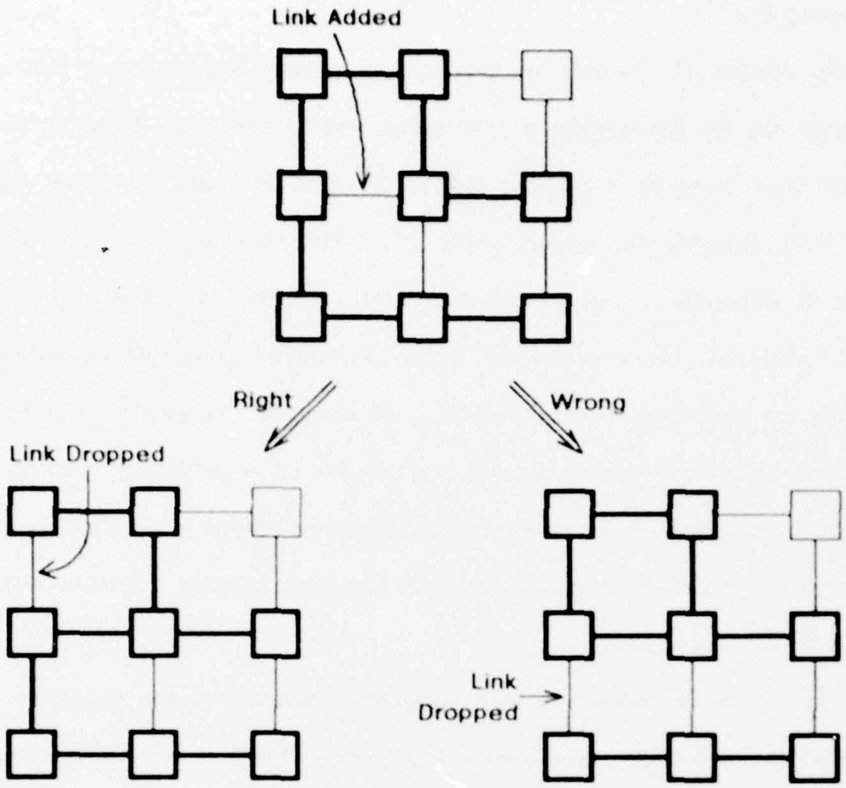


Figure 5.12: A dangerous reference tree reorganization

is obviously a good idea to reorganize if the processor being connected to has a copy of the text. If it does not have a copy of the text, then it will either have to have some idea how far the nearest text is (a piece of information that might become obsolete every time a text moved) or other considerations will have to be invoked.

### 5.4.2: Reliability

As the number of components in a system grows, the likelihood that all these components will be operational at the same time decreases. Although much of a reference tree network can continue to function in spite of some component failures, such failures may nevertheless affect the network more seriously than is desirable or necessary. For a moderate-sized network, this need not be a major concern; after all, there are many large centralized computer installations with essentially no resiliency against failures, at least in the central processor, and their overall reliability record remains acceptable. If a reference tree network is viewed as a replacement for one of these machines, there is no *a priori* reason to believe that a reference tree network with the same number of components should be any more prone to failure.

One attraction of reference tree networks, however, is the possibility of scaling them up to very large sizes, where the failure of one or more components might be a common occurrence. Furthermore, if reference tree networks have the potential for enhanced reliability through appropriate design changes, it is a shame not to take advantage of this possible benefit, even in networks of modest size. Finally, it may often be desirable to *plan* "failures" of various components to take them temporarily out of circulation for preventive maintenance or reconfiguration. It turns out that even such planned shutdowns are difficult to manage with the basic reference tree scheme.

It is useful to categorize failures as either failures of nodes or failures of links connecting nodes. The basic problem that arises when a link fails is that all reference trees going through that link become partitioned — operations on objects whose reference trees do *not* include that link will not be affected. The obvious solution to this problem is to devise a protocol for rerouting the affected reference trees through other, still-functioning links, without making the reference tree cyclic or otherwise leaving the reference tree information in an inconsistent state. A scheme for doing this, as well as dealing with node failures, is the subject of thesis research by Clark Baker[2].

Node failures are more serious than link failures. Not only does a node failure look like a failure of all links leading to that node, but the integrity of, or at least access to, data stored at that node may be compromised. If the only copy of an object's text is stored at a failed node, and that copy is destroyed in a failure, it is difficult to see how to recover it. If that problem is to be solved, it must be solved either by redundant storage of object texts, or some higher-level mechanism for regenerating lost object texts, or both. Also serious is the problem of what to do with events that may have been stored on the failed processor, and how to tell, if such events vanish, whether they vanished before or after being executed. These subjects, too, are being studied by Clark Baker, but it remains to be seen how the cost of avoiding these problems will trade off against the improvement in reliability obtained thereby.

### 5.4.3: Global Names

The network protocols as currently envisioned require that each object have a unique global name, assigned when the object is created, and unchanged thereafter. This global name is used in determining whether references that arrive at a processor by different routes actually refer to the same object. This testing of objects for identity is an operation which the user himself may well wish to perform, and is also necessary if cycles in reference trees are to be avoided.

Since objects are created asynchronously at several nodes in the network, the problem arises of making sure that all these nodes deal out unique global names.

The simplest solution is to partition the set of possible global names among the processors, for example by having each processor insert a unique identifying string into the global names of the objects it creates. This solution suffers from a minor lack of expansibility, since the space of unique processor ID's must be large enough to accommodate all processors which might be added to the system in the future. More seriously, after some period of time, a processor will run through all the global names allotted to it. Even if it were to "borrow" some yet-to-be-used names from other processors, after a long enough interval all global names will have been used. The problem may be solved for practical purposes by allowing for a name space large enough to last for a very long time, but it can also be solved by "recycling" the global names of objects which are garbage-collected, reusing those names for new objects. Then the global name space need only be as large as the total number of objects that can exist in the system at one time — a bound that is difficult to improve on as long as each object requires a global name.

Aside from these complications, there are other reasons for disliking global names. For example, they make it difficult to interconnect systems that previously had been operating independently. It is possible, in fact, to devise a reference tree scheme which uses no global names. Such a reference tree scheme cannot easily prevent cycles in reference trees, but the kind of "cycles" that form are not fatal. They resemble helices more than cycles, with successive coils of a helix falling on top of each other. What prevents such cycles from being harmful is that each time a coil of a helix passes through a processor, it appears as a different reference. Only by initiating some kind of "trace" operation can a processor discover whether two such references actually refer to the same object. This possibility that an object might manifest itself as two superficially different references, or, put another way, that two apparently distinct references might be found to refer to the same object, puts a new wrinkle into the semantics of VIM.* Without this mechanism, it can be determined if two references refer to the same object by simply checking if both are actually the same reference, a relatively inexpensive operation. While some may regard this checking of objects for identity as a "dirty" operation, it is often a useful one. For example, the implementation of a LISP-like language will be a good deal more efficient if the equivalent of the LISP EQ test[25] can be performed by simply comparing references. The generation of unique objects to serve as *keys* (as in Henderson[17]) and their later comparison is another example of a situation where being able to check objects for identity is useful.

In conclusion, although our scheme for doing without global names has its attractions, it harbors possible inefficiencies, as the reference tree for just one object can grow without bound. Only experimentation can show whether this kind of helical reference tree would pose a serious problem in practice.

### 5.4.4: Non-Homogeneous Networks

So far, we have been assuming that reference tree networks are homogeneous: although nodes may exhibit various differences in capacity and features, all use substantially the same internal representations of data and algorithms. This is an entirely reasonable approach for many uses of reference tree networks, especially with increasing standardization of computing hardware around conventions such as 8-bit bytes and 16-bit words. However, because of special features available or advancing technology, it may become desirable to include various nonconforming processors into a network. Such processors could probably handle

*The "link" mechanism proposed by Gula[14] adds a similar wrinkle.

differences in data representation, at least within limits, by automatic translations of incoming and outgoing object texts. However, it might be difficult to translate texts which contain executable code this way. A possible solution here is to maintain several *versions* of such texts, so that a processor could call for the version of a text which suited its need. In fact, if executable texts were originally expressed in some higher-level language, that higher-level version might be retained as well, and new machine-level versions generated upon demand.

### 5.5: Summary

The reference tree mechanisms described in this chapter provide a complete set of capabilities for managing objects in a network. The protocols outlined allow for side effects to be performed on objects, for the maintenance of multiple copies of objects, and for garbage collection of inaccessible objects, even when these have become known across several processors.

A primary objective of the reference tree design is to permit an extremely flexible view of event and object text locations. By keeping down the overhead involved in moving objects and events around, frequent readjustment of their locations is encouraged. One sense in which this overhead is low is that all updates to the reference tree data base are strictly local and asynchronous. Since increases in total system size have no effect on the overhead of any particular reference tree transaction, the use of reference trees is compatible with building systems that can be scaled up to arbitrarily large sizes.

The only nonlocal aspects of reference tree management are the use of global names for objects and the use of globally unique time stamps for conflict resolution. The cost of storing these names and time stamps increases logarithmically in the size of the network. This increase in cost is the only crimp in the unlimited scalability of reference tree networks. Section 5.4.3 suggests a way of doing without global names, at some cost; perhaps an analogous scheme exists for resolving conflicts without recourse to globally unique time stamps.

Beyond these limits on the scalability of reference tree networks, the reference tree mechanisms are, at least in theory, prey to various kinds of inefficiencies and unreliabilities. There is reasonable hope, however, that methods proposed in this chapter, and others like them, can conquer these problems well enough to make reference trees truly practical on a large scale.

# Chapter 6: Performance of Reference Tree Networks

Up to this point, our primary concern has been with the architectural details of reference tree networks: ensuring that necessary facilities are present, and that they will operate correctly. Especially since a principal motivation of reference tree networks is economic, however, we cannot be satisfied with a design simply because it allows the expression of useful algorithms and does not make mistakes in interpreting them. Accordingly, this chapter is devoted to understanding performance aspects of reference tree networks. We begin by proposing a simple model of reference tree networks, designed to highlight variables that most directly affect performance. The meanings of various measures of performance, in the context of this model, are then discussed. Finally, conclusions obtained by applying the model to a variety of hypothetical situations are presented.

## 6.1: Models of Network Elements

It is logical, when modeling something, to mimic its gross structure. In our model of reference tree network hardware, we copy the overall topology of the network, substituting idealized models for processors and links. Our task is thus reduced to that of constructing suitable idealizations of these elements. What we would like to do is characterize the resources available in each kind of network element and the effect of each network activity on the consumption of each resource.

For a processor, the resources available to be consumed are CPU time and memory space. Memory space is occupied by events and by object texts (texts containing both data and algorithms). CPU time is needed for execution of events. CPU time is also used for the encoding of events and object texts preparatory to sending them across a link to another processor, as well as for the corresponding decoding of incoming events and object texts. Other sinks of CPU time exist also, such as garbage collection and interrupt processing load. Since the relationship of these loads to the primary activities of the processor is often indirect and unclear, we do not represent these loads explicitly in the model. To the extent that they are constant, these loads can be accounted for by reducing the total CPU time available for event execution and message processing. To the extent that the loads vary with the frequency of any activity that *is* represented in the model, the time taken can be added in as "overhead" to the time required by that activity. To the extent that other factors cause these loads to vary, the model will be inaccurate.

The model for links is influenced by an expectation that most reference tree networks will exist within a fairly localized area, so that any delays associated with message traffic will be due primarily to bandwidth limitations and not to physical delay between one end of the message channel and the other. Accordingly, the only resource available in a link is "message time," which is used up by messages (*i.e.*, shipment of events and object texts) in proportion to their length. (We concentrate exclusively on transmission of events and object texts, treating membership protocol messages, inquiries, and the like as overhead for links, just as garbage collection and interrupt processing are treated as overhead for CPU's. This simplification is generally in accord with the relative sizes and frequencies of different kinds of messages on the MuNet.) However, the link has no delay, in the sense that the moment a message begins to be sent at one end, it begins to be received at the other, and the moment the last bit has finished being sent at one end, it has finished being received at the other. Thus a link resembles an orifice, or narrow bottleneck, rather more than a pipeline. This is a fairly accurate mathematical representation of communication over short distances using a parallel or serial line. This delayless assumption will be relevant in some applications of the model, but for others the results would not be affected even if the interprocessor messages suffered delay as well as a bandwidth limitation.

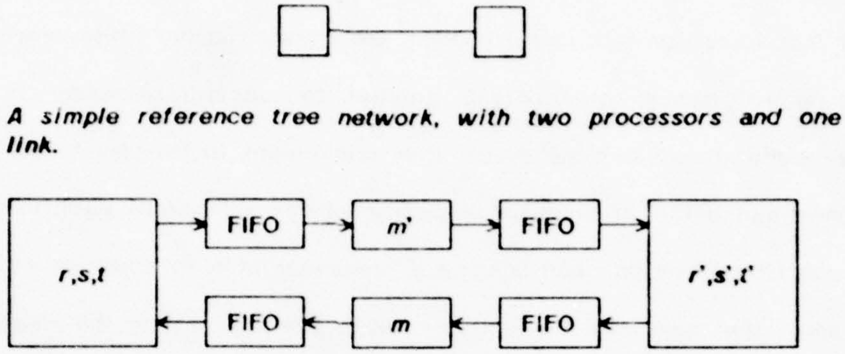
Our first excursion into model building will ignore memory limitations of processors, along with space constraints in general, to concentrate solely on time constraints. Under these circumstances, it is convenient to imagine a first-in-first-out (FIFO) message buffer of unlimited capacity interposed before each link and also after each link. Then one can imagine a processor able to compose several messages and dump them into the FIFO preceding the link, letting the messages percolate at their own pace through the link into its following FIFO. The receiving processor can then look at the messages as they come in, or allow them to accumulate for a while in the FIFO.* Assuming these FIFO's, which may be thought of as message buffers within the processors (but accessed directly by the links), simplifies application of the model to situations where the ability of a processor to keep a message channel full, by initiating a new message as soon as the channel becomes free, might otherwise be in question.

Our model of links is unidirectional, whereas actual reference tree links are required to be bidirectional. A bidirectional link, of course, may be constructed out of two unidirectional links running in opposite directions, but our unidirectional model is useful in examining situations where we wish to constrain the flow of events and

*This is a reasonably accurate representation of what actually goes on in the MuNet.

object texts over a link to be in only one direction.[†]

The final model is perhaps best illustrated by Figure 6.1, which shows our model of a network composed of two processors and one link connecting them.



*(a) A simple reference tree network, with two processors and one link.*

*(b) A model of the network shown in (a), with performance parameters* r, s, t, r', s', t', m, *and* m'.

Figure 6.1: The reference tree network model

Inside the active elements of the model are shown performance parameters. The parameter *m* (or *m'*) of a link may be thought of as the number of time units taken by that link to transmit one word of message. The parameter *r* (or *r'*) of a processor is the number of units of CPU time required per word of incoming (or *r*eceived) message; similarly, *s* (*s'*) gives the amount of CPU time used per word of message sent. Finally, *t* (*t'*) indicates the speed of the processor for event execution. *t* may be thought of as the amount of CPU time consumed per unit of computation during event execution. The primed parameters are shown to emphasize that every element of the network may, in general, have different sets of parameters, indicative of their differing capacities to perform the various operations that may be

[†] Even if such a constraint were adopted in a real reference tree network, however, the reverse channel would still be needed for reference tree protocol messages.

required of them. The parameters *r*, *s*, *t*, *m*, *etc.*, can be further generalized in terms of concepts presented in the next section.

A formal description of a network topology may be given in terms of a set **N** of nodes, a set L of links, and two Boolean matrices *I* and *O*. An element $I_{jm}$ is 1 if and only if link $j \in$ L enters node $m \in$ **N**, else is 0. Similarly, an element $O_{jm}$ is 1 if and only if link *j* leaves node *m*.

### 6.2: A Model of Computations

Constructing a plausible model of the network hardware is trivial in comparison to the task of modeling the computations the network is to perform. Many computations of real practical interest have anatomies complex enough to defy any mathematical modeling approach other than simulation. Among the computations that can be usefully modeled, the universe of alternatives is too diverse to be encompassed under any one modeling procedure. We are forced, therefore, to commit ourselves, at least temporarily, to studying computations belonging to some fairly specific class.

The class of computations on which we choose to concentrate is not the most interesting class of possible computations for a reference tree network, but it has the virtue of being fairly simple. Furthermore, it can be extended sufficiently so that it is at least plausible to argue that the scheduling strategy which performs the best on computations in this class will also be among the best performers in actual practice, even when applied to computations outside the class. In other words, significant further improvement in network efficiency may only be achievable by adopting radically different approaches, such as making available to the scheduling algorithm additional precompiled information indicating the best treatment for each specific computation.

This class of computations to be modeled is a class of recurring computations where each computation has fairly simple structure. "Recurring" is intended to convey the notion of a series of sets of input values arriving at periodic intervals for processing. There is little or no interaction between the processing of one set of input values and the processing of other sets of values, and the processing of each set of values proceeds along a similar course. An example of such a recurrent computation would be a system receiving three-dimensional vectors and performing a perspective transformation and windowing to yield a sequence of two-dimensional vectors. One could imagine a steady source of three-dimensional vectors, contributing either to continued drawing of a picture or to successive frames of a movie. In either case, the processing undergone by each vector follows the same outline, or at worst has a few variants according to whether or not the vector turns out to be visible.

This concentration on recurring tasks is convenient for studying the ultimate performance capabilities of a network topology, since any amount of parallelism that might be needed in order to use the network resources to their full capacity can be obtained by simply adding more concurrent tasks. This approach carefully sidesteps any consideration of how much parallelism can be used in the solution of any particular task. In many other situations this constraint of the task may very well be a much more significant constraint on performance than saturation of the network elements with work. Nevertheless, our interest here is in the overheads and limitations involved in the use of network resources, as opposed to the plausibility of actually bringing all those resources into play, and thus we proceed with our recurrent-computation model.

Given a recurring stream of input data, the question becomes how to model the computation applied to each element of the stream. This is done by means of *events* and *transitions*. An *event* represents a computation in a certain intermediate state, and corresponds to a VIM event. A *transition* represents the process of swallowing up an event and producing zero or more other events. It thus corresponds to an event execution under VIM. An event is something which can be passed from one processor to another, or held for future execution. The important parameter of an event is its size, which determines the storage required to hold the event, and the cost of shipping it between processors.* A transition has two basic parameters: the time required to effect, or execute, the transition, and the space used to represent the algorithm which implements the transition. Of these parameters, execution time is the more significant for the purposes of this chapter.

In the model, events fall into some small number of *event classes* containing events with the same, or at least statistically similar, properties. Similarly, transitions may be grouped into *transition classes*. Two of the event classes are distinguished as the *initial event class* and the *final event class*. Receipt of a set of values to be processed is modeled by the receipt of an event in the initial event class, containing these values, at some processor; output of a set of result values is represented by the sending out of a corresponding event in the final event class.

More formally, then, our model recognizes a set E of event classes. Each event class has an associated size which is characteristic of events in that class. One event class is distinguished as the initial event class and another as the final event class. The model also recognizes a set T of transition classes, where each transition class has an execution time and algorithm size pertaining to transitions in

*It is assumed that *all* data required for execution of an event is accounted for in calculating the "size" of the event.

that class. There is additionally an E×T matrix $G$, such that $G_{ik}$ gives the number of events of class $i$ consumed (if $G_{ik} < 0$) or *generated* by a transition of class $k$. Ordinarily, there would be exactly one element $G_{ik}$ less than zero for each $k$, and that $G_{ik}$ would equal $-1$, indicating the transition (of class $k$) consumes one event of class $i$. Other elements $G_{jk}$, $j \neq i$, would be either zero, indicating no involvement by the transition with events of class $j$, or positive integers, indicating production by the transition of $G_{jk}$ events of class $j$. However, none of these assumptions enters into the mathematics below, and it can be meaningful to depart from them.

Using the concept of event and transition classes, additional network performance parameters can be described in terms of matrices $M$, $R$, $S$, and $T$. $M_{ij}$ is the time taken by an event of class $i$ to traverse link $j$. $R_{im}$ is the CPU time required to receive an event of class $i$ on node $m$. Similarly, $S_{im}$ is the CPU time required to send an event of class $i$ from node $m$. Finally, $T_{km}$ is the CPU time required to process a transition of class $k$ on node $m$. These matrices may often have a fairly simple structure; for example, all entries $M_{ij}$ might have the same value $m$. But using these matrices allows the representation of a great deal of diversity among the various elements of a network, including variations in the capacity of the elements, and even variations in the relative capacity of elements for operations of different kinds.

### 6.3: Measures of Performance

The purpose of engaging in this modeling effort is to obtain a simplified abstraction in which the performance effects of scheduling strategies can be judged. Before performance can be measured and compared, though, it must be defined. The plausible measures of performance are varied and sometimes antagonistic, but all are parameters of particular *execution histories*, not of particular scheduling *algorithms*. A scheduling algorithm, especially if it contains any random or pseudorandom component, may exhibit different performance on successive runs, and will almost certainly have different performance characteristics when applied to different programs. Thus our approach to evaluating a scheduling algorithm must be to apply the algorithm to some program, then examine the resulting execution history to determine the performance. This approach to network performance measurement pays one additional dividend: it is possible to define *optimum*, or *maximum*, performance as the maximum value of the performance measure over all possible execution histories for the program at hand, without respect to what, if any, scheduling algorithm could actually have produced the winning history. Then it can be seen how far the performance achieved by any particular scheduling algorithm falls short of the optimum. This may give some indication of how worthwhile it is to complicate a scheduling algorithm in hopes of coming closer to the optimum performance level.

Several plausible performance criteria may be derived from examination of an execution history:

- *minimum resource usage.* The theory behind this criterion is that the fewer system resources (processor time and space, link time) used in executing a particular program, the more are left over for other uses (unspecified in the model). Alternatively, one may suppose that a scheduling algorithm which is frugal in its use of resources exhibits greater potential to be applied to more demanding problems without exhausting the resources available. Unfortunately, incomplete use of available resources may impact adversely on other plausible measures of performance, such as response time. This consideration leads us, paradoxically, to our next suggested performance measure.

- *maximum resource usage.* This criterion may be defended on the grounds that large degrees of resource usage are evidence that an effort has been made to derive the maximum possible benefit from every network resource. Unfortunately, high resource usage does not necessarily correspond to productive resource usage; it is entirely possible to consume resources in useless churning. Furthermore, maximum resource usage is not desirable in itself. Rather, it is conjectured to accompany execution histories with other desirable properties. A more direct approach, then, is to characterize those properties directly.

- *minimum response time.* A parameter that has more to do with the actually observed input/output behavior of a system is response time, but in the context of recurring computations its meaning needs to be defined more precisely. When an event arrives at the system and passes through several transitions, ultimately one or more output events, specifically traceable to that input event, may be caused. The response time for an individual input event in an execution history is then the length of time that elapses from the introduction of the input event into the system until the last associated output event has been emitted from the system. If it is possible for some input events to lead to *no* output events, response time will be a less useful measure of performance, since it must be defined arbitrarily in such cases. Since a particular response time applies only to a specific event in an execution history, it is possible to define both an *average* and a *worst-case* response time over an entire history. Knowing the response-time results of applying a scheduling algorithm to a problem is indeed useful, but response characteristics are difficult to model analytically. This is because of their significant dependence on timing relationships that develop accidentally during execution, and on the specific way in which processors handle tasks (for example, whether one task is allowed to pre-empt another). For these reasons, response time can probably be determined only by simulation or actual execution, and the optimum possible response time may be very difficult to determine at all.

- *minimum "event inventory"*. An alternative parameter, closely related to average response time, but which may be somewhat more tractable, is "event inventory," the number of events that have been introduced into the system and not yet discharged (in the sense, discussed above, of all associated output events having been emitted). As before, one can talk about average or worst-case event inventory. Neither measure bears any relation to worst-case response time, since events are not necessarily discharged in the order of their introduction, but the *average* event inventory bears a simple algebraic relationship to the *average* response time. Although it still resists exact treatment, the event inventory approach may facilitate the derivation of good approximations.

- *throughput.* We may define the throughput of an execution history as the rate at which input events are discharged by the system. Unless there is to be an unbounded buildup of events within the system, this rate will over the long term be equal to the rate at which input events are introduced into the system. Cases where unbounded buildups of events occur represent overloaded situations to which the system's transient response may be interesting, but which cannot persist over the long term. Since, in our model, the rate at which input events are introduced is an independent variable, examination of a particular execution history to determine throughput can only yield a "yes-no" answer: yes, the system exhibited adequate throughput to prevent an unbounded buildup of events, or no, it did not. It is, however, possible to determine from several execution histories generated using a particular scheduling algorithm and different event input rates the maximum throughput allowed by that algorithm.* What makes throughput an especially interesting standard of comparison is that it is also possible to determine, for computations represented by our model, the maximum throughput of which a particular hardware configuration is capable, irrespective of the scheduling algorithm used.

*Of course, some algorithms may not be able to meet certain requested throughputs, even while being able to meet higher ones. In such cases, the definition of "maximum throughput" admits of some ambiguity, but such behavior is anomalous, or at least undesirable, and is best described in terms of the actual critical throughputs observed, not by attempts to define any one "maximum."

On the basis of the considerations discussed above, we select throughput as the primary measure of performance to use in comparisons. Not only is it analytically the most convenient, but it captures well the spirit of our broader endeavor to construct computer systems of higher capacity. The scheduling algorithm with the highest maximum throughput may not exhibit the best response time at lower throughputs, but no other algorithm can match it when throughput is at that maximum. High throughput is also a sign that inefficient use of resources due to wasted motion is being kept at a minimum.

### 6.4: Calculating Maximum Possible Throughput

The maximum possible throughput of a network is the maximum throughput of which that network is capable, regardless of the scheduling algorithm used. For computations representable in our model, it may be calculated, for a specific network and a specific type of computation, by a linear programming procedure. Numerical results may be obtained in this fashion, but except in the most elementary cases, it is unfortunately impossible to give simple analytical formulae.

The linear programming approach is based on a kind of network flow model which considers the long-term average frequencies of event transmissions over links and through nodes, and of occurrences of transitions at nodes. The maximum attainable frequencies are constrained by the capacities of the various network elements.

In more detail, the linear programming model contains one variable for

- each event class for each link, indicating the frequency of transmissions of events of that class over that link; we denote these by $e_{ij}$, for event class $i$ and link $j$.

- each transition class for each node, indicating the frequency of execution of transitions of that class on that processor; these are written as $t_{km}$, where the transition class is $k$ and the node is $m$.

In the following discussion, we use these and several other variable names without further explanation. For a summary of our nomenclature, refer to Table 6.2.

| | |
|---|---|
| **E** | the set of event classes |
| $i$ | a member of **E** |
| $i_0$ | the initial event class |
| $i_1$ | the final event class |
| **T** | the set of transition classes |
| $k$ | an element of **T** |
| **L** | the set of links in the network |
| $j$ | an element of **L** |
| $\mathbf{L}_0$ | the set of input links: $\mathbf{L}_0 \subseteq \mathbf{L}$ |
| $\mathbf{L}_1$ | the set of output links: $\mathbf{L}_1 \subseteq \mathbf{L}$ |
| **N** | the set of nodes in the network |
| $m$ | an element of **N** |
| $e_{ij}$ | the flow of events of class $i$ over link $j$ |
| $t_{km}$ | the frequency of transitions of class $k$ on node $m$ |
| $O_{jm}$ | 1 iff link $j$ leaves node $m$, else 0 |
| $I_{jm}$ | 1 iff link $j$ enters node $m$, else 0 |
| $G_{ik}$ | the number of events of class $i$ generated by transition $k$ |
| $M_{ij}$ | time taken by an event of class $i$ traversing link $j$ |
| $R_{im}$ | CPU time to receive an event of class $i$ on node $m$ |
| $S_{im}$ | CPU time to send an event of class $i$ from node $m$ |
| $T_{km}$ | CPU time for a transition of class $k$ on node $m$ |
| $C_m$ | memory size at node $m$ |
| $P_k$ | size of algorithm implementing transition of class $k$ |

Table 6.2: Network Modeling Nomenclature

The number of actually independent variables is reduced by several identities relating them. For each node $m$ and each event class $i$, there is a conservationof-events condition

$$\sum_{j}(I_{jm}-O_{jm})e_{ij}+\sum_{k}G_{ik}t_{km}=0 \qquad (6.1)$$

Furthermore, for purposes of introducing events of the initial class $i_0$ and removing events of the final class $i_1$, we define a set of in-links $L_0$ whose sources are indeterminate but whose destinations are nodes, and a corresponding set $L_1$ of out-links. Then we have the additional constraints that only events of the initial class may flow over in-links:

$$e_{ij}=0,\ i\neq i_0 \text{ and } j\in L_0 \qquad (6.2)$$

and only events of the final class may flow over output links:

$$e_{ij}=0,\ i\neq i_1 \text{ and } j\in L_1 \qquad (6.3)$$

The constraints on the linear programming model are, first of all, that all variables must be non-negative. Events cannot "un-flow," nor transitions "un-execute." Secondly, event flow over any link $j$ cannot exceed the capacity of the link:

$$\sum_{i}M_{ij}e_{ij}\leq 1 \qquad (6.4)$$

and computational activity (this includes execution of transitions as well as processing of messages) on any node $m$ must not exceed its capacity:

$$\sum_{i}\sum_{j}(I_{jm}R_{im}+O_{jm}S_{im})e_{ij}+\sum_{k}T_{km}t_{km}\leq 1 \qquad (6.5)$$

Finally, subject to the above identities and constraints, we wish to maximize

$$\sum_{j\in L_0}e_{i_0 j} \qquad (6.6)$$

which is the inflow of events of the initial class which can be maintained over the long term without overloading any part of the network. The values of variables which maximize (6.6) indicate optimum distributions of activities around the network.

### 6.5: Memory Constraints

The model developed in the previous section, although it will be our principal source of results concerning the theoretical capacity of a network, deals solely with temporal constraints on the operation of the network. In a real network, spatial constraints may very well also play a role. In computations fitting our model, space may be used in two ways: for storage of events, both while being held at nodes and while stored in buffers at ends of links, and for storage of the algorithms that implement the transitions. (Reference tree overhead and temporary storage required during actual execution of a transition are taken into overhead.)

The linear programming model developed above, by being insensitive to spatial constraints, really embodies the assumption that sufficient space is available at every node to buffer all events that may need to be buffered there, and to hold algorithms for all transitions that may need to be executed there. Since the model concentrates on long-term flows, it is very difficult to determine anything about buffering requirements even from examination of a solution. Buffering requirements are more a property of the microstructure of operations on each processor, a variable deliberately abstracted out in the course of building our model. In many cases, though, events may be assumed to be relatively small, and the buffering requirements relatively modest, or at least relatively unaffected by changes in event distribution. In such cases, the storage of algorithms may be the main contributor to memory use.

A simple assumption about memory use is that if a transition is executed at all on a processor, however infrequently, a copy of the corresponding algorithm must reside on that processor. Then the memory used at node $m$ is

$$\sum_k P_k u_1(t_{km}) \qquad (6.7)$$

where $P_k$ is the size of the algorithm implementing transitions of class $k$, and $u_1(x)$

is the unit step function

$$u_1(x) = \begin{cases} 0 \text{ if } x \leq 0 \\ 1 \text{ if } x > 0 \end{cases} \tag{6.8}$$

Note that expression (6.7) is nonlinear in $t_{km}$ and hence cannot be used in either a constraint or an optimality criterion in linear programming. Be that as it may, a more serious question is how memory usage ought to be accounted for in any model, linear or not. Should some weighted average of throughput and memory usage be used as the optimality criterion, rather than throughput alone? But what is the point, given sufficient memory in the system, of settling for lower throughput simply because it makes less intensive use of memory the system already has? One justification for so doing is the existence in the system of activities other than that being modeled. If the computation being modeled takes up less space, then more space is free for other uses. But why account for the spatial impact of other activities when no such accounting is made of their temporal impact? On the whole, it is of some interest to observe the memory requirements of the optimal (maximum throughput) solution or, if there are several, the optimal solution requiring the least memory, but it is difficult to justify proclaiming the superiority of a lower-throughput solution simply on the grounds that it uses less memory.

A different tack is to treat memory usage not as part of any optimality criterion, but as a constraint on possible solutions. Borrowing from equation (6.7), a new constraint of the form

$$\sum_k P_k u_1(t_{km}) \leq C_m \tag{6.9}$$

could be added for each node $m$, where $C_m$ is the memory capacity at node $m$. The resulting problem is no longer a linear programming program, but may still be solved by combinatorial means. This approach at least begins to deal with the reductions in network capacity that may be forced by space constraints on the distribution of algorithms (we continue to ignore space requirements for buffering). Such space constraints will only be significant, however, if the size of algorithms is of the same order of magnitude as the memory capacities of processors. For computations accurately representable by our simple model, this is unlikely to be the case. Furthermore, the purpose of constructing this model is to determine the optimum performance, for comparison with that exhibited by various scheduling algorithms. Incorporating into these algorithms consciousness of fairly severe space constraints adds a whole new level of complication to their construction and analysis.

For these reasons, memory usage will be of relatively little interest to us. There is, however, room to wonder whether our step-function approach to measuring memory usage is always the most accurate. At a processor where some transitions are being executed only relatively infrequently, and neighboring network elements are not fully loaded, a dynamic solution in which algorithms are continually moved back and forth may actually further economize memory usage without reducing network throughput. This may cause total memory usage to be lower than given by equation (6.7). On the other hand, (6.7) certainly provides an upper bound on memory requirements, as long as buffering requirements continue to be regarded as negligible.

### 6.6: Discussion

Simple and limited though our linear programming model may be, it can be used to gain understanding of some important situations, and perhaps explode a few preconceptions one might be tempted to form regarding the performance of reference tree networks. One such preconception is that throughput of a reference tree network, calculated as described earlier in this chapter, can be made arbitrarily large by just adding more processors. As long as communication costs are nonzero, this is not true. Also untrue, as long as communication has a cost, is the thesis that adding more processors will cause a linear increase in throughput. These statements should be qualified somewhat: increasing the number of processors will always result in a linear increase in network *capacity*, but if events emanate from a fixed number of sources, more and more of that capacity will be eaten up in communication overhead as the network grows. If, on the other hand, the number of event sources increases with the size of the network, this effect can be avoided.

### 6.6.1: Performance of Some Simple Topologies

A few examples will help illustrate this point, and also give some context for evaluating MuNet performance results to be presented later. Consider the simplest possible computation, in which there are only two event classes: the initial event *class and the final event class*. There is only one kind of transition, which converts an event of the initial class into an event of the final class. As far as physical properties of the network are concerned, we assume that all processors and links are identical (*i.e.*, have the same performance parameters). We further assume that the bandwidth of links does not constrain system throughput. This is equivalent to positing that the CPU time to process incoming and outgoing events is greater than the link time required for their transmission, something which is true of the MuNet and a plausible hypothesis about many other systems. Finally, we assume that the CPU send and receive times for all events are equal to the constant $c$ (*i.e.*, $R_{im} = S_{im} = c$) and that the time to execute a transition on any processor is $t$ (*i.e.*, $T_{km} = t$).

Consider then the performance of a "pipeline" of $N$ processors, as shown in

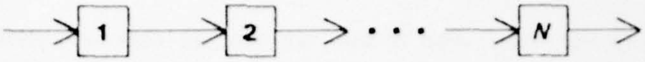


Figure 6.3: A pipeline topology

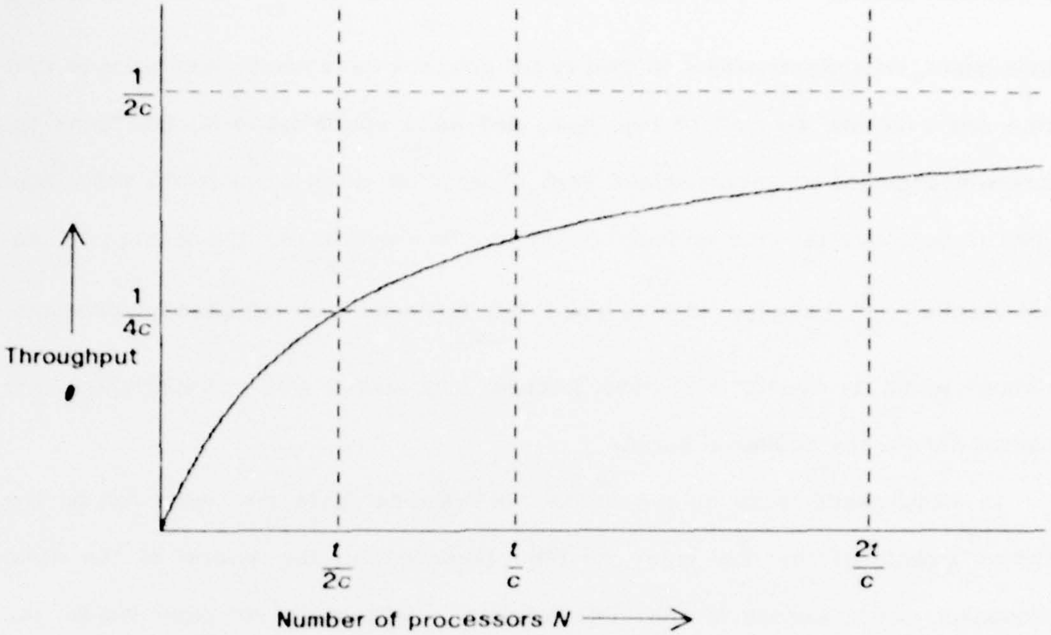
Figure 6.3, where events of the initial class are presented at processor **1**, and events of the final class are extracted from processor $N$. Our linear programming analysis gives as the throughput $\theta$ of this pipeline

$$\theta = \frac{N}{t+2cN}. \qquad (6.10)$$

The numerator in this expression gives the total number of units of CPU time available per unit of real time, while the denominator indicates the total number of units of CPU time needed to process each incoming event: $t$ units of time to actually execute the transition that converts it from an initial event to a final event, and $2c$ time units per processor to forward the event along from the beginning to the end of the pipeline. As $N$ becomes large, the throughput approaches an asymptote of $\frac{1}{2c}$, as each processor spends a higher and higher proportion of its time simply forwarding events. Thus adding unbounded numbers of processors does not result in unbounded increases in throughput. It is possible to graph equation (6.10) as shown in Figure 6.4.

In order to avoid the performance degradation that comes with having to send every event through a long pipeline, while still having only one source and sink of events, we might be tempted to try a "bushier" structure such as that in Figure 6.5.[*] The linear programming model now gives a throughput of

[*] This topology may be construed as violating the "limited number of neighbors" condition of reference tree network topologies, but this limitation is less important for our purposes than the performance limitations of the topology.



$t$ = time to execute a transition on a processor
$c$ = CPU time to send or receive an event on a processor

Figure 6.4: Throughput of the pipeline topology shown in Figure 6.3

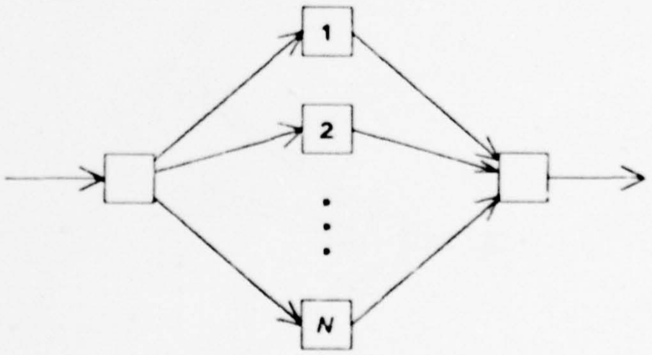


Figure 6.5: A parallel topology

$$\phi = \min\left(\frac{1}{2c}, \frac{N+2}{t+6c}\right) \qquad (6.11)$$

which increases linearly with $N$ up to the critical throughput $\frac{1}{2c}$. Below this critical throughput, an analysis similar to that in our previous case holds: $N+2$ units of CPU time are available per unit of real time, and each event takes $6c$ time units for communication (to be received and sent at each of three processors) and $t$ time units to be converted from an initial event to a final event. But the throughput cannot exceed $\frac{1}{2c}$ because at this point the left-hand and right-hand processors, through which all events must pass, become fully loaded with communications and cannot handle any additional events.

In many cases, such as the actual performance tests that were run on the MuNet, events of the final class must be taken out of the system at the same processor where events of the initial class are introduced. In other words, the answers must appear at the same place where the problems were posed. Thus the evaluation of MuNet performance results is more directly related to two topologies we now consider. *The first of* these, shown in Figure 6.6, is similar to the parallel topology of Figure 6.5, except that the source and sink for events is on the same processor.

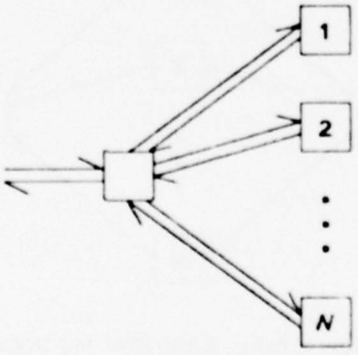


Figure 6.6: Parallel topology with common source and sink

The linear programming model gives for the throughput of this network

$$\theta = \frac{1}{t+2c}\left[1+(t-2c)\min\left(\frac{1}{4c},\frac{N}{t+2c}\right)\right] \tag{6.12}$$

which increases linearly with $N$ until it reaches a maximum value of $\theta = \frac{1}{4c}$ at $N = \frac{t}{4c} + \frac{1}{2}$. Numbers representative of the MuNet applications to be discussed later are $t$ = 0.8 seconds and $c$ = 30 to 70 ms, giving a maximum throughput of 4 to 8 tasks per second, first reached for $N$ between 7 and 3. Obviously this number is strongly influenced by the ratio $\frac{t}{c}$ that characterizes a particular application; on the MuNet, estimates of $c$ are particularly unreliable, since communication overhead seems to depend in a significantly nonlinear way on the amount of communication and on the kinds of other activities occurring on a processor and its neighbors.

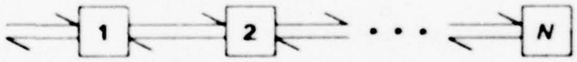
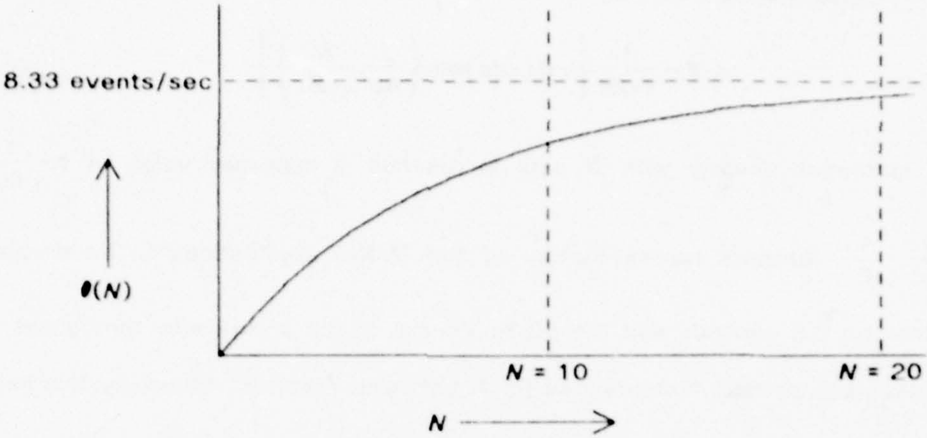


Figure 6.7: "Stack" topology

A final family of topologies, interesting because it was also the subject of actual MuNet experiments, and interesting as well because it is a simple example of a family of topologies whose maximum possible throughput has no simple closed form, is the family of "stack" topologies illustrated in Figure 6.7. The throughput $\theta(N)$ of a stack topology of length $N$ is given by the recurrence relation

$$\begin{aligned} \theta(N+1) &= \min\left(\frac{1}{4c},\frac{1+(t-2c)\theta(N)}{t+2c}\right) \\ \theta(1) &= \frac{1}{t+2c} \end{aligned} \tag{6.13}$$

A plot of $\theta(N)$ using the parameter values $t$ = 0.8 seconds and $c$ = 30 ms is shown in Figure 6.8.



0.8 seconds to execute a transition on a processor
30 ms of CPU time to send or receive an event at a processor

Figure 6.8: Performance of the stack topology shown in Figure 6.7

### 6.6.2: Assignment of Events to Processors

Other applications of the linear programming model are to computations more complex than the very simple case we have been considering. When there are several classes of events, the strategy issue arises of which kinds of events to execute where. This decision is not always as simple as it might seem. For example, consider the computation of the function $f(g(x))$ of some stream of input values $x$. Such a computation might be performed in two steps and involve three classes of events: an initial class containing input values $x$, an intermediate class containing partial results $g(x)$, and a final class containing results $f(g(x))$.

Figure 6.9: Two-processor pipeline topology

If this computation were performed on a two-processor pipeline topology such as that shown in Figure 6.9, it would seem appropriate to make the "natural" correspondence between the pipelined computation and the pipeline topology by doing all computations of $g$ on processor 1 and all computations of $f$ on processor 2. Such an assignment is actually best, however, only under fairly limited sets of circumstances. First of all, it will not generally be the case that this assignment will result in a balanced load on the system: only for very specific values of communication and computation costs will performing computations of $g$ on processor 1 at some rate $\theta$ keep that processor fully utilized while computations of $f$ on processor 2 occurring at the same rate $\theta$ are also keeping that processor fully utilized. More likely, whichever processor has been assigned the computation that involves lower computation or communication costs will have some idle time, meaning that system throughput could be increased by shifting to that processor some of the load on the other processor. Thus if the computation of $g$ were, say, only half as expensive as that of $f$, throughput greater than that exhibited by the "obvious" pipeline would be achieved by a solution in which processor 1 performed all computations of $g$ and also some of $f$, while processor 2 performed the remaining computations of $f$.

Beyond this minor load-balancing adjustment, however, lies room for questioning the whole idea of pipelining the computation at all. An alternative is to have each processor perform equal numbers of computations of $f$ and $g$, never passing any intermediate results $g(x)$ between processors. Only events of the initial and final classes would then be passed between processors. This approach will be inferior if the intermediate events are less expensive to pass between processors than the initial or final events, for then it will pay to convert initial events as quickly as possible to the more cheaply communicated intermediate form, and to put off for as long as possible the conversion to the final form — that is, it will pay to use the pipelined approach. On the other hand, if the intermediate events are more expensive to communicate than the initial or final events (a common situation situation in programs which build up a large collection of partial results before disgorging an answer), then the highest throughput will be attained if intermediate events are never communicated, but are consumed as soon as they are produced.

Thus the unmodified "natural" pipeline arrangement will very rarely yield the optimal throughput. In most cases, the only reason that would dictate such an arrangement would be memory constraints making it impossible to fit the algorithms for *f* and *g* both on the same processor. Although this might sometimes be a real problem, it is not likely to be an overriding concern in general.

The observations made in this section generalize in the obvious way to longer pipelines. The linear programming model remains able not only to give the maximum possible throughput of such configurations, but also to indicate the distributions of events which will bring about the *maximum possible* throughput. Perhaps more important than this capability, however, is the lesson that the distributions that seem the most natural are not always the best, and that the look of a distribution which *is* optimal may be profoundly influenced by small changes in the relative costs of various operations. A lesson for the development of scheduling strategies is that appealing heuristics such as "find and exploit any natural pipeline structures in the object program" should be approached with extreme caution. A scheme that is likely to be more generally satisfactory ought to adjust its behavior according to the run-time load effects produced by previous decisions, or else ought to rely on the results of a more thorough analysis of the program to be scheduled.

### 6.7: Summary

A model has been developed for the performance of reference tree networks executing certain simple computations of a recurring nature. Applications that would fall into this class include some kinds of real-time data acquisition, along with periodic repeated computations such as would be required, for example, to update a moving picture of a three-dimensional scene. Using linear programming methods, it is possible to determine the maximum possible throughput for such computations of a particular network. This throughput can then be used as a standard for comparison with the throughput achieved by particular scheduling algorithms.

The linear programming model can also be used to explore the inherent performance limitations of various topologies, and to look at strategies for assigning events to processors. The discussion of these topics at the end of this chapter is not intended to convey the impression that reference tree networks are not scalable. For one thing, the point at which the limitations examined become manifest depends strongly on the ratio of computation to communication speed at each node. More importantly, the examples in Section 6.6 are not designed to imply that reference tree network capacity increases less than linearly in the size of the network. The speed of performing any one set of calculations will at some point stop increasing as more processors are added, because at some point communication costs will dominate the extra computation power available; however, a larger network will still have increased capacity for *additional* concurrent tasks.

# Chapter 7: Scheduling Strategies for Reference Tree Networks

The term "scheduling strategies" is perhaps something of a misnomer for the subject of this chapter, which is little concerned with the scheduling of events in time, but much more concerned with the distribution of objects and events in space, *i.e.*, their allocation among the various nodes of the network. The former subject is not wholly without interest — the treatment by the system of tasks with differing priorities, or the response of the system to tasks with real-time constraints, would be important issues in many cases. However, spatial scheduling decisions crucially affect the efficiency of the system by determining the amount of internode communication that will be necessary in the course of executing a program. The amount of this overhead can have a profound impact on the ability of the system to meet real-time demands, but is of interest even to the user whose only desire is that the system execute his program in a reasonably expeditious and fair manner.

An interesting attribute of scheduling strategies that sets them apart from the rest of the material presented in this thesis is their discretionary nature. Up to this point, actions performed by the system have been prescribed by algorithms for conflict resolution, reference tree maintenance, *etc.* These algorithms are applied mechanistically, after having been invoked either by some part of a user's program, or by desires whose origin has until now remained unspecified, such as a desire on the part of a processor to obtain locally a copy of a text. The origin of these desires, which are not expressed directly by the user, is the network scheduling strategy. For example, if the scheduling strategy determines that an event is to be executed on a particular processor, and that event requires a copy of a certain object text, then it will become necessary to obtain a copy of that text on that processor.

In the context of a reference tree network, the discretion that may be exercised by scheduling strategies is extremely broad. VIM programs contain no explicit directions for allocation of events and objects to processors, and the reference tree algorithms will support, at varying levels of efficiency, any distribution whatsoever. Furthermore, moving events or objects is sufficiently inexpensive that it is practical whenever there is an indication that some benefit might be gained thereby.

One degree of freedom open to a scheduling strategy is the choice between demand-driven and what might be called notice-driven behavior. Any scheduling strategy will have to face situations that demand some action. For example, an executing event may request access to some text not currently locally available. In such a case, there are several plausible courses of action: a request can be made to bring a copy of the desired text to the current processor (so that the event can resume execution), the event can be sent to a processor that already has a copy of the text, or some compromise solution can be adopted. Whatever decision is made affects the distribution of events and objects and is properly termed a scheduling decision. A scheduling strategy might also operate in a more contemplative mode, however, observing the operation of the system and making adjustments that, although not demanded by any immediately critical situation, will help the system operate in a more efficient manner in the future. Examples of such "notice-driven" scheduling behavior include processor and memory load balancing, grouping of related objects on nearby processors, and the like. This chapter will concentrate mainly on notice-driven strategies, for it is here that the choices are widest and the opportunities greatest.

### 7.1: The Diffusion Strategy

The simplest imaginable load-balancing strategy that operates strictly on a local basis is a "diffusion" strategy in which load is continually transferred over each link from the more loaded end to the less loaded end. Under this strategy, any concentration of load at a particular point will gradually flatten and spread out with time, much as impurities diffuse through a crystal lattice. This strategy can be applied both to events (*primarily to balance processing load*) and objects (*primarily* to balance memory load). The strategy is appealing for its simplicity and elegance, but the applicability of the simplest version is limited by the fact that it ignores information about relationships between events and objects which could be used to make better scheduling decisions. Nevertheless, it does begin to address the issue of bringing into play as many as possible of the resources of a network, and can be modified to take into account some of the information that the simple version ignores. The diffusion strategy is also of interest because it is the current scheduling strategy on the MuNet, and thus some preliminary performance results can be reported.

The current diffusion strategy for events on the MuNet incorporates a parameter known, for historical reasons, as QFUDGE. The eagerness of events to diffuse throughout the MuNet is curbed by the requirement that, in order for an event to diffuse from a processor to its neighbor, the number of active events on the processor must exceed that on the neighbor by at least QFUDGE. There are two reasons for doing this. One is to enhance the stability of the system, and cut down on redundant communication, by trying to ensure that once an event has diffused to a new processor, it will not immediately diffuse back. The other is that diffusing events across a wider set of processors will, in general, increase the amount of communication that must occur. A particuler value of QFUDCE reflects a certain tradeoff between the benefits of using more of the system resources and the extra communication costs incurred.

### 7.1.1: Tests for the Diffusion Strategy

It is easy to construct examples where the simple diffusion strategy will exhibit very poor performance. The more structured the relationships among the events and objects that make up a computation, the more likely *diffusion* is to make "hash" out of that structure and stretch communication lines to unfortunate lengths. Indeed, many programs that would actually be written (assuming they contain enough parallelism to be able to benefit from multiprocessing in the first place) would probably suffer under the diffusion strategy. This section includes results from some tests contrived to show the diffusion strategy in the best possible light, to give an indication of what is possible and of what may be the ultimate limitations of the approach. While the actual computations performed were rather artificial, an attempt will be made to relate these to actual tasks that one might carry out using a reference tree network.

In the first test, which we shall call the "asynchronous" test, a controlled number of parallel tasks is introduced into the network, such that each task computes for a fixed period (this is a fixed amount of CPU time, not a fixed period of real time), notifies a metering actor of its completion, and then restarts execution from the beginning of the task. The tasks are simple timing loops not performing any useful computation, but the scenario is similar to that of a system periodically accepting packets of input data, processing them in some fashion, and emitting output packets.

To form an exact analogy with our test, a packet-processing application would have to have the property that a new input packet is received every time an output packet is emitted. Then the "degree of parallelism," or total number of input packets currently undergoing processing, always remains constant. Perhaps a more realistic scenario would be to assume that input packets arrived at regular intervals, irrespective of the rate of generation of output packets. This was not done on the MuNet due to the difficulty of generating tasks at precisely timed moments. However, a glance at the graphs presented below should convince the reader that if packets are presented at regular intervals not shorter than the average interval between task completions in our test (for the particular degree of parallelism that is of interest), then the MuNet should have adequate throughput to handle the job.

Another test (the "synchronous" test) conducted on the MuNet is closely related to this one, except that after the controlled number of tasks is introduced into the system, all must report their completion to the metering actor, after which all will be restarted in unison. A real-world analogy for this test might be found in a graphic display system, where at intervals (for example, when a viewpoint changes) it might be desired to recalculate the position of each point and line on the screen. Whereas the asynchronous test reveals only the *average* delay between the initiation of a task and its completion, the synchronous test is sensitive to *worst-case* behavior, and will perform more poorly than the asynchronous one if the network takes anomalously long times to complete some of the tasks.

A third test, conducted under somewhat less rigorously controlled circumstances, was of a program to compute, in a parallel fashion, solutions to the classical "eight queens" problem. The problem is to find placements for eight queens on a standard chessboard such that no queen could in one move capture any of the others. It is clear, from the rules of chess, that every queen must be *placed in a different file, or column, and that one queen must be placed in each file.* Thus a standard algorithm for solving the problem is to tentatively place a queen in one of the squares of the first file, and then continue placing queens in subsequent files in positions where they could not be captured by any queens already placed. If it is found impossible to place a queen in some file, then the algorithm must backtrack and find a different, previously untried arrangement of the queens already placed. The algorithm in effect searches a tree of possibilities, where each internal node corresponds to some partial queen placement. Each such node has eight sons corresponding to the eight possible placements of a queen in the next file. Leaf nodes represent potential solutions in which all eight queens have been placed. A tree-pruning heuristic in the algorithm causes backtracking when it is clear that no leaf node under the current node could possibly be a solution.

The program tested on the MuNet explores the same tree, but all of a node's sons that cannot immediately be pruned are explored in parallel, leading to a maximum of several hundred concurrent tasks. Each internal node under which active exploration is proceeding has an associated object recording the number of active sons of that node, and the number of solutions that have been found so far. The solution count is passed back up the tree as computations finish, until when all computations cease the root node has a record of the total number of solutions found. Thus the eight queens program does have some structure of interreferencing objects, and as a result we would expect the simple diffusion strategy to be less kind to it and lead to higher communication costs than in our previous tests. An additional factor pointing in this direction is that calculations in the eight queens program are not artificially prolonged, as in our previous tests; thus the ratio of communication time to computing time should be higher.

### 7.1.2: MuNet Performance Peculiarities

The reader will be able to make a more educated assessment of MuNet performance results if a few quirks in the operation of the MuNet are explained first. In the MuNet, being attached to a communication link is an item of significant CPU time overhead for a processor, whether any objects or events are being communicated over the link or not. This is because of a steady dialogue of status and acknowledgment messages maintained over each link by the processors at its ends, plus some reference tree protocol messages triggered by garbage collection activity. Thus, all other things being equal, adding a link to a system reduces the amount of available CPU time in the system. Comparing one-processor topologies to two-processor topologies is especially hazardous, for on the one-processor networks all the CPU time can be devoted to event execution, while on the two-processor versions not only does the diffusion strategy have its first opportunity to mix things up, but also the overhead associated with the link is felt for the first time.

Amusingly enough, if a processor is sufficiently busy executing events, the load it causes its neighbors usually drops! This is because, with a surplus of events to execute, less time is devoted to garbage collection and generation of status information. Thus the performance statistics show a reduction in communication link overhead for heavily loaded MuNets.

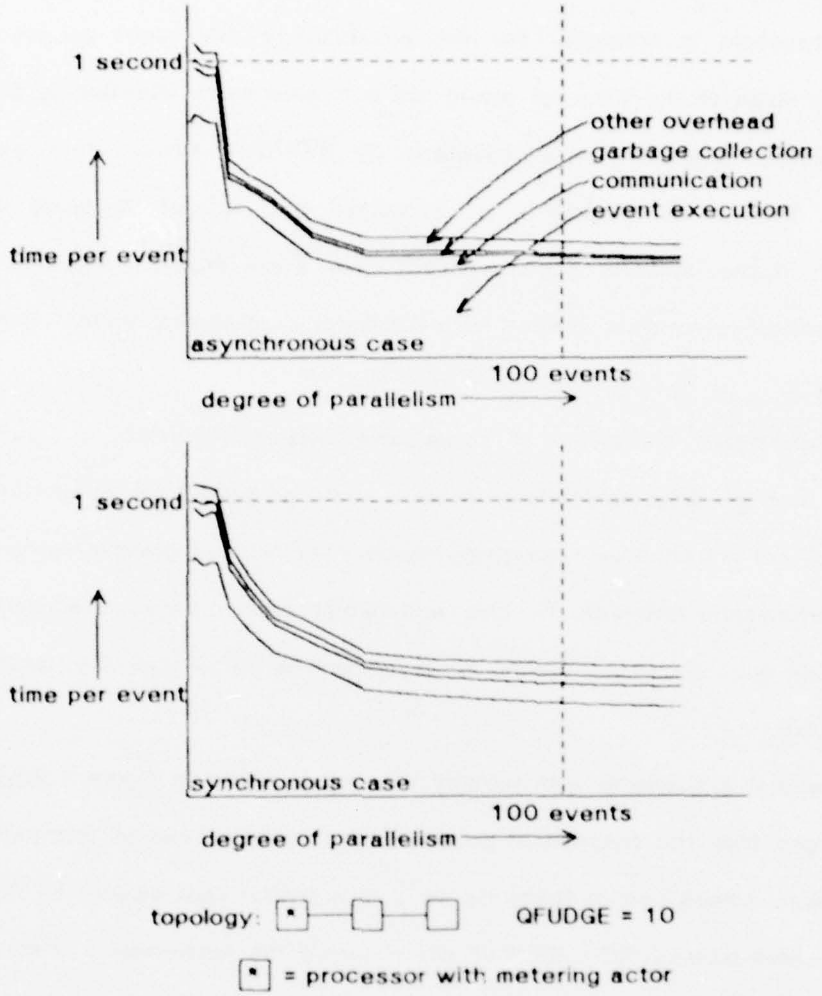
Neither of these performance quirks is an inherent property of the reference tree network architecture; the cause is the slowness of the LSI-11 microprocessors in the MuNet, combined with the simplicity of the link hardware which, as a result, requires a fair amount of attention from the processor. The quirks are discussed here only so that the reader can discount their effects, to the extent that he feels is justified, in examining the following performance results.

### 7.1.3: MuNet Performance Results

All MuNet performance tests included a special metering actor which recorded the start and finish times for task executions. For a variety of reasons, many of which are uninteresting from the point of view of this thesis, the metering actor used a special MuNet facility to "wire itself down" to a particular processor (this facility is described in Appendix A). Thus all new tasks were created on this processor, and all completed tasks ultimately had to report back to it. One dividend of this approach is that it makes possible a fairly direct comparison to the linear programming model, assuming one input link and one output link each attached to the processor containing the metering actor. Another payoff is that the metering actor can record the proportion of time spent *on its processor* for event execution, communication, and other overhead (also getting a *simultaneous picture* of the state of other processors is harder). This data helps explain some of the performance results recorded, and also provides the basis for crude estimates of performance parameters for comparisons with the linear programming model.

The execution times for tasks used in these tests are quite long. In part, this is due to the slowness of the LSI-11 microprocessors executing them, but it is likely that similar tasks in real applications would be shorter, even on LSI-11's. Another aspect of the slowness of LSI-11's is that interprocessor communication times (the parameter $c$ in Section 6.6.1) are relatively long on the MuNet. To bring the ratio of communication to compute times closer to what one can realistically expect in the future, the compute time was artificially lengthened (it is true that this also makes the performance data look more pleasing). The only case where this was not done was in the eight queens program.

Figure 7.1 gives an example of the direct output from the performance tests. As can be seen, the time per event (which is the reciprocal of the throughput)



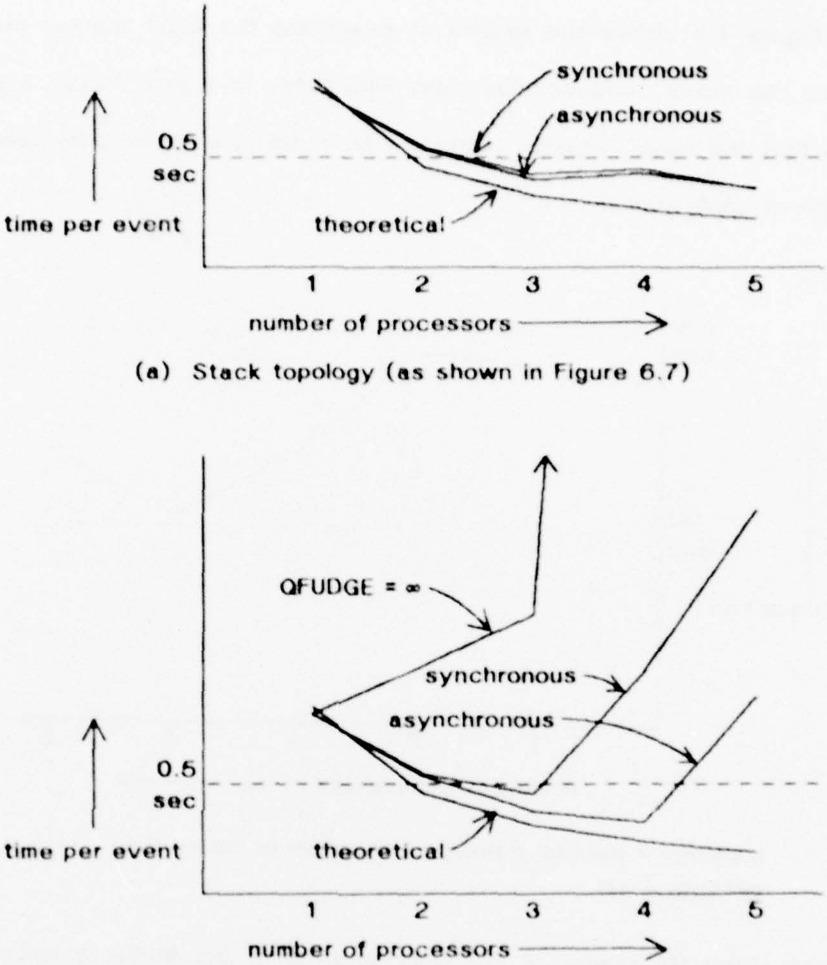
*These graphs show raw performance data obtained from the MuNet on both the asynchronous test (top graph) and the synchronous test (bottom graph). The "degree of parallelism" is the number of events simultaneously active in the system. The "time per event," or inverse throughput, is given both in the aggregate (top curve in each graph) and by components (labeled by name in the top graph). These components indicate the fractions of CPU time devoted to different activities* on the metering processor only. *The graphs show that, on this three-processor topology, throughput increases by a factor of between two and three in the presence of enough parallelism.*

*Figure 7.1:* MuNet performance data

starts out at a high level, when the degree of parallelism is too low to cause any diffusion to take place, then drops and levels off at a new, lower value as the QFUDGE threshold is crossed. For the experiments that were conducted, this asymptotic value of the time per event did not seem much affected by the exact QFUDGE used; only the precipitousness of the drop toward that asymptote changed. Thus it is reasonable to associate this asymptotic value with the throughput of the network, when sufficient events are present, and compare this with theoretical predictions derived from the linear programming model. This is done in Figure 7.2.

The reasonable agreement of actual and theoretical results in Figure 7.2(a) indicates that the diffusion strategy is doing about as good a job as can be done in this case — not a particularly surprising result. The lack of greater improvement in actual performance between the one- and two-processor cases is a result of the performance quirk discussed earlier, which occurs as the original processor gets its first neighbor.

The actual experience with parallel topologies shown in Figure 7.2(b) eventually diverges from the theoretical because of a serious case of this performance quirk. These curves cannot really be seen in a proper light except by comparison with the curve labeled "QFUDGE = ∞" which shows the performance characteristics of the topology when all events are constrained to execute on the metering processor. As the number of processors passes four (the metering processor plus three neighbors), throughput drops *almost to zero. Examination of the detailed* curves for this case reveals that the metering processor is spending virtually all its *time in communication overhead.* This is because MuNet processors give communication functions unconditional priority over event execution; with three or four neighbors, at least one of them is almost certain to have something to send a processor at any given time. Thus there is very little chance for events to execute on



(a) Stack topology (as shown in Figure 6.7)

(b) Parallel topology (as shown in Figure 6.6)

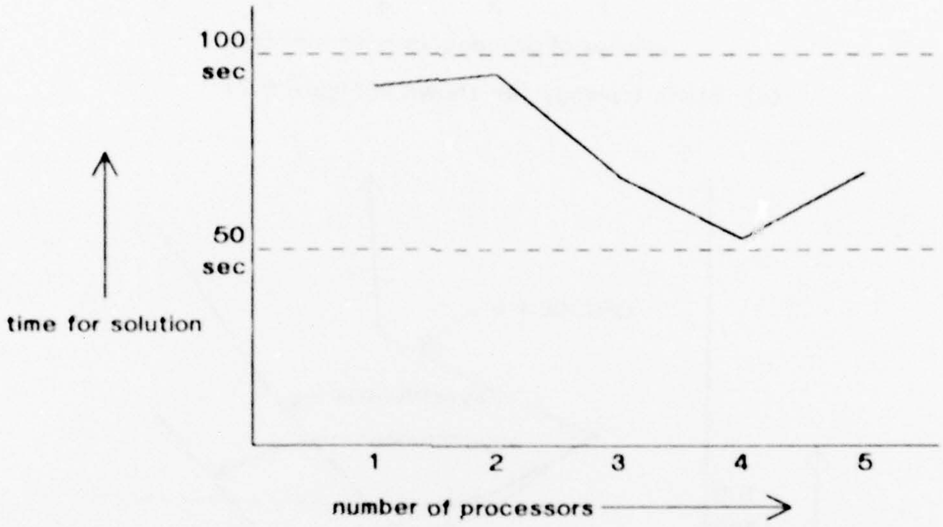
Parameters used for theoretical model: $t$ = 0.8 seconds, $c$ = 30 ms.

*These graphs show the actual throughput characteristics of two kinds of MuNet topologies in the presence of large numbers of concurrently executing events, for both the asynchronous and synchronous tests. Theoretical predictions derived from the linear programming model of Chapter 6 are included for comparison. Anomalies in the lower graph are discussed in the text.*

Figure 7.2: Throughput of the MuNet

the metering processor. When other processors are busy executing events, this effect is ameliorated somewhat, but it still eventually makes itself felt.

Finally, Figure 7.3 shows the results of executing the eight queens program. The shape of this curve compares favorably with those of Figure 7.2(b), a reason for optimism that the basic diffusion strategy can in fact be useful over some reasonable range of applications.



topology = parallel topology (as shown in Figure 6.6)

QFUDGE = 10

*This graph shows the amount of real time required by the MuNet to solve the eight queens problem, using different numbers of processors.*

Figure 7.3: Eight queens performance results

### 7.1.4: Diffusion of Objects

The foregoing discussion has concentrated largely on the diffusion of events through the network to take greater advantage of the computing power available. It is equally reasonable to think of *diffusion as a distribution strategy for objects* to take better advantage of the memory space collectively available among the nodes in a network. Of course, even without explicit diffusion of objects, diffusion of events does imply a certain strategy for distributing objects about a network. Once an event has diffused to some location where it is to be executed, objects it accesses must either already be present there or be brought there; otherwise, the location of the event will not have been determined solely by diffusion, but also by other factors. This latter may well be a preferable state of affairs, but represents a departure from the pure event diffusion strategy.

The moral of the story is that the distribution strategy for objects cannot be divorced from that for events, and if independent diffusion strategies are to be used for each, then some policy must be adopted for resolving conflicts when they arise. A reasonable approach would be to use object diffusion to secrete away little-used objects wherever there is extra memory space, but let frequently accessed objects be distributed so as to be near the events that use them.

The object distribution question has other facets as well. When and where should multiple copies of an object be made? What kind of strategy will keep related objects on the same or neighboring processors, to minimize reference tree overhead and make sure that once an event has been moved to a processor where there is a copy of one object it needs, copies of other useful objects will be available nearby? In general, it will probably be harder to operate efficiently in an environment where objects *must* be distributed due to lack of memory space than one in which events *must* be distributed due to lack of computing power.

Nevertheless, the questions cannot be completely separated, so the next section makes a stab at a better strategy for event and object distribution.

## 7.2: Extensions to the Diffusion Strategy

### 7.2.1: Pull Factors

The main problem with the simple diffusion strategy is not its concept of spreading the load across the network, but its naive assumption that events (or objects) are all interchangeable, and that sending one event out in a particular direction is as good as sending another. In fact, different events may well have "roots" (*i.e.*, copies of objects they access) in different directions, and if it is desired to send an event in a particular direction for load balancing, some events may be much more appropriate for this purpose than others.

To reflect this distinction, one might imagine adding "pull factors" to the basic diffusion strategy. If an event were being moved closer to copies of the texts of any objects it referenced, the QFUDGE in effect for moving the event in that direction would be reduced by the number of such objects times the pull factor. If an event were being moved away from copies of the texts of objects it referenced, multiples of the pull factor would be added to QFUDGE. The event would thus experience a "pull" in whatever direction most of the object texts likely to be useful to it might lie. This strategy is analogous (assuming for the moment that the distribution of objects has been fixed in advance) to attaching a spring between each event and every object it references, then allowing the events to be pulled around until equilibrium is reached. A form of our original diffusion strategy can be incorporated into this analogy by imagining a repulsive force between any pair of events, encouraging them, other things being equal, to spread as widely as possible across the network.

Just as pull factors can be used to improve the diffusion strategy for events, they can be used with the diffusion strategy for objects. Of course, the use of pull factors alone would tend to pull all objects together onto one processor. Such a distribution might well demand memory space in excess of that available on that processor, and does not open any very interesting possibilities for the distribution of the events that use those objects. These problems can be solved by counterbalancing the attractive force of object references with a repulsive force tending to move objects away from high concentrations of other objects. The synergy of these two forces would then cause tightly related groups of objects to cluster together on the same processor, while objects more tenuously related would be stored farther away.

To get some idea of the possible effects of this strategy, some simulations were carried out using a structure of interreferencing objects that would be characteristic of numerical iterations over two-dimensional arrays of grid points, as in solutions of Laplace's equation or other sets of partial differential equations. Semantically, one may imagine these objects arranged in a two-dimensional grid (this says nothing about their actual physical distribution in a network) such that each object has references to its neighbors on the north, east, west, and south. Objects along the edges of the grid, of course, have fewer neighbors.

The iterative computation to be performed over these grid point objects can be described in general terms as follows. When an iteration is to begin, this fact is signalled to each grid point. Each grid point then sends to each of its neighbors the current values of the relevant state variables at that point. When a grid point object has received these state variable values from all its neighbors, those values are combined with the current state at the grid point to produce the new state that will reign at that grid point when the iteration is complete.

As our running example of object distribution strategies, we will assume a 12×12 array of grid point objects, mapped onto a 4×4 square network of processors. Since each iteration of the computation results in a side effect (updating the state variables) to each grid point object, there is little point in storing multiple copies of any of these objects. We will assume that the object distribution algorithm is aware of this and keeps only a single copy of the text of each object. Thus it is meaningful to talk about *the* location of a particular object text.

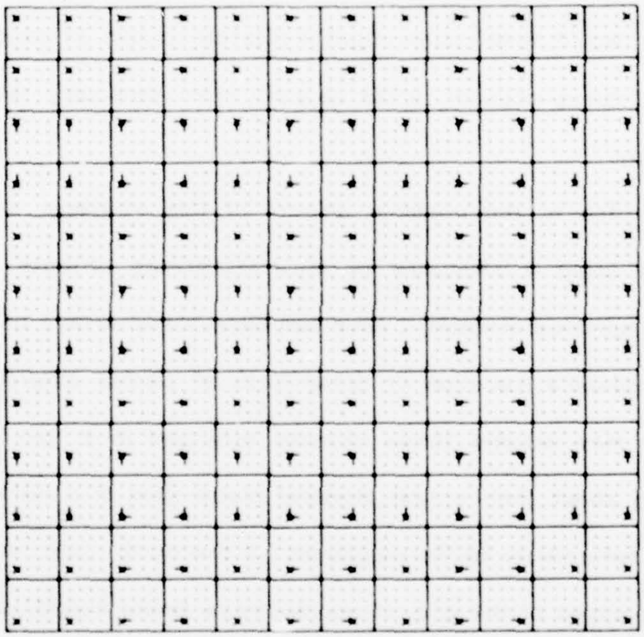


Figure 7.4: Ideal distribution of grid point objects

The "obvious" ideal assignment of objects to processors in this example is the assignment of a 3×3 subarray of grid point objects to each processor, such that adjacent subarrays are assigned to adjacent processors. This assignment is depicted in Figure 7.4. The 12×12 array of squares in this figure represents the

12×12 array of grid point objects. Within the square corresponding to each object is shown a picture of the reference tree for that object. The 4×4 array of dots in each picture represents the 4×4 processor array; the processor marked with an "x" is the processor where the object's text is stored. Solid lines mark the interprocessor links that comprise the object's reference tree. The reader can verify that in every case this tree includes every processor containing a reference to the object.

Results of applying the simple diffusion strategy to distributing objects are shown in the series of snapshots in Figure 7.5. Figure 7.5(a) shows the initial configuration, with all grid points stored on the lower left-hand processor. To obtain each subsequent snapshot in Figure 7.5 from the previous one, one pass was made over the entire set of grid point objects (in a randomly determined order), and the simple diffusion criterion applied to each one to see if it should be moved to a neighboring processor. This is not an exact simulation of the way in which things would happen on a real reference tree network (presumably there would be more concurrency), but the results should be fairly similar. In this case, we can see that the result is a series of long, stretched-out reference trees.

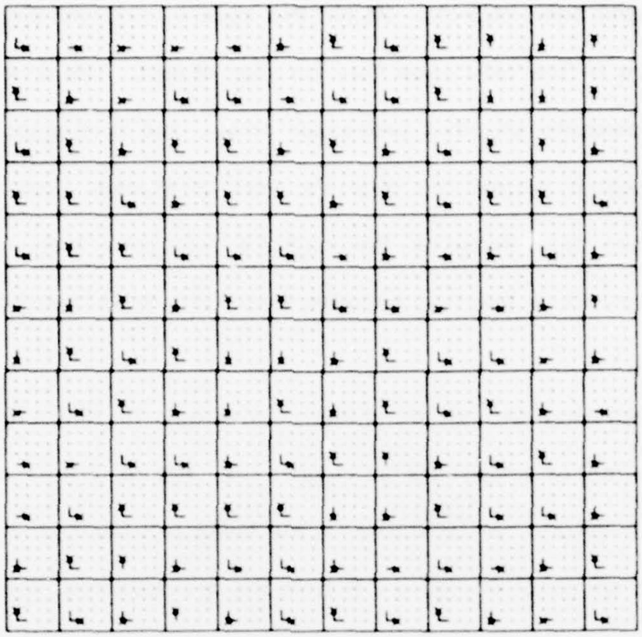


Figure 7.5(b): After one pass using the simple diffusion strategy

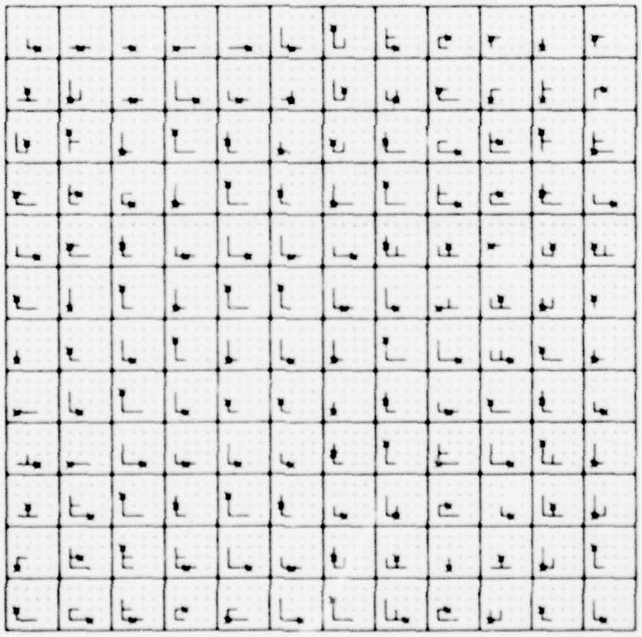


Figure 7.5(c): After two passes using the simple diffusion strategy

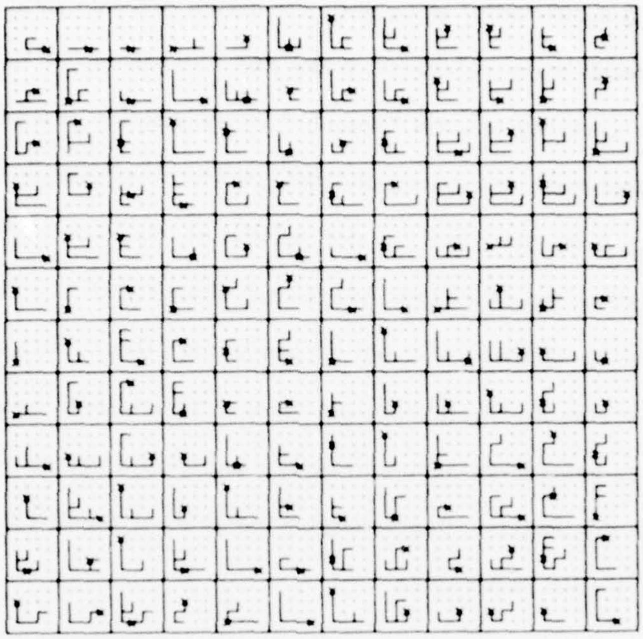


Figure 7.5(d): After four passes using the simple diffusion strategy

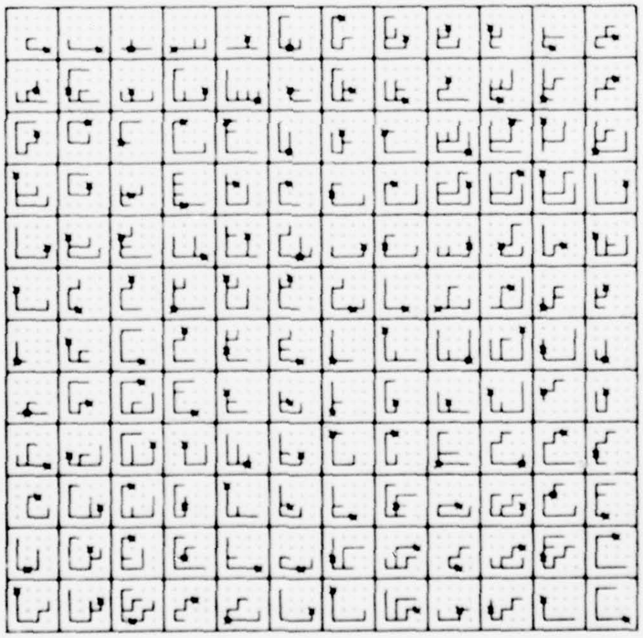


Figure 7.5(e): After six passes using the simple diffusion strategy

Results of using a pull factor along with the simple diffusion strategy are shown in Figure 7.6. Although many reference trees still become larger than in the ideal situation, there are certainly several "domains" of neighboring objects that have all been assigned to the same processor, and the growth of reference trees, although still far greater than in the ideal, is not nearly as luxuriant as in Figure 7.5. Another thing to notice about Figure 7.6 is that the use of pull factors has put enough of a damper on diffusion that not all processors are being used equally. In fact, several processors on the upper right have not been used at all. Thus lower communication costs have been achieved at the expense of an increased computational load at some processors. Depending on the ratio of these costs, this may or may not represent a good compromise.

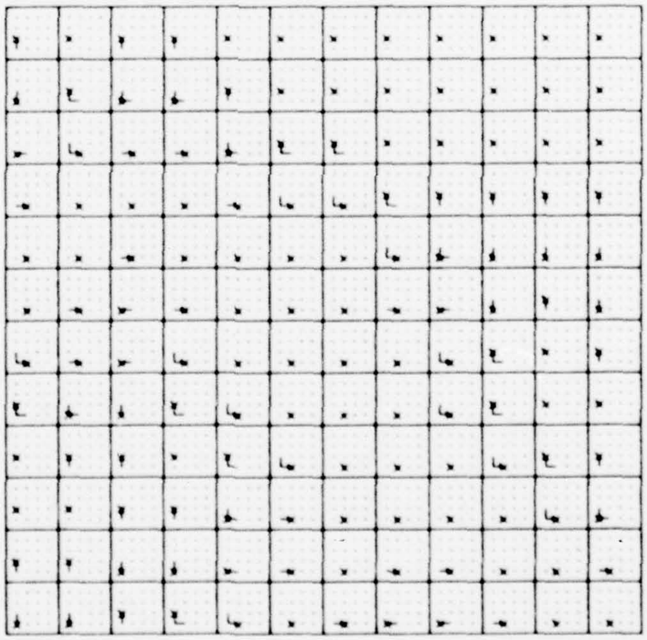


Figure 7.6(a): After one pass using pull factors

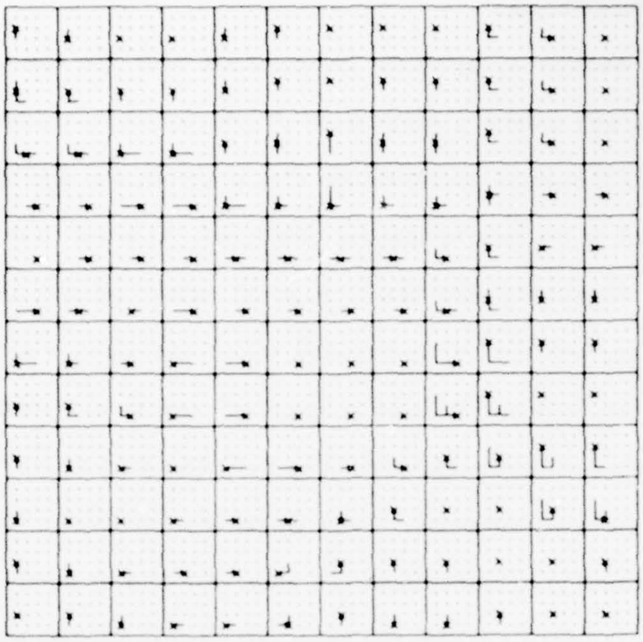


Figure 7.6(b): After two passes using pull factors

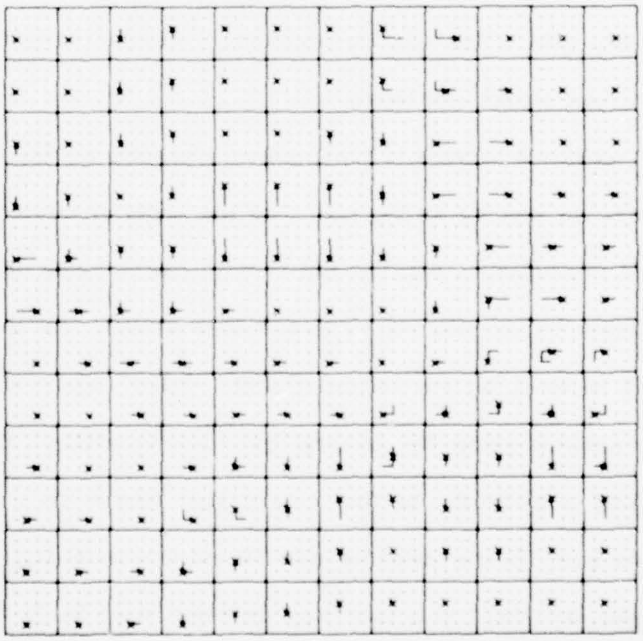
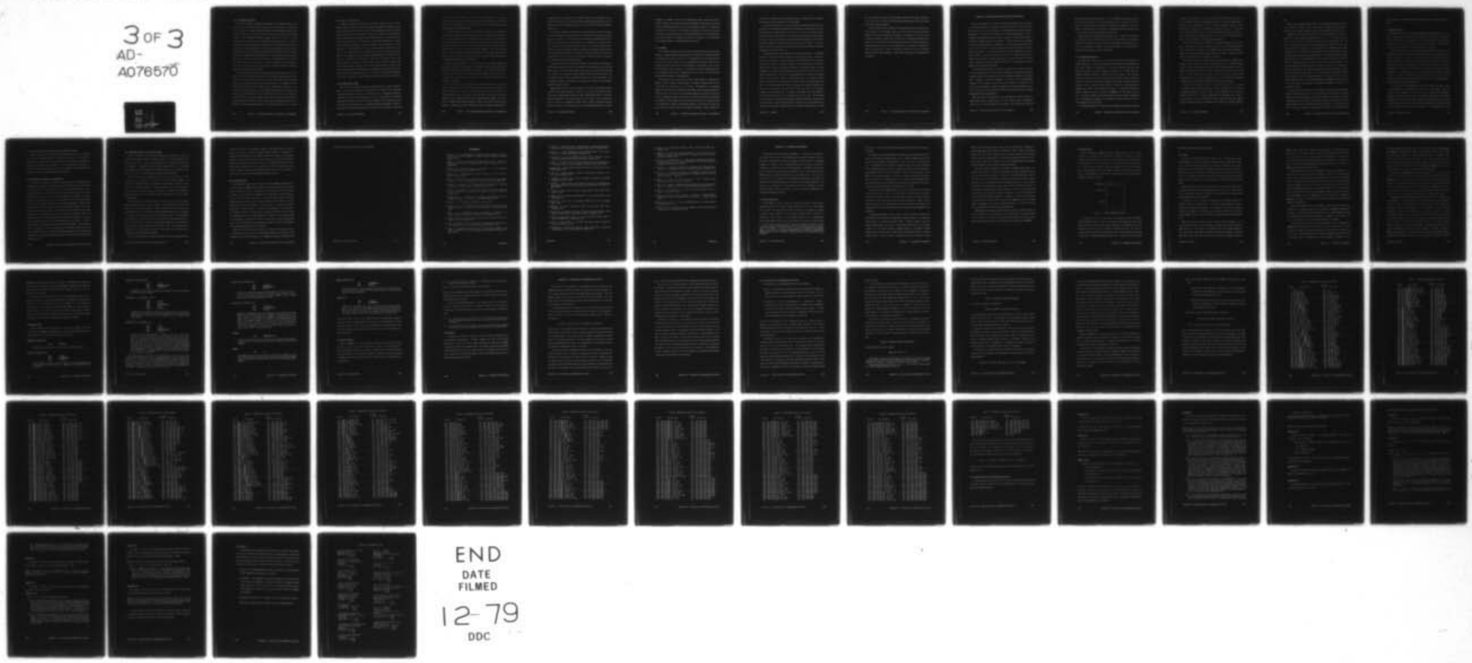


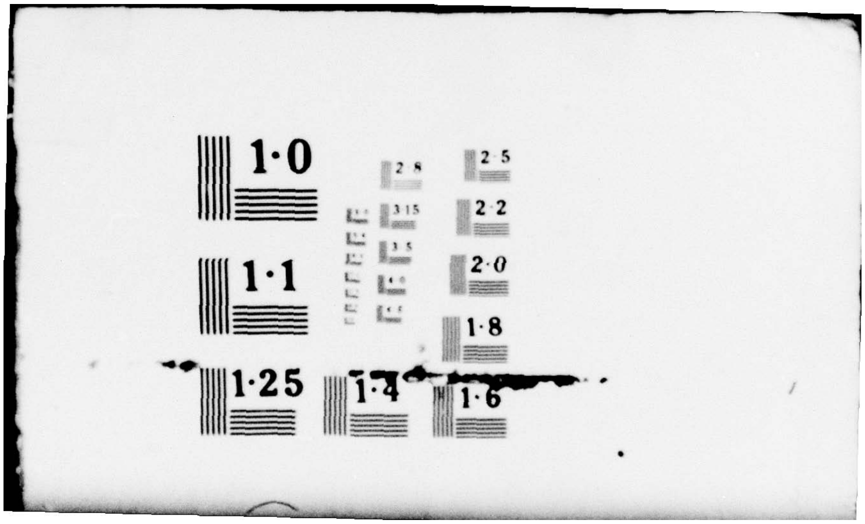
Figure 7.6(c): After nine passes using pull factors

AD-A076 570 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 5/8
REFERENCE TREE NETWORKS: VIRTUAL MACHINE AND IMPLEMENTATION.(U)
JUL 79 R H HALSTEAD N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-222 NL

3 OF 3
AD-
A076570



END
DATE
FILMED
12-79
DDC


1·0
2·8
2·5
3·15
2·2
3·5
1·1
2·0
1·8
1·25
1·4
1·6

### 7.2.2: Variable Pull Factors

Pull factors manage a significant improvement over the simple diffusion strategy, but there is still clearly a great deal of room for improvement. Unfortunately, we are coming to the edge of information that a scheduling algorithm might reasonably be expected to be able to discover for itself. In order to make much further progress, some scheduling information supplied either by the user or by some language processor will have to be included with objects and events. In general, it is desirable for this information to be expressed in as topology-independent a form as possible, to make it applicable to running the same program on a wide variety of different reference tree networks. Even within this constraint, the most appropriate form for scheduling specifications will vary. For the array-processing example dealt with in the previous section, the best form of specification might be an indication that the grid point objects fit on a Cartesian grid, giving the co-ordinates of each object on the grid. This section presents a simpler scheme that should be suitable for many less structured tasks.

One problem with pull factors is that every reference contained in an event or object text exerts an equally strong pull. For the array-processing example given above, this is not especially a liability, but in most cases some objects referenced are much more likely to actually be accessed than others. For example, an event should be pulled hardest by its target object. Other object references in the event might be passed along for several generations without the corresponding objects ever being accessed; to the extent that this is true, such references should exert less pull. Similar remarks can be made concerning pulls on objects. Certain references contained in a text may be much more likely than others to be followed by anyone examining the text; the references most likely to be used should exert the strongest pull.

If different references contained in an event or object text are to exert different degrees of pull on it, the question arises of whether the strengths of pull should be specified in the event or object text being pulled, a particular strength being assigned to each slot containing a reference, or whether the strength of pull should be determined by the object being referenced, so that a given object will pull with the same force on every event or object text referencing it. In other words, should the strength of pull be a property of the referencer or of the referencee? A strong argument for it being a property of the referencer is that an object is not equally likely to be accessed over all paths by which it might be reached. For example, an object named as the target object of an event is certain to be accessed by the execution of that event. A reference to the same object, entered in a directory of a file system, might be passed over almost every time the directory was examined. Even on the occasions when the reference was retrieved from the directory, another long interval might elapse without the associated object text actually being accessed. It stands to reason that the reference to this object should exert a much stronger pull on the event than on the directory.

## 7.3: Communication Graphs

Thus far, the scheduling strategies suggested all have a fairly mechanistic flavor: a user with a program to run might have the operation of the scheduling mechanism explained to him and be introduced to the changeable parameters accessible to him, but then he would be on his own. There would be no guidance other than experiment to tell the user which parameter values would best accomplish what he wants, namely to run his program as quickly as possible. The user is being asked to inform the system about the nature of his program, so that the system can execute it efficiently. But the form of the specification is so alien to the way in which the user might describe the program that the only way of telling which of two specifications is the more accurate is to experiment with them and see which results in faster execution!

The situation faced by a user of this mechanism is similar to that confronted by a designer of real-time systems whose only means for influencing system behavior is by assigning priorities to tasks. What the user would like to do is specify the required real-time behavior, which could then be satisfied by the system in any suitable manner. What he ends up doing is tinkering with priorities of tasks until the system appears to meet the required real-time demands. This is bad because there is no formal framework to indicate what sets of priorities should be used, and there is often no assurance that the set finally chosen will really guarantee the required performance under all circumstances.

Similarly, it would be better if a user of a reference tree network could simply describe the behavior of his program in reasonably high-level terms, rather than indulge in tweaking little-understood fudge factors. One possibility along these lines would be for the user (or some programming language processor) to supply a "communication graph" showing the frequency of communication between pairs of modules in the program.

The actual way in which communication would happen between a pair of VIM objects is that an event or a set of related events would access both objects. If copies of the texts of the objects were stored at some distance from each other, then either object texts or events or both would have to be moved, incurring communication costs roughly proportional to the original distance between the object texts. A communication graph would have a node for each object, and an arc for each pair of objects that might communicate. Each arc would be labeled with the communication cost incurred per unit distance that the corresponding pair of objects were separated. The object distribution problem would then reduce to the problem of finding an assignment of objects to processors that minimized communication costs.

The obvious assignment that minimizes total communication costs is the assignment of all objects to the same processor. In some cases, this might be ruled out by memory constraints. In most other cases, this would be undesirable because it would encourage all events using the objects to congregate on the same processor, resulting in poor load sharing. Perhaps some arcs should have *negative* communication costs, indicating that the objects involved do not communicate, and furthermore that if they are assigned to the same processor, events using them will be competing for CPU time. It must be said, however, that this solution is rather *ad hoc*, and that the problem needs more study.

Another aspect of object distribution strategy is the decision as to whether to *make multiple copies of an object*. The wisdom of this will depend on the relative frequencies of accesses and side effects to the object. A good way of conveying this information in a communication graph might be to label each *node* with the cost of dispersing copies of the corresponding object over some unit distance. This cost would be high if side effects to the object were expected to be frequent; otherwise it would be lower.

Making the best use of all the information in a communication graph is almost certainly an NP-complete problem; however, it is an open question what kinds of approximate methods might be able to produce good enough results to make the approach worthwhile. Other open questions exist as well. Can a communication graph be useful to a distributed scheduler each of whose components operates strictly on the basis of local information, or must global knowledge of the network topology be available? How can the communication graph concept deal with a dynamic situation in which objects, and hence potentially nodes and arcs, are being created and garbage-collected? How is all the detailed information in a communication graph to be supplied? Clearly communication graphs are currently a long way from being a practical medium for scheduling specifications, but the development of some such mechanism for allowing the nature of a program to be communicated more straightforwardly is an important goal for the future.

### 7.4: Summary

This chapter, more than most in this thesis, raises many more questions than it answers. A simple scheduling strategy, based on "diffusion" of events and objects to less-loaded processors, seems to do a reasonable job of load balancing, but does not obtain any guidance from the relationships among data items and may therefore increase communication costs disastrously. An example of such a disastrous increase is given in Figure 7.5. Incorporation of fixed or variable "pull factors" ameliorates this situation considerably (see, *e.g.*, Figure 7.6), but still leaves an undesirable situation in several respects.

One attribute of the "pull factors" solution which might be regarded as a feature is that it provides a large number of parameters through which the user can twiddle the performance of the system. Unfortunately, the relationship between the structure of a program and the optimum values of these parameters is often obscure. The concept of *communication graphs* is an attempt to bring the level of scheduling specifications closer to the level at which the user thinks about his program, but many details of both the structure and use of such graphs remain to be worked out. Whether communication graphs or pull factors are used, however, there is still a large amount of scheduling information to supply. How much of that can be calculated by some language processor scanning the user's program, and how much must be supplied by the user himself?

Even though the introduction of pull factors works a marked improvement in the performance of the diffusion strategy, many situations are still treated in far from the ideal manner (compare Figures 7.6 and 7.4). It is hard to know what improvements over the pull factor method can be achieved by using communication graphs or the like, since the performance achieved using these depends on just how the information contained in them is analyzed and used by the network. It is reasonable to conjecture, though, that no such method will match the results of doing the distribution by hand, as in Figure 7.4.

To keep from being *too* pessimistic about this state of affairs, it is worthwhile to make an analogy with paging strategies on virtual memory systems. These strategies, too, must make object distribution decisions (between primary and secondary memory) and respond to changing conditions. These strategies, too, attempt to make decisions based on the observed behavior of running programs, and these strategies, too, could benefit from being told more about the exact nature of a program being executed. Yet in most cases programmers have refrained from fiddling with the virtual memory strategy or taking matters out of the paging system's hands by doing their own input/output explicitly. There are several reasons for this. One is that for many large systems of programs, the optimal scheduling strategy is no less obvious to the system than to the user. Another is that paging strategies do well enough that it is ordinarily not worthwhile for the programmer to try to squeeze that last ounce of performance from the system. With hardware costs decreasing, this argument becomes more and more compelling. Although there are several ways in which the virtual memory problem is less complicated than the reference tree network scheduling problem, a reasonable research goal is to make the two reasons given above valid statements about reference tree networks as well. Then users can go back to getting their algorithms right, rather than fussing with tunable system parameters.

The performance results presented in this chapter are very preliminary, but at least they are not discouraging. Much work with more substantial applications remains to be done to determine the adequacy, or inadequacy, of even the current scheduling strategies. Beyond this, it must be seen if reasonable scheduling strategies can be built around the cherished reference tree network philosophy of decision-making on the basis of local information only, or whether more global information and interaction is required. Although it would be nice to have a beautiful theory to answer these questions, it seems that a great deal of experimentation lies ahead.

# Chapter 8: Conclusions and Directions for Future Research

The future development of computer systems will be influenced by two thrusts of technological progress: the ability to build circuits which are faster, and the ability to build circuits which are bigger (*i.e.*, which have greater numbers of logical components). The former thrust will lead to continued improvement in what is attainable using architectures that are now in vogue, but the latter thrust poses a challenge to develop whole new architectures. If the construction of a computing system with a greater number of components is to render it more powerful, there is no choice but to design a system in which more and more operations can be performed concurrently. In an architecture based on strict sequentiality, once the logic necessary for all functions has been provided, it is difficult to achieve any further speed increase by adding more logic, since what matters is the total number of operations needed to implement a function, not the variety of physical gates which perform those operations.

The technological opportunity to build more complex circuits, if they can be put to use, can be exploited to some extent by switching to architectures which are internally parallel, even though they still give a superficial appearance of sequentiality. However, to take full advantage of the fruits of the very-large-scale integration (VLSI) revolution, the user must be introduced to the world of concurrency and encouraged to express his algorithms in as parallel a fashion as possible. The purpose of the research described in this thesis has been to develop methodologies for accomplishing this.

More specifically, this research may be described as a search for ways of making multiprocessor systems usable. The word "multiprocessor" is intended to convey, first of all, the presence of concurrency, and secondly, the idea that general-purpose processing is going on. This is not meant to imply any restriction on the physical incarnation of that processing power, but is intended to narrow our scope to include only systems capable of general-purpose computation. The architecture of special-purpose VLSI systems, for example, although an important topic in its own right, is not a topic addressed by this thesis.

Within the sphere of general-purpose multiprocessor systems, the strategy of this thesis has been to make a broad cut across the entire domain, covering both a proposed user interface to such systems, an implementation supporting that interface, and a preliminary evaluation of that implementation. It is appropriate to review the progress made in each of these areas.

### 8.1: The VIM Virtual Machine

The VIM virtual machine is only one of many that have been suggested for parallel computation: communicating sequential processes[19] and data flow[7,30] are notable examples of alternatives. On the more concrete side, the virtual machines presented by modern timesharing systems, such as MULTICS[31] or UNIX[29], deal with interaction between concurrent activities. These systems, however, frequently either fail to provide facilities powerful enough to support all the kinds of interaction that might be desirable (the limitations of UNIX *pipes* are an example of this), or provide facilities difficult to support on many multiprocessor systems (*e.g.*, memory shared between processes). A final virtual machine for parallel computation, the actor machine of PLASMA[18], is the closest cousin to VIM. The design of VIM was heavily influenced by the actor model of computation, and in fact VIM may be thought of as a reduction to practice of many of the concepts embodied in the actor model.

The interrelationships of the various parts of VIM have already been discussed in Chapter 2, particularly in Section 2.7, and will not be further dealt with here. By way of comparison with other alternatives, however, it is worth recapitulating the advantages of the VIM approach. First of all, VIM is simple and tractable. It can be described precisely by the blackboard interpreter of Section 2.5; that interpreter could be modified easily to cover the enhancements proposed in Chapter 3. The essential simplicity of VIM stems from its roots in the mu calculus[15,38], a simple syntactic formalism for message-passing computation.

Second, VIM is flexible and adaptable. This flexibility manifests itself in two ways: in the ability to create complex structures such as sophisticated synchronization operators out of the basic set of VIM primitives, and in VIM's amenability to extension to support additional demands imposed by desires to implement various operating system functions, as illustrated in Chapter 3.

Third, VIM is designed for efficient execution on hardware of a familiar nature. Although it does support actor-style message passing and a garbage-collected space of objects, two concepts that have not often been associated with run-time efficiency, VIM also allows direct access to object texts and permits the composition of several functions into the text of one type object, obviating some message-passing activity at the very lowest level. As time passes and hardware designs progress, this attribute of VIM may well decrease in importance.

Finally and most importantly, VIM is a powerful and effective tool. Its orientation toward objects corresponds with various modern views about how to structure programming systems[5,24], and its ability to handle parallelism suits it ideally to multiprocessor systems. The various guarantees that VIM makes with respect to the co-ordination of event executions and object accesses reduce the range of possible execution histories a programmer must consider and thereby simplify his task.

VIM is hardly a finished product, though. The operating system support extensions discussed in Chapter 3 need to be worked out, implemented, and reconciled with the ideas set forth by Gula[13,14] on structuring a MuNet operating system.

Significantly, "machine-level" VIM is not a very easy language to program in, despite the various simplifying guarantees made by it. This is primarily because many of the machine-level concepts are at too low a level for everyday use. The need to avoid resource requests after performing a side effect during the same event execution, for example, is a constant distraction. Additionally, many useful concepts, such as procedures and environments, must be synthesized out of more primitive facilities available at the machine level. Thus some higher-level programming language, such as MuSpeak[34,35], is vitally needed to make practical use of VIM. An interesting question, in fact, is the extent to which the nature of the basic VIM execution environment should be visible to a user of such a language. It cannot be argued that the simplifying guarantees made by VIM actually uncomplicate the programmer's task unless it can be shown how those guarantees affect the environment in which the programmer works. If all characteristics of VIM are hidden snugly beneath a rather different-appearing programming language, then the use of VIM may be a convenience to the language implementor, but can be of no concern to the user.

Use of the VIM virtual machine can help make a multiprocessor system simple, flexible, efficient, and powerful. This is true whether or not any of the other implementation mechanisms suggested in this thesis are employed. However, the design of languages to serve as the primary interface between programmers and such a system remains an open (and important) research question. The extent to which the characteristics of VIM are visible through this interface will help determine the real significance of the VIM concept as a good organization for multiprocessor systems.

### 8.2: Reference Trees

Reference trees are the principal implementation mechanism proposed in this thesis. Although the VIM virtual machine could be used on any kind of computer system, reference trees only make sense on certain architectures, notably those discussed in Section 4.1. Then again, on these architectures, reference trees could be used to support an environment quite different from VIM.

The principal attractions of the reference tree mechanism are its simplicity and completeness: *simplicity* in that the various machinations of the reference tree scheme, although not necessarily intuitively obvious, are nevertheless uncomplicated; *completeness* in that all essential object management functions (support for side effects, multiple copies, garbage collection) can be handled easily. The principal liabilities of the reference tree scheme are in the areas of reliability and, possibly, efficiency. The efficiency of a system using reference trees will be a strong function of the shapes of reference trees that actually arise in the system. More work needs to be done to determine the effect of various scheduling strategies on the shapes in which reference trees grow.

To the extent to which reliability and efficiency problems exist, many of them can be remedied by complicating the basic reference tree scheme. Some such modifications are suggested in Section 5.4 and in forthcoming work by Baker[2]. Ultimately, reference trees may need to be only one of many object management strategies in a multiprocessor system. Reference trees should be especially good for objects known only in certain local areas, a category which probably includes the vast majority of objects. Objects known more or less globally across the system might benefit from being managed using a different strategy.

Finally, the usefulness of reference trees depends heavily on the conformability of a physical network topology to the model given in Section 4.1. Although networks of this sort have much to recommend them, technological progress might ultimately favor a different kind of topology. If a network cannot be viewed as a collection of nodes, each of which has "only a few" neighbors, then the usefulness of reference trees on that network will be greatly diminished.

### 8.3: Performance of Reference Tree Networks

This thesis can hardly be considered a definitive study of the performance of reference tree networks. The performance results reported here are too primitive to form a basis for comparison with Cm* or any other existing system. This is partly because of the paucity of applications studied, partly because of the many improvements that can still be made in the MuNet implementation, and partly because of the large cost, on LSI-11's, of implementing such functions as message input and output, garbage collection, and free storage management. About all that can be said at this point is that in some cases the scalability promises of the architecture have been realized, and in others there is still much work left to do. Many different sets of implementation choices and scheduling strategies remain to be explored. Beyond that, the really important performance characteristic of a system is that obtained in actual use, not that exhibited by toy benchmarks. In order to observe this, the system must be made sufficiently serviceable to attract ordinary users. This implies not only the development of scheduling strategies that are at least adequate, but also of a congenial programming language and system software to support it. Only at this point could a reference tree network be considered truly a going concern.

## 8.4: Additional Directions for Further Research

There is virtually no area touching multiprocessor systems that could not benefit from additional research; herewith a brief list of issues that relate to the material in this thesis. The first is the development of applications for reference tree networks. Frameworks for general-purpose applications such as timesharing and simulation, and parallel algorithms for special purposes such as alpha-beta tree search, or solution of partial differential equations, are both needed.

Work on some of these applications may in turn call into question one of the design decisions underlying the construction of reference tree networks: the invisibility of network topology, interprocessor communication, and monitor operations such as garbage collection. Schemes for making visible these aspects of system operation may help improve the efficiency of certain critical algorithms, although in general such low-level details should remain hidden so as not to distract the programmer.

Another important research area may be opened up by the advent of easily tailored special-purpose VLSI circuits. Ideally, in a system where, say, matrix multiplication is an important operation, it should be possible to hook up and use a special-purpose matrix multiplier without either disrupting operation of the remainder of the system or needing to make extensive software changes. The specification of constraints on the design of both the system and the special-purpose VLSI circuitry so that this can be done in a clean and general fashion can be expected to be an important research question for the future. This can be regarded as a subquestion of the more general question of how best to incorporate special-purpose VLSI into general-purpose systems.

Finally, the multiprocessor systems discussed in this thesis, although they do not depend crucially on the assumption, assume a fairly tight degree of coupling between components. Beyond the merely physical degree of coupling, it is assumed that all elements of the system are under the control of one operating authority, and that physical security of the system is not an issue. Applying the technology developed in this thesis to more of a "distributed computing" context where these assumptions may not be valid leads to whole new research issues involving protection, awareness of physical system structure, and co-ordination of independent authorities for name generation and other functions.

### 8.5: Final Conclusions

The marriage of VIM, reference trees, and some hardware has already produced a system, the MuNet, which is capable of improving its performance by making use of multiple processors. Although much more work remains to be done, there is reason to be optimistic that the MuNet and its descendants will eventually be able to exhibit acceptable performance over a wide range of tasks. The most important feature of these systems, however, will not be their efficiency but their interface with users and programs. The introduction of a parallel environment brings with it a whole new set of concerns for the programmer. Our response to this situation must be to remove as many as possible of the old concerns, leaving the programming job still at a tractable level of complexity. Thus a good system for programming a multiprocessor network must perform as many housekeeping chores (such as communication and garbage collection) as possible automatically, freeing the programmer's mind for other worries.

Whether or not reference tree networks are the wave of the future, the technological arguments favoring use of multiprocessor systems are irresistible. Some methodology for using them, based on the fundamental design considerations that underlie reference tree networks, must be developed.

# REFERENCES

1. Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM*, August 1978.

2. Baker, C., "Reliable Distributed Object Management Schemes," S.M. thesis, Department of Electrical Engineering and Computer Science, M.I.T., in preparation.

3. Baker, H., *Actor Systems for Real-Time Computation*, LCS TR-197, Laboratory for Computer Science, M.I.T., March 1978.

4. Batcher, K.E., "Sorting Networks and their Applications," *Spring Joint Computer Conference*, 1968.

5. Bishop, P., *Computer Systems with a Very Large Address Space and Garbage Collection*, LCS TR-178, Laboratory for Computer Science, M.I.T., May 1977.

6. Bobrow, D., and Wegbreit, B., "A Model and Stack Implementation of Multiple Environments," *Communications of the ACM*, October 1973.

7. Dennis, J., and Misunas, D., "A Preliminary Architecture for a Basic Data-Flow Processor," *Second IEEE Symposium on Computer Architecture*, New York, January 1975.

8. Eswaran, K., *et al.*, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, November 1976.

9. Farber, D.J., "A Ring Network," *Datamation*, February 1975.

10. Farber, D.J., et al., "The Distributed Computing System," *Proceedings of the Seventh Annual IEEE Computer Society International Conference*, February 1973.

11. Gray, J., *et al.*, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," IBM Research Report RJ 1654, September 1975.

12. Greif, I., *Semantics of Communicating Parallel Processes*, MAC TR-154, Project MAC, M.I.T., September 1975.

13. Gula, J., "Operating System Considerations for Multiprocessor Architectures," *Proc. Seventh Texas Conf. on Computing Systems*, November 1978.

14. Gula, J., "A Distributed Operating System for an Object Based Network," S.M. thesis, Department of Electrical Engineering and Computer Science, M.I.T., June 1979.

15. Halstead, R., *Multiple-Processor Implementations of Message-Passing Systems*, LCS TR-198, Laboratory for Computer Science, M.I.T., February 1978.

16. Halstead, R., "Object Management on Distributed Systems," *Proc. Seventh Texas Conf. on Computing Systems*, November 1978.

17. Henderson, D.A., *The Binding Model: A Semantic Base for Modular Programming Systems*, MAC TR-145, Project MAC, M.I.T., February 1975.

18. Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages," A.I. Working Paper 92, Artificial Intelligence Laboratory, M.I.T., April 1976.

19. Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM*, August 1978.

20. Kahn, R.E., "Resource-Sharing, Computer Communication Networks," *Proc. IEEE 60(1)*, November 1972.

21. Kernighan, B., and Ritchie, D., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1978.

22. Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," Massachusetts Computer Associates Technical Report CA-7603-2911, March 1976.

23. Learning Research Group, *Personal Dynamic Media*, Xerox PARC Report SSL76-1, 1976.

24. Liskov, B., *et al.*, "Abstraction Mechanisms in CLU," *Communications of the ACM*, August 1977.

25. McCarthy, J., *et al.*, *LISP 1.5 Programmer's Manual*, M.I.T. Press, Cambridge, Mass., 1962.

26. Metcalfe, R., *Packet Communication*, MAC TR-114, Project MAC, M.I.T., December 1973.

27. Metcalfe, R., and Boggs, D., *Ethernet: Distributed Packet Switching for Local Computer Networks*, Xerox PARC Report CSL75-7, November 1975.

28. Redell, D., *Naming and Protection in Extendible Operating Systems*, MAC TR-140, Project MAC, M.I.T., September 1974.

29. Ritchie, D., and Thompson, K., "The UNIX Time-Sharing System," *Communications of the ACM*, July 1974.

30. Rumbaugh, J., *A Parallel Asynchronous Computer Architecture for Data Flow Programs*, MAC TR-150, Project MAC, M.I.T., May 1975.

31. Saltzer, J., ed., *Introduction to Multics*, MAC TR-123, Project MAC, M.I.T., February 1974.

32. Stearns, R.E., Lewis, P.M., and Rosenkrantz, D.J., "Concurrency Control for Database Systems," *IEEE Symposium on Foundations of Computer Science CH1133-8C*, October 1976.

33. Strachey, C., and Wadsworth, C.P., "Continuations: A Mathematical Semantics for Handling Full Jumps," Technical Monograph PRG-11, Oxford University Computing Laboratory, January 1974.

34. Strovink, E., "Compilation Strategies for Multiprocessor Message-Passing Systems," *Proc. Seventh Texas Conf. on Computing Systems*, November 1978.

35. Strovink, E., "Compilation Strategies for a Multiple Processor Message-Passing System," S.M. thesis, Department of Electrical Engineering and Computer Science, M.I.T., June 1979.

36. Swan, R.J., Fuller, S.H., and Siewiorek, D.P., "Cm* — A Modular, Multi-Microprocessor," *AFIPS Conf. Proc. 46*, 1977.

37. Ward, S., "The MuNet: A Multiprocessor Message-Passing System Architecture," *Proc. Seventh Texas Conf. on Computing Systems*, November 1978.

38. Ward, S., and Halstead, R., "A Syntactic Theory of Message Passing," to appear in *Journal of the ACM*.

39. Ward, S., Halstead, R., Gula, J., Strovink, E., and Baker, C., "MuNet Implementation Notes", internal memoranda, M.I.T. Laboratory for Computer Science, 1978.

40. Wulf, W., and Bell, C.G., "C.mmp — A Multi-Mini-Processor," *AFIPS Conference Proceedings*, Fall 1972.

41. Wulf, W., et al., "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications of the ACM*, June 1974.

# Appendix A: The MuNet Virtual Machine

The virtual machine supported by the MuNet is in some ways an elaboration, and in other ways a subset, of the VIM machine discussed in Chapters 2 and 3 of this thesis. The differences reflect the pragmatics of optimizing use of the MuNet hardware (a collection of LSI-11 microprocessors), and of decisions intended to expedite the construction of the initial implementation. The MuNet also bears the scars of having been produced by an evolutionary process; several aspects of the MuNet would be constituted differently in any re-implementation. The purpose of this appendix is to document the current MuNet virtual machine, and give some examples of both good and bad ideas in reference tree network architecture. The appendix is not designed to be self-contained, but should be intelligible in combination with Chapters 2 and 3 of this thesis. Further details about the MuNet exist as internal implementation notes[39].

## A.1: Object References

On the MuNet, an object reference occupies a 16-bit word.* Ordinary object references are pointers to words in memory and hence, on the LSI-11, are always even numbers. This fact opens the door for the use of odd numbers as special "reserved" object references. An odd number appearing where a reference is expected is not a reference to any particular object, but is a distinguishable entity, differentiable from all other odd numbers and from all ordinary object

*This is a local name, valid only on the processor where the reference exists; it does not imply any limitation, such as $2^{16}$, on the number of objects that could exist in the network. It is thus incorrect to say that the MuNet has an "address space" of only 16 bits. Taking into consideration the size allotted in the MuNet for global names of objects, the MuNet "address space" can be considered as large as 40 bits.

references. In certain contexts, specific odd numbers have special meanings to the system.

As in VIM, each ordinary MuNet object reference has an associated text, which may be accessed using suitable monitor calls. In addition, a MuNet object reference has other attributes, independent of its text. Since the only facilities that exist for changing an object operate by changing its text, these other attributes must be determined when an object is created, and cannot be changed thereafter. Every MuNet object has two such attributes. The first is its *type*, which in the MuNet is a separate property of an object, not simply a distinguished reference within its text, as it is in VIM (see Figure 2.3). It follows that every MuNet object has a type, whether or not the object is ever used as a target object in an event, and that an object's type must be specified when it is created.

Experience with the MuNet shows that this is an inferior way of handling types. Not only does it rob the user of the power to change an object's type, but the special treatment for types complicates various sections of monitor code and increases the space and time overhead for object management. Finally, it forces objects to have a type even where (as in the case of purely "data" objects) the VIM type concept is not exactly what is desired, leading to further wastage of time and space.

The MuNet distinguishes several classes of objects according to their types. *General objects* are most like the target object shown in Figure 2.3; the type of a general object is a reference to another object. This object, however, cannot be *another general* object, but must be a *type object*, whose type is the reserved integer 5. If the "closure" aspect of general objects is not needed, a target object may be a *function object*, whose type is the reserved integer 3. When a target object is a function object, the machine code to be executed is found directly in the text of the function object. Thus a *function object* is analogous to a VIM target object whose type is a reference to itself. Finally, the reserved integer 7 may be used as an object's type to denote a data object which should never be used as a target object. No other odd integer may be used as a type; these other numbers are reserved for use by the implementation to distinguish objects from events and from other entities used internally.

The other permanent attribute of an object is a Boolean *device flag*. If an object's device flag is set, the text of the object cannot be moved from the location where the object was created. Therefore, an event that requests access to the text of such a "device object" is forced to execute on the processor where that text is — the text cannot be moved to where the event is. Device objects are intended to model peripheral devices which exist on particular processors. An event whose execution directly involves transfers of data to or from a peripheral must execute on the processor where the peripheral is. If the target object used for final transactions with the peripheral is installed as a "device" on the appropriate processor, this outcome will be ensured (since **gtext** access to the target object is requested in the course of beginning execution of an event). A mechanism for creating device objects will be discussed later.

Although devices are quite a "dirty" mechanism, they are extremely useful any time it is desired to force execution of a particular event in a particular place. This is the case not only when interacting with peripherals, but also for performance monitoring and other kinds of intervention in the activities of the MuNet.

## A.2: Object Texts

An object text within a MuNet processor is represented as a series of words in memory. A text begins with a header word, followed by a sequence of *r* words containing object references, followed by *w* words containing uninterpreted binary data (see Figure A.1). The low-order byte of the header word contains the quantity *r*, while the high-order byte contains the quantity *r+w*. Since this quantity must be representable in a byte, the length of object texts may not exceed 255 words. Larger aggregates of data must be represented as several objects joined together using object references.

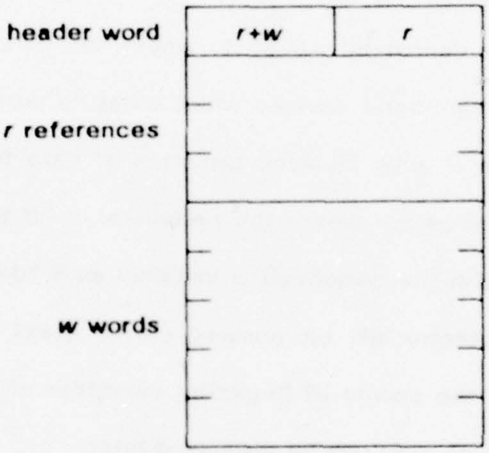


*Figure A.1: Format of MuNet object texts*

If an object is a function or type object, it is presumed to contain executable, position-independent PDP-11 machine code starting at the beginning of the "binary words" portion of its text. Control will be transferred to the beginning of this code whenever the object's code is invoked in executing an event. The ability to include references along with actual machine code in an executable text provides a useful way of making available references (*e.g.*, to other parts of the program) which may be needed during event execution.

### A.3: Events

An event has the same format as an object text: it begins with a header word, followed by a series of references (of which the first is viewed as the reference to the target object), followed by a (possibly null) series of binary words. When an event is to be executed, a pointer to the event is placed in the PDP-11 register **r0**. A pointer to the text of the *target* object is then placed in **r1** (note this is *not* necessarily the text that contains the machine code to be executed). Next, control is transferred to the executable code associated with the target object.

While executing, an event can call on monitor services via **jsr** or **jmp** instructions through transfer vectors at absolute locations in low core. The calls available will be enumerated presently; some further conceptual differences between VIM and the MuNet should be discussed first.

In VIM, every event is created by a **newev** request, eventually executes successfully, and then becomes inactive. If an event causes additional events to further the progress of a computation, it must create each such event by means of another call to **newev**. It is possible to use this mode of operation when programming the MuNet, but for the sake of efficiency, an alternative has been made available.

Many events cause exactly one event to carry on the computation; *i.e.*, in most cases, the thread of event causality is basically linear, with relatively few forks and joins. In this situation, it is desirable to *reuse* the storage allocated to the current event, rather than go through the overhead of throwing it away and allocating a new event. Thus if execution of an event on the MuNet ends with a

**nextev** request, rather than a **done**, the current event is not thrown away, but remains active and is enqueued for re-execution. Unless it was the user's intention to create an endless loop, he will presumably have modified the contents of the event so that it now calls for the next step in the computation; *i.e.*, he will have written into it what would otherwise have been written into an event created using **newev**, had that alternative been adopted.

A few additional comments about the **nextev** mechanism are appropriate. First, writing into the current event must be considered a side effect (in the sense that it would set the blackboard-interpreter side-effect flag $\sigma$ to true), and consequently must be done after all resource requests have been made — if an aborted event has been modified, re-execution of the event is not likely to yield correct results. Second, the **nextev** mechanism is not inconsistent with forks — it is entirely reasonable to create several new events using **newev** and also reuse the current event by calling **nextev**. Third, an **evxpnd** primitive (described below), analogous to **objxpnd**, is provided for handling situations where reusing the current event may require increasing its size.

Another note regarding events in the MuNet — every event has implicitly associated with it an *operating system object*, accessible via the **getos** and **setos** monitor calls. The idea behind operating system objects is to make available the kinds of capabilities associated with *process objects* in chapter 3 of this thesis; however, none of these functions is currently implemented on the MuNet.

A second difference between VIM and the MuNet is that in VIM, **gtext** and **locktext** requests serve only to reserve access rights to object texts; no values are returned. When an executing event actually accesses data in an object text, it does so by supplying a reference to the object along with identification of a slot in the object text. The implementation is then responsible for finding the text in core and performing the desired access. For the sake of efficiency, MuNet **gtext** and **locktext** requests return a pointer to the relevant text, which may then be accessed directly using PDP-11 indexed addressing modes. This pointer is only good for the remainder of the current event execution, since the object text might be moved before any subsequent execution. Any subsequent execution wishing to access the object will have to issue another **gtext** or **locktext** request in any case, and should use the text pointer returned from that request.

Unfortunately, the **objxpnd** request, which increases the size of the area allocated to hold a text, may need to move the text in order to fulfill its mission. Thus **objxpnd** returns a new text pointer which obsoletes any text pointer previously obtained for the object during the same event execution. The resulting potential for confusion is mitigated somewhat by the infrequency of **objxpnd** requests, but it is fairly clear that, in the presence of appropriate hardware support for accessing texts, a policy of not explicitly releasing text pointers is simpler and cleaner.

A third departure from VIM results from the nature of the LSI-11 hardware base for the MuNet. In VIM, the legality of various operations (such as aborting an event) is contingent on the setting of the side-effect flag $\sigma$, indicating whether any side effects have been performed by the event in question. The lack of memory management hardware on the MuNet LSI-11's precludes any attempt to maintain such a flag automatically. It would be possible to construct a monitor call setting the side-effect flag — the user could then invoke this call upon performing a side effect. However, this approach was not taken, and in fact no side-effect flag is explicitly maintained for events in the MuNet. This means that the state of an event's side-effect flag cannot be used to determine the legality of aborting it. MuNet programs should still follow the same rules as if the side-effect flag were maintained and checked, though; thus it is still unwise, for example, to make a

**gtext** request after having performed a side effect. The MuNet monitor cannot signal an error if this is done, since it has no knowledge of whether a side effect has been performed. Rather, this discipline must be followed by the user, or he risks obtaining incorrect results.

The need to be able to abort events is as real on the MuNet, however, as it is in VIM. Since there is no explicit side-effect flag on the MuNet to indicate whether aborting an event is permissible, *events on the MuNet can only be aborted at places where it can be inferred that the side-effect flag, had it been maintained, would be false.* This inference can be made any time an event makes a request which, according to the rules of VIM, is only legal when the side-effect flag is false. Thus events *can* be aborted *only* while trying to make such requests.

### A.4: Monitor Calls

A listing of MuNet system calls follows. In each case, a schematic form of the system call (*e.g.*, **gtype(***ref***) returns(***ref***)**) is given, along with the actual PDP-11 calling sequence and a description of the effect of the call.

**gtype(***ref***) returns(***type***)**:

```
                mov         @ref,type
```

returns (moved into *type*) the type of the object referenced by *ref*.

**gtext(***ref***) returns(***textptr***)**:

```
                mov         ref,r0
                jsr         pc,@#gtext_v
                mov         r1,textptr
```

obtains sharable (read-only) access to the text of the object referred to by *ref*, and returns a pointer to that text. **gtext** may not be called following a side effect.

**locktext(*ref*) returns(*textptr*)**:

```
                    mov         ref,r0
                    jsr         pc,@#locktext_v
                    mov         r1,textptr
```

like **gtext** in every respect, except that it obtains non-shared (read/write) access to the text in question.

**newobj(*type*,*size*) returns(*ref*,*textptr*)**:

```
                    mov         type,r0
                    mov         size,r1
                    jsr         pc,@#newobj_v
                    mov         r0,ref
                    mov         r1,textptr
```

creates an object with type *type* and *size* words of text (not counting the header word). A reference to the newly created object is returned, along with a pointer to the new object's text. **newobj** may not be called following a side effect.

**objxpnd(*ref*,*size*) returns(*textptr*)**:

```
                    mov         ref,r0
                    mov         size,r1
                    jsr         pc,@#objxpnd_v
                    mov         r1,textptr
```

ensures that the area allocated to storage of the text of the object referred to by *ref* is at least large enough to support a text of length *size* words (not counting the header word). If the area allocated to the object text is not large enough, the text is moved to a new, suitably large location. The *textptr* returned is a pointer to the text *after* any required relocation, and supersedes any *textptr* previously obtained for the same object during the same event execution.* On the presumption that a caller of **objxpnd** intends to modify at least the header word of the text, **objxpnd** performs an implicit **locktext** on the object. **objxpnd** may not be called following a side effect. However, since **objxpnd** does not actually modify the text, calling it is *not* considered a side effect.

*A better approach would be to have **objxpnd** never return a *textptr*, but instead abort the requesting event if the text is moved. The text having been moved to a more spacious location, the **objxpnd** would presumably succeed upon the next execution of the event. Aborting the event when a text is moved, however, forces all *textptr*s used by the event to be re-obtained, thus avoiding the possibility of needing to discard a *textptr* because of a subsequent **objxpnd**, and the hazards of negligently continuing to use such a *textptr*.

**newev(***nwords***) returns(***evptr***)**:

```
mov     nwords,r1
jsr     pc,@#newev_v
mov     r1,evptr
```

creates a new event *nwords* words long and returns a pointer to it in *evptr*. The new event will be released for execution only if execution of the current event terminates successfully. As with a **newobj** request, a **newev** request may not be made following a side effect.

**evxpnd(***nwords***) returns(***evptr***)**:

```
mov     nwords,r1
jsr     pc,@#evxpnd_v
mov     r1,evptr
```

**evxpnd** is like **objxpnd**, except that it operates on the currently executing event rather than on an object text. **evxpnd** is called when reuse of the current event is contemplated, and assurance is desired that at least *nwords* words of event space are available to hold references and binary words. Like **objxpnd**, **evxpnd** is not considered a side effect (does not actually modify the contents of the event), may not be called following a side effect, and may relocate the event to a more spacious location. Thus after a call to **evxpnd**, the returned *evptr* should be used in lieu of any previously obtained pointer to the current event.

**nextev()**:

```
mov     @#nextev_v,pc
```

terminates (successfully) execution of the current event and re-enqueues it on the event list. Any other events or objects created are also formally installed.

**done()**:

```
rts     pc
```

like **nextev**, but does not re-enqueue the current event. Because of its use of a return address on the processor stack, the stack should be popped back to its state when execution of the current event began before **done** is invoked.

**getos() returns(*osref*):**

```
jsr     pc,@#getos_v
mov     r0,osref
```

returns a reference to the operating system object associated with the current event. This request can never cause an event to be aborted and may be executed at any time.

**setos(*osref*):**

```
mov     osref,r0
jsr     pc,@#setos_v
```

changes the operating system object of the currently *executing* event to *osref*. This is considered a side effect and should not be done if the event might subsequently be aborted. Note that the operating system object for any newly created events (made using **newev**) will be that operating system object in effect at the *end* of the execution of the creating event.

A few other monitor calls exist also, allowing access to internal monitor variables, interaction with the UNIX development environment, and implementation of other specialized functions. These calls are not intended for use by ordinary users, but aid in the construction of special actors performing various system initialization and monitoring functions.

### A.5: Special Objects

When the MuNet is in normal operation, two kinds of special objects exist within it: *processor objects* and a *system object*. There is one system object for the entire MuNet, which may be invoked, as a target object, to obtain information and perform functions of system-wide significance. Among the kinds of events that the MuNet system object will respond to are (using *S* to denote a reference to the system object)

(*S* 1 *C*): causes the event (*C X*), where *X* is a reference to an array of references to all the processor objects.

(*S* 3 *C*): causes one event (*C P*) for each processor object *P*.

Other calls to the MuNet system object allow I/O to the UNIX console from which the MuNet was invoked, and provide mechanism for processor objects to inform the system object of their existence.

In a fully operational MuNet, there is one processor object for each physical processor. Each processor object is a "device" installed on the processor it corresponds to. Among the kinds of events that a processor object *P* will respond to are

(*P* 1 *X C*): causes the event (*C Y*) where *Y* is a copy of the object *X* (same type, same text contents) installed as a device on the processor whose processor object *P* is.

(*P* 3 *C*): causes the event (*C X*) where *X* is a new object whose text contains various pieces of status information about the processor associated with *P*.

### A.6: Summary

This appendix has given an overview of the virtual machine supported by the current MuNet implementation. Generally speaking, the VIM virtual machine described in the text of the thesis should be regarded as superior to the MuNet virtual machine where their features differ; the purpose of describing the MuNet virtual machine here has been to expose some of the influences on and alternatives to VIM, and to show the elaboration of some of the VIM concepts down to a more concrete and practical level.

## Appendix B: Correctness of the Membership Protocol

The purpose of this appendix is to demonstrate that the reference tree membership protocol described in Section 5.3 performs the functions claimed, *viz.*, that it prevents reference trees from becoming disconnected, prevents cycles from forming in them, and is not prey to any other sort of error condition. As a side effect, perhaps the demonstration will shed a little additional light on the workings of the reference tree mechanisms.

The membership protocol is specified by the list of state transitions in Table 5.9, plus several restrictions not explicitly stated in the table. In this appendix, it will be convenient to refer to various of the state transition rules in Table 5.9. The notation

*old-state : message-received / message-sent : new-state*

will be used, so, for example, "X:R+/L+:S" refers to a transition that happens when a processor in state X receives an R+ message, changing to state **S** and emitting an L+ message back to the sender of the R+ message. Spontaneous transitions have a null *message-received* field, and a null *message-sent* field indicates a transition not accompanied by any output. Thus **X:/:N** denotes a spontaneous transition from state X to state **N** which results in no output.

There are two kinds of important restrictions on the applicability of the spontaneous transitions listed in Table 5.9. One applies only to the transition **M:/-:X?**, by which a processor leaves the reference tree. A processor is allowed to make this *transition only if* its state for every other link attached to it is either **N** or **N?**. This means, among other things, that the processor making the transition must be a leaf node of the reference tree. At the time when the transition is made, the processor's other link states of **N** and **N**? must be changed to X and X?, respectively.

The other kind of restriction applies to the transitions **N**:/:X, **N**?:/:X?, X:/:**N**, and X?:/:**N**?. The first two of these exist only to represent the state changes on other links that are associated with a processor leaving the reference tree, as discussed in the previous paragraph, and should not be used at any other time. Thus, although the transitions *look* spontaneous, they are in fact only used when other activities on the processor cause it to leave the reference tree. Similarly, the transitions X:/:**N** and X?:/:**N**? exist only to handle state changes that must occur when a processor *joins* a reference tree. Since it is always the receipt of a message that causes a processor to join a reference tree, there is a sense in which these two transitions are never "spontaneous."

Showing the correctness of the membership protocol divides into two tasks: determining some local properties of the protocol as it applies to a microcosm of just two nodes and one link, and generalizing from these to prove global properties of entire reference trees. An assumption that we shall make throughout is that only one thing (a spontaneous transition or absorption of a message) happens at a time. In practice, this means only that each processor must act as an arbiter among messages arriving at it over different links, and process them sequentially. The need to ensure more global sequentiality can be circumvented by noting that the effects of state changes at processors are strictly local. Thus simultaneous events at different processors can be *considered* to have happened sequentially, in any order, for the purposes of this appendix.

## B.1: Local Properties of the Membership Protocol

Local properties of the membership protocol are the following:

- closure — all configurations reachable from any possible starting configuration should be possible to handle following the rules of the protocol; no processor should ever receive a message that it cannot handle in its current state.

- consistency — all quiescent configurations (*i.e.*, configurations in which no messages are in transit) reachable from any possible starting configuration should show the desired relationship between processor states; for example, we would not like to see a quiescent configuration in which two processors each thought they were masters of the same link.

From our local examination of the membership protocol we should also like to abstract a few properties useful in showing global properties of the protocol. These properties deal with the relationship of the membership status (*i.e.*, in or out of a reference tree) of pairs of adjacent nodes to the membership status of the link connecting them.

As discussed in Section 5.3, every node, or processor, has (conceptually) a *processor state* for each reference tree. This state can be either "in the reference tree" or "not in the reference tree." The processor state is manifested in the values of the processor's link state for each link attached to the processor, such that if a processor is "not in" a particular reference tree, each of its link states for that tree will be either X or X?. Conversely, if the processor is "in" the tree, each of its link states for that tree will be some state other than X or X?. Thus it is easy to determine whether a processor is a member of a particular reference tree; examination of any of that processor's link states for that tree will yield the answer.

It is more difficult for a processor to find out whether a *link* that connects to it is a member of a particular reference tree. In general, knowing the link state of that processor for that link is not sufficient for an unambiguous determination (exceptions are states **M** and **N!**, which always mean the link is in the reference tree, and states X, X?, **N**, and **N?**, which always mean it is not). In fact, even knowing the link state at *both* ends of the link will not always be enough to decide whether the link is in the reference tree. In addition to the link states, the queues[*] of pending messages (messages sent but not yet received) on the link must be taken into consideration to determine with certainty whether a particular link is a member of a particular reference tree.

Thus in order to completely specify the condition of a link between processors **A** and **B**, for a particular object's reference tree, four pieces of information are needed: the link states of processors **A** and **B** for that link with respect to that object, and the queues of messages[†] sent but not yet received from **A** to **B** and from **B** to **A**. In our discussions, we shall represent the condition of a link as follows:

**(A-*state* A-*to*-B-*queue* B-*state* B-*to*-A-*queue*)**

Thus the meaning of the link condition

**(M? (R+ A-) N?1 ())**

[*] The reader is reminded that messages on a link must be received in the same order as they were sent, in order for the protocols to work properly. Consequently, the notion of a "queue" of messages on each link is an accurate one.

[†] These messages may be any of R+, L+, L-, +, -, A+, A-, or *LN*; in other words, any of the messages mentioned in Table 5.9.

is that the link state of processor **A** for this link is **M?** and the link state of processor **B** is **N?1**, that an A– followed by an R+ message has been sent by **A** to **B** but neither has yet been received by **B**, and that no unreceived messages have been sent by **B** to **A**. Since interprocessor links in reference tree networks are symmetrical, the link condition

(**A**-*state* **A**-*to*-**B**-*queue* **B**-*state* **B**-*to*-**A**-*queue*)

is for all practical purposes equivalent to

(**B**-*state* **B**-*to*-**A**-*queue* **A**-*state* **A**-*to*-**B**-*queue*)

in that any comment that may be made about one of them is valid, *mutatis mutandis*, about the other. In this appendix, we shall follow the practice of allowing one of these versions to stand for both, rather than taking the extra space to list both versions of all asymmetrical link conditions.

Obviously an infinite number of different link conditions are possible, since each message queue could be an arbitrarily long string. Therefore, it would seem necessary to formulate a rule, or algorithm, that could be applied to a particular link condition to determine whether that link should be considered a member of the relevant *reference tree*. Unfortunately, it is troublesome to design a concise rule for this purpose, so an alternative route is taken in this appendix.

At the birth of a system, when we are entitled to believe that no inconsistencies exist and no messages are already flowing, every link will be in one of the following conditions:

(**X** () **X** ()) (**X** () **N** ()) (**N** () **N** ()) (**L** () **L** ()) (**S** () **M** ())

Thus the only link conditions that will really be of interest to us are those that can be reached from this initial set by following the state transition rules spelled out in Table 5.9. This closed subset of the entire set of link conditions is the only one in which we are concerned with being able to determine membership of links in reference trees. Although this subset is still infinite, we can group its elements into a finite number of useful categories by making one simple observation. Since sending a local name never has any effect on the sender's state, and receiving any series of consecutive repetitions of a local name has the same effect on the receiver's state as receiving just one, we can use the symbol *LN* in a message queue to denote not just a single local name, but any sequence of one or more repetitions of the local name. As a result, transitions involving the receipt of a local name by a processor must be performed in each of two ways: by removing the *LN* from the input queue, the proper action if the *LN* represented *only one local name*, and by leaving the *LN* in the input queue, the proper action if it represented *several repetitions of the local name.*

With the simplification of letting one instance of the symbol *LN* represent any number of repetitions of the local name, the number of link conditions accessible from our initial set becomes finite. A mechanically generated list of all these accessible link conditions is given below in Table B.1.

In the table, each link condition is given with a reference number, an indication of whether the link in question should be considered a member of the reference tree, and the reference numbers of all link conditions that could be successors of the current one (*i.e.*, which could be derived from it by applying some transition in Table 5.9). The status of the link as being in or out of the reference tree has been chosen so that

- the status *remains* invariant *from* a link condition to *any* successor *except* that
    - when one of the processors undergoes one of the transitions **X**:R+/L+:**S** or **X**?:*LN*/:**N**!, the link (which should not previously have been in the tree) joins the tree, and
    - when the link and both processors are in the tree and one of the processors undergoes the transition **M**:/-:**X**?, leaving the tree, the link also leaves the tree, and
- the link is *not* in the tree for the quiescent conditions

  **(X () X ())   (X () N ())   (N () N ())   (L () L ())**

  but *is* in the tree for the quiescent condition **(S () M ())**.

That the assignment of link status (in or out of the reference tree) to link conditions in Table B.1 has these properties has been verified mechanically; the reader is invited to peruse the table to convince himself of this and, perhaps, gain further insight into our approach. The reader should bear in mind, when following the relationships between link conditions and their successors, that only one of the link conditions belonging to each symmetrical pair is listed. Thus the change from a link condition to its successor in Table B.1 may involve not only a state transition at a processor, but also a reversal of the roles of processors **A** and **B**.

*Table B.1: Membership Protocol Link Conditions*

| *Number* | *Link Condition* | *Is Link In Tree?* | *Successors* |
|---|---|---|---|
| 1. | (L () L ()) | *No* | 2, 4 |
| 2. | (L () L (*LN*)) | *No* | 1, 2, 3, 5, 15 |
| 3. | (L (*LN*) L (*LN*)) | *No* | 2, 3, 8 |
| 4. | (L () L? (-)) | *No* | 6, 15, 17, 225 |
| 5. | (L () L? (- *LN*)) | *No* | 4, 5, 7, 8, 18 |
| 6. | (L () L? (*LN* -)) | *No* | 6, 16, 19, 226 |
| 7. | (L () L? (*LN* - *LN*)) | *No* | 6, 7, 9, 20 |
| 8. | (L (*LN*) L? (- *LN*)) | *No* | 5, 8, 9, 15, 21 |
| 9. | (L (*LN*) L? (*LN* - *LN*)) | *No* | 7, 9, 16, 22 |
| 10. | (L (L-) **M?** ()) | *No* | 1, 11, 13, 27 |
| 11. | (L (L-) **M?** (*LN*)) | *No* | 2, 10, 11, 14, 28 |
| 12. | (L (+) **M?** (*LN*)) | *No* | 2, 12, 29, 64, 66 |
| 13. | (L (*LN* L-) **M?** ()) | *No* | 2, 13, 14, 30 |
| 14. | (L (*LN* L-) **M?** (*LN*)) | *No* | 3, 13, 14, 31 |
| 15. | (L? (-) L (*LN*)) | *No* | 4, 15, 16, 18, 57 |
| 16. | (L? (*LN* -) L (*LN*)) | *No* | 6, 16, 23, 61 |
| 17. | (L? (-) L? (-)) | *No* | 19, 55 |
| 18. | (L? (-) L? (- *LN*)) | *No* | 17, 18, 20, 23, 56 |
| 19. | (L? (-) L? (*LN* -)) | *No* | 19, 24, 58, 227 |
| 20. | (L? (-) L? (*LN* - *LN*)) | *No* | 19, 20, 25, 59 |
| 21. | (L? (- *LN*) L? (- *LN*)) | *No* | 18, 21, 22 |
| 22. | (L? (- *LN*) L? (*LN* - *LN*)) | *No* | 20, 22, 23, 26 |
| 23. | (L? (*LN* -) L? (- *LN*)) | *No* | 19, 23, 25, 60 |
| 24. | (L? (*LN* -) L? (*LN* -)) | *No* | 24, 62 |
| 25. | (L? (*LN* -) L? (*LN* - *LN*)) | *No* | 24, 25, 63 |
| 26. | (L? (*LN* - *LN*) L? (*LN* - *LN*)) | *No* | 25, 26 |
| 27. | (L? (- L-) **M?** ()) | *No* | 4, 28, 32 |
| 28. | (L? (- L-) **M?** (*LN*)) | *No* | 15, 27, 28, 33 |
| 29. | (L? (- +) **M?** (*LN*)) | *No* | 15, 29, 34, 67 |
| 30. | (L? (- *LN* L-) **M?** ()) | *No* | 5, 31, 35 |
| 31. | (L? (- *LN* L-) **M?** (*LN*)) | *No* | 8, 30, 31, 36 |
| 32. | (L? (*LN* - L-) **M?** ()) | *No* | 6, 32, 33 |
| 33. | (L? (*LN* - L-) **M?** (*LN*)) | *No* | 16, 32, 33 |
| 34. | (L? (*LN* - +) **M?** (*LN*)) | *No* | 16, 34, 69 |
| 35. | (L? (*LN* - *LN* L-) **M?** ()) | *No* | 7, 35, 36 |
| 36. | (L? (*LN* - *LN* L-) **M?** (*LN*)) | *No* | 9, 35, 36 |
| 37. | (L? () **M?1** (+ A- -)) | *No* | 42, 126, 236 |
| 38. | (L? () **M?1** (+ A- - *LN*)) | *No* | 37, 38, 43, 47 |
| 39. | (L? () **M?1** (+ A- *LN*)) | *No* | 39, 44, 48, 124 |
| 40. | (L? () **M?1** (+ A- *LN* -)) | *No* | 45, 49, 237 |
| 41. | (L? () **M?1** (+ A- *LN* - *LN*)) | *No* | 40, 41, 46, 50 |
| 42. | (L? () **M?1** (*LN* + A- -)) | *No* | 42, 153, 238 |
| 43. | (L? () **M?1** (*LN* + A- - *LN*)) | *No* | 42, 43, 51 |
| 44. | (L? () **M?1** (*LN* + A- *LN*)) | *No* | 44, 52, 151 |
| 45. | (L? () **M?1** (*LN* + A- *LN* -)) | *No* | 45, 53, 239 |
| 46. | (L? () **M?1** (*LN* + A- *LN* - *LN*)) | *No* | 45, 46, 54 |
| 47. | (L? (*LN*) **M?1** (+ A- - *LN*)) | *No* | 38, 47, 51, 126 |
| 48. | (L? (*LN*) **M?1** (+ A- *LN*)) | *No* | 39, 48, 52, 125 |
| 49. | (L? (*LN*) **M?1** (+ A- *LN* -)) | *No* | 40, 49, 53, 240 |
| 50. | (L? (*LN*) **M?1** (+ A- *LN* - *LN*)) | *No* | 41, 49, 50, 54 |

*Table B.1:* ***Membership Protocol Link Conditions***

| *Number* | *Link Condition* | *Is Link In Tree?* | *Successors* |
|---|---|---|---|
| 51. | (L? (*LN*) **M?1** (*LN* + A- - *LN*)) | *No* | 43, 51, 153 |
| 52. | (L? (*LN*) **M?1** (*LN* + A- *LN*)) | *No* | 44, 52, 152 |
| 53. | (L? (*LN*) **M?1** (*LN* + A- *LN* -)) | *No* | 45, 53, 241 |
| 54. | (L? (*LN*) **M?1** (*LN* + A- *LN* - *LN*)) | *No* | 46, 53, 54 |
| 55. | (L? () **N?1** (A- -)) | *No* | 37, 227, 252 |
| 56. | (L? () **N?1** (A- - *LN*)) | *No* | 38, 55, 56, 60 |
| 57. | (L? () **N?1** (A- *LN*)) | *No* | 39, 57, 61, 225 |
| 58. | (L? () **N?1** (A- *LN* -)) | *No* | 40, 62, 253 |
| 59. | (L? () **N?1** (A- *LN* - *LN*)) | *No* | 41, 58, 59, 63 |
| 60. | (L? (*LN*) **N?1** (A- - *LN*)) | *No* | 47, 56, 60, 227 |
| 61. | (L? (*LN*) **N?1** (A- *LN*)) | *No* | 48, 57, 61, 226 |
| 62. | (L? (*LN*) **N?1** (A- *LN* -)) | *No* | 49, 58, 62, 254 |
| 63. | (L? (*LN*) **N?1** (A- *LN* - *LN*)) | *No* | 50, 59, 62, 63 |
| 64. | (**M?** () L (+)) | *No* | 1, 12, 65, 67 |
| 65. | (**M?** () L (*LN* +)) | *No* | 2, 65, 66, 68 |
| 66. | (**M?** (*LN*) L (*LN* +)) | *No* | 3, 65, 66, 71 |
| 67. | (**M?** () L? (- +)) | *No* | 4, 29, 69 |
| 68. | (**M?** () L? (- *LN* +)) | *No* | 5, 70, 71 |
| 69. | (**M?** () L? (*LN* - +)) | *No* | 6, 34, 69 |
| 70. | (**M?** () L? (*LN* - *LN* +)) | *No* | 7, 70, 72 |
| 71. | (**M?** (*LN*) L? (- *LN* +)) | *No* | 8, 68, 71, 72 |
| 72. | (**M?** (*LN*) L? (*LN* - *LN* +)) | *No* | 9, 70, 72 |
| 73. | (**M?** (+) **M?** (+)) | *No* | 64, 74 |
| 74. | (**M?** (+) **M?** (*LN* +)) | *No* | 12, 65, 74, 75 |
| 75. | (**M?** (*LN* +) **M?** (*LN* +)) | *No* | 66, 75 |
| 76. | (**M?1** () L (A+)) | *Yes* | 82, 88, 100, 301 |
| 77. | (**M?1** () L (A+ *LN*)) | *Yes* | 76, 77, 83, 89, 101 |
| 78. | (**M?1** () L (L- A-)) | *No* | 10, 84, 90, 102 |
| 79. | (**M?1** () L (L- A- *LN*)) | *No* | 78, 79, 85, 91, 103 |
| 80. | (**M?1** () L (+ A-)) | *No* | 64, 86, 92, 104 |
| 81. | (**M?1** () L (+ A- *LN*)) | *No* | 80, 81, 87, 93, 105 |
| 82. | (**M?1** () L (*LN* A+)) | *Yes* | 82, 94, 106, 302 |
| 83. | (**M?1** () L (*LN* A+ *LN*)) | *Yes* | 82, 83, 95, 107 |
| 84. | (**M?1** () L (*LN* L- A-)) | *No* | 13, 84, 96, 108 |
| 85. | (**M?1** () L (*LN* L- A- *LN*)) | *No* | 84, 85, 97, 109 |
| 86. | (**M?1** () L (*LN* + A-)) | *No* | 65, 86, 98, 110 |
| 87. | (**M?1** () L (*LN* + A- *LN*)) | *No* | 86, 87, 99, 111 |
| 88. | (**M?1** (*LN*) L (A+)) | *Yes* | 76, 88, 94, 127, 303 |
| 89. | (**M?1** (*LN*) L (A+ *LN*)) | *Yes* | 77, 83, 89, 95, 128 |
| 90. | (**M?1** (*LN*) L (L- A-)) | *No* | 11, 78, 90, 96, 129 |
| 91. | (**M?1** (*LN*) L (L- A- *LN*)) | *No* | 79, 90, 91, 97, 130 |
| 92. | (**M?1** (*LN*) L (+ A-)) | *No* | 12, 80, 92, 98, 131 |
| 93. | (**M?1** (*LN*) L (+ A- *LN*)) | *No* | 81, 92, 93, 99, 132 |
| 94. | (**M?1** (*LN*) L (*LN* A+)) | *Yes* | 82, 94, 133, 304 |
| 95. | (**M?1** (*LN*) L (*LN* A+ *LN*)) | *Yes* | 83, 94, 95, 134 |
| 96. | (**M?1** (*LN*) L (*LN* L- A-)) | *No* | 14, 84, 96, 135 |
| 97. | (**M?1** (*LN*) L (*LN* L- A- *LN*)) | *No* | 85, 96, 97, 136 |
| 98. | (**M?1** (*LN*) L (*LN* + A-)) | *No* | 66, 86, 98, 137 |
| 99. | (**M?1** (*LN*) L (*LN* + A- *LN*)) | *No* | 87, 98, 99, 138 |
| 100. | (**M?1** () L? (- A+)) | *Yes* | 112, 127, 310 |

*Table B.1: Membership Protocol Link Conditions*

| *Number* | *Link Condition* | *Is Link In Tree?* | *Successors* |
|---|---|---|---|
| 101. | (**M?1** () L? (- A+ *LN*)) | *Yes* | 100, 101, 113, 128 |
| 102. | (**M?1** () L? (- L- A-)) | *No* | 27, 114, 129 |
| 103. | (**M?1** () L? (- L- A- *LN*)) | *No* | 102, 103, 115, 130 |
| 104. | (**M?1** () L? (- + A-)) | *No* | 67, 116, 131 |
| 105. | (**M?1** () L? (- + A- *LN*)) | *No* | 104, 105, 117, 132 |
| 106. | (**M?1** () L? (- *LN* A+)) | *Yes* | 118, 133, 311 |
| 107. | (**M?1** () L? (- *LN* A+ *LN*)) | *Yes* | 106, 107, 119, 134 |
| 108. | (**M?1** () L? (- *LN* L- A-)) | *No* | 30, 120, 135 |
| 109. | (**M?1** () L? (- *LN* L- A- *LN*)) | *No* | 108, 109, 121, 136 |
| 110. | (**M?1** () L? (- *LN* + A-)) | *No* | 68, 122, 137 |
| 111. | (**M?1** () L? (- *LN* + A- *LN*)) | *No* | 110, 111, 123, 138 |
| 112. | (**M?1** () L? (*LN* - A+)) | *Yes* | 112, 139, 313 |
| 113. | (**M?1** () L? (*LN* - A+ *LN*)) | *Yes* | 112, 113, 140 |
| 114. | (**M?1** () L? (*LN* - L- A-)) | *No* | 32, 114, 141 |
| 115. | (**M?1** () L? (*LN* - L- A- *LN*)) | *No* | 114, 115, 142 |
| 116. | (**M?1** () L? (*LN* - + A-)) | *No* | 69, 116, 143 |
| 117. | (**M?1** () L? (*LN* - + A- *LN*)) | *No* | 116, 117, 144 |
| 118. | (**M?1** () L? (*LN* - *LN* A+)) | *Yes* | 118, 145, 314 |
| 119. | (**M?1** () L? (*LN* - *LN* A+ *LN*)) | *Yes* | 118, 119, 146 |
| 120. | (**M?1** () L? (*LN* - *LN* L- A-)) | *No* | 35, 120, 147 |
| 121. | (**M?1** () L? (*LN* - *LN* L- A- *LN*)) | *No* | 120, 121, 148 |
| 122. | (**M?1** () L? (*LN* - *LN* + A-)) | *No* | 70, 122, 149 |
| 123. | (**M?1** () L? (*LN* - *LN* + A- *LN*)) | *No* | 122, 123, 150 |
| 124. | (**M?1** (+ A-) L? ()) | *No* | 125, 151, 180 |
| 125. | (**M?1** (+ A-) L? (*LN*)) | *No* | 124, 125, 152, 181 |
| 126. | (**M?1** (+ A- -) L? (*LN*)) | *No* | 37, 126, 153, 200 |
| 127. | (**M?1** (*LN*) L? (- A+)) | *Yes* | 100, 127, 139, 316 |
| 128. | (**M?1** (*LN*) L? (- A+ *LN*)) | *Yes* | 101, 127, 128, 140 |
| 129. | (**M?1** (*LN*) L? (- L- A-)) | *No* | 28, 102, 129, 141 |
| 130. | (**M?1** (*LN*) L? (- L- A- *LN*)) | *No* | 103, 129, 130, 142 |
| 131. | (**M?1** (*LN*) L? (- + A-)) | *No* | 29, 104, 131, 143 |
| 132. | (**M?1** (*LN*) L? (- + A- *LN*)) | *No* | 105, 131, 132, 144 |
| 133. | (**M?1** (*LN*) L? (- *LN* A+)) | *Yes* | 106, 133, 145, 317 |
| 134. | (**M?1** (*LN*) L? (- *LN* A+ *LN*)) | *Yes* | 107, 133, 134, 146 |
| 135. | (**M?1** (*LN*) L? (- *LN* L- A-)) | *No* | 31, 108, 135, 147 |
| 136. | (**M?1** (*LN*) L? (- *LN* L- A- *LN*)) | *No* | 109, 135, 136, 148 |
| 137. | (**M?1** (*LN*) L? (- *LN* + A-)) | *No* | 71, 110, 137, 149 |
| 138. | (**M?1** (*LN*) L? (- *LN* + A- *LN*)) | *No* | 111, 137, 138, 150 |
| 139. | (**M?1** (*LN*) L? (*LN* - A+)) | *Yes* | 112, 139, 319 |
| 140. | (**M?1** (*LN*) L? (*LN* - A+ *LN*)) | *Yes* | 113, 139, 140 |
| 141. | (**M?1** (*LN*) L? (*LN* - L- A-)) | *No* | 33, 114, 141 |
| 142. | (**M?1** (*LN*) L? (*LN* - L- A- *LN*)) | *No* | 115, 141, 142 |
| 143. | (**M?1** (*LN*) L? (*LN* - + A-)) | *No* | 34, 116, 143 |
| 144. | (**M?1** (*LN*) L? (*LN* - + A- *LN*)) | *No* | 117, 143, 144 |
| 145. | (**M?1** (*LN*) L? (*LN* - *LN* A+)) | *Yes* | 118, 145, 320 |
| 146. | (**M?1** (*LN*) L? (*LN* - *LN* A+ *LN*)) | *Yes* | 119, 145, 146 |
| 147. | (**M?1** (*LN*) L? (*LN* - *LN* L- A-)) | *No* | 36, 120, 147 |
| 148. | (**M?1** (*LN*) L? (*LN* - *LN* L- A- *LN*)) | *No* | 121, 147, 148 |
| 149. | (**M?1** (*LN*) L? (*LN* - *LN* + A-)) | *No* | 72, 122, 149 |
| 150. | (**M?1** (*LN*) L? (*LN* - *LN* + A- *LN*)) | *No* | 123, 149, 150 |

*Table B.1: Membership Protocol Link Conditions*

| *Number* | *Link Condition* | *Is Link In Tree?* | *Successors* |
|---|---|---|---|
| 151. | (**M?1** (*LN* + A-) **L?** ()) | *No* | 151, 152, 182 |
| 152. | (**M?1** (*LN* + A-) **L?** (*LN*)) | *No* | 151, 152, 183 |
| 153. | (**M?1** (*LN* + A- -) **L?** (*LN*)) | *No* | 42, 153, 201 |
| 154. | (**M?1** (+) **M?** (A+)) | *Yes* | 76, 158, 162, 357 |
| 155. | (**M?1** (+) **M?** (A+ *LN*)) | *Yes* | 77, 154, 155, 159, 163 |
| 156. | (**M?1** (+) **M?** (+ A-)) | *No* | 73, 80, 160, 164 |
| 157. | (**M?1** (+) **M?** (+ A- *LN*)) | *No* | 81, 156, 157, 161, 165 |
| 158. | (**M?1** (+) **M?** (*LN* A+)) | *Yes* | 82, 158, 166, 358 |
| 159. | (**M?1** (+) **M?** (*LN* A+ *LN*)) | *Yes* | 83, 158, 159, 167 |
| 160. | (**M?1** (+) **M?** (*LN* + A-)) | *No* | 74, 86, 160, 168 |
| 161. | (**M?1** (+) **M?** (*LN* + A- *LN*)) | *No* | 87, 160, 161, 169 |
| 162. | (**M?1** (*LN* +) **M?** (A+)) | *Yes* | 88, 162, 166, 359 |
| 163. | (**M?1** (*LN* +) **M?** (A+ *LN*)) | *Yes* | 89, 162, 163, 167 |
| 164. | (**M?1** (*LN* +) **M?** (+ A-)) | *No* | 74, 92, 164, 168 |
| 165. | (**M?1** (*LN* +) **M?** (+ A- *LN*)) | *No* | 93, 164, 165, 169 |
| 166. | (**M?1** (*LN* +) **M?** (*LN* A+)) | *Yes* | 94, 166, 360 |
| 167. | (**M?1** (*LN* +) **M?** (*LN* A+ *LN*)) | *Yes* | 95, 166, 167 |
| 168. | (**M?1** (*LN* +) **M?** (*LN* + A-)) | *No* | 75, 98, 168 |
| 169. | (**M?1** (*LN* +) **M?** (*LN* + A- *LN*)) | *No* | 99, 168, 169 |
| 170. | (**M?1** (+ A-) **M?1** (+ A-)) | *No* | 156, 172 |
| 171. | (**M?1** (+ A-) **M?1** (+ A- *LN*)) | *No* | 157, 170, 171, 173, 176 |
| 172. | (**M?1** (+ A-) **M?1** (*LN* + A-)) | *No* | 160, 164, 172, 177 |
| 173. | (**M?1** (+ A-) **M?1** (*LN* + A- *LN*)) | *No* | 161, 172, 173, 178 |
| 174. | (**M?1** (+ A- *LN*) **M?1** (+ A- *LN*)) | *No* | 171, 174, 175 |
| 175. | (**M?1** (+ A- *LN*) **M?1** (*LN* + A- *LN*)) | *No* | 173, 175, 176, 179 |
| 176. | (**M?1** (*LN* + A-) **M?1** (+ A- *LN*)) | *No* | 165, 172, 176, 178 |
| 177. | (**M?1** (*LN* + A-) **M?1** (*LN* + A-)) | *No* | 168, 177 |
| 178. | (**M?1** (*LN* + A-) **M?1** (*LN* + A- *LN*)) | *No* | 169, 177, 178 |
| 179. | (**M?1** (*LN* + A- *LN*) **M?1** (*LN* + A- *LN*)) | *No* | 178, 179 |
| 180. | (**M?1** (+) **N** (A-)) | *No* | 78, 156, 182, 210, 222 |
| 181. | (**M?1** (+) **N** (A- *LN*)) | *No* | 79, 157, 180, 181, 183, 211 |
| 182. | (**M?1** (*LN* +) **N** (A-)) | *No* | 90, 164, 182, 212, 223 |
| 183. | (**M?1** (*LN* +) **N** (A- *LN*)) | *No* | 91, 165, 182, 183, 213 |
| 184. | (**M?1** (+ A-) **N!** ()) | *Yes* | 154, 185, 188 |
| 185. | (**M?1** (+ A-) **N!** (*LN*)) | *Yes* | 155, 184, 185, 189 |
| 186. | (**M?1** (+ A- *LN*) **N!** ()) | *Yes* | 184, 186, 187, 190 |
| 187. | (**M?1** (+ A- *LN*) **N!** (*LN*)) | *Yes* | 185, 186, 187, 191 |
| 188. | (**M?1** (*LN* + A-) **N!** ()) | *Yes* | 162, 188, 189 |
| 189. | (**M?1** (*LN* + A-) **N!** (*LN*)) | *Yes* | 163, 188, 189 |
| 190. | (**M?1** (*LN* + A- *LN*) **N!** ()) | *Yes* | 188, 190, 191 |
| 191. | (**M?1** (*LN* + A- *LN*) **N!** (*LN*)) | *Yes* | 189, 190, 191 |
| 192. | (**M?1** (+ A-) **N?** ()) | *No* | 180, 193, 196, 214 |
| 193. | (**M?1** (+ A-) **N?** (*LN*)) | *No* | 181, 192, 193, 197, 215 |
| 194. | (**M?1** (+ A- *LN*) **N?** ()) | *No* | 192, 194, 195, 198, 216 |
| 195. | (**M?1** (+ A- *LN*) **N?** (*LN*)) | *No* | 193, 194, 195, 199, 217 |
| 196. | (**M?1** (*LN* + A-) **N?** ()) | *No* | 182, 196, 197, 218 |
| 197. | (**M?1** (*LN* + A-) **N?** (*LN*)) | *No* | 183, 196, 197, 219 |
| 198. | (**M?1** (*LN* + A- *LN*) **N?** ()) | *No* | 196, 198, 199, 220 |
| 199. | (**M?1** (*LN* + A- *LN*) **N?** (*LN*)) | *No* | 197, 198, 199, 221 |
| 200. | (**M?1** (+ A-) **N?1** (A- *LN*)) | *No* | 171, 181, 200, 201, 236 |

*Table B.1: Membership Protocol Link Conditions*

| *Number* | *Link Condition* | *Is Link In Tree?* | *Successors* |
|---|---|---|---|
| 201. | (**M?1** (*LN* + A-) **N?1** (A- *LN*)) | *No* | 176, 183, 201, 238 |
| 202. | (**M?1** () **S** (L+ A-)) | *Yes* | 204, 206, 265 |
| 203. | (**M?1** () **S** (L+ A- *LN*)) | *Yes* | 202, 203, 205, 207 |
| 204. | (**M?1** () **S** (*LN* L+ A-)) | *Yes* | 204, 208, 267 |
| 205. | (**M?1** () **S** (*LN* L+ A- *LN*)) | *Yes* | 204, 205, 209 |
| 206. | (**M?1** (*LN*) **S** (L+ A-)) | *Yes* | 202, 206, 208, 266 |
| 207. | (**M?1** (*LN*) **S** (L+ A- *LN*)) | *Yes* | 203, 206, 207, 209 |
| 208. | (**M?1** (*LN*) **S** (*LN* L+ A-)) | *Yes* | 204, 208, 268 |
| 209. | (**M?1** (*LN*) **S** (*LN* L+ A- *LN*)) | *Yes* | 205, 208, 209 |
| 210. | (**M?1** (+) **X** (A-)) | *No* | 180, 202, 212, 557 |
| 211. | (**M?1** (+) **X** (A- *LN*)) | *No* | 181, 203, 210, 211, 213 |
| 212. | (**M?1** (*LN* +) **X** (A-)) | *No* | 182, 206, 212, 558 |
| 213. | (**M?1** (*LN* +) **X** (A- *LN*)) | *No* | 183, 207, 212, 213 |
| 214. | (**M?1** (+ A-) **X?** ()) | *No* | 192, 210, 218 |
| 215. | (**M?1** (+ A-) **X?** (*LN*)) | *No* | 193, 211, 214, 215, 219 |
| 216. | (**M?1** (+ A- *LN*) **X?** ()) | *No* | 184, 186, 194, 220 |
| 217. | (**M?1** (+ A- *LN*) **X?** (*LN*)) | *No* | 185, 187, 195, 216, 217, 221 |
| 218. | (**M?1** (*LN* + A-) **X?** ()) | *No* | 196, 212, 218 |
| 219. | (**M?1** (*LN* + A-) **X?** (*LN*)) | *No* | 197, 213, 218, 219 |
| 220. | (**M?1** (*LN* + A- *LN*) **X?** ()) | *No* | 188, 190, 198, 220 |
| 221. | (**M?1** (*LN* + A- *LN*) **X?** (*LN*)) | *No* | 189, 191, 199, 220, 221 |
| 222. | (**N** () **M?** (+)) | *No* | 10, 73, 223, 557 |
| 223. | (**N** () **M?** (*LN* +)) | *No* | 11, 74, 223, 558 |
| 224. | (**N** () **N** ()) | *No* | 222, 559 |
| 225. | (**N?1** (A-) **L?** ()) | *No* | 124, 226, 242 |
| 226. | (**N?1** (A-) **L?** (*LN*)) | *No* | 125, 225, 226, 243 |
| 227. | (**N?1** (A- -) **L?** (*LN*)) | *No* | 55, 126, 227, 253 |
| 228. | (**N?1** () **M?** (A+)) | *Yes* | 154, 232, 265 |
| 229. | (**N?1** () **M?** (A+ *LN*)) | *Yes* | 155, 228, 229, 233 |
| 230. | (**N?1** () **M?** (+ A-)) | *No* | 156, 222, 234 |
| 231. | (**N?1** () **M?** (+ A- *LN*)) | *No* | 157, 230, 231, 235 |
| 232. | (**N?1** () **M?** (*LN* A+)) | *Yes* | 158, 232, 266 |
| 233. | (**N?1** () **M?** (*LN* A+ *LN*)) | *Yes* | 159, 232, 233 |
| 234. | (**N?1** () **M?** (*LN* + A-)) | *No* | 160, 223, 234 |
| 235. | (**N?1** () **M?** (*LN* + A- *LN*)) | *No* | 161, 234, 235 |
| 236. | (**N?1** (A-) **M?1** (+ A-)) | *No* | 170, 180, 230, 238 |
| 237. | (**N?1** (A-) **M?1** (+ A- *LN*)) | *No* | 171, 231, 236, 237, 239 |
| 238. | (**N?1** (A-) **M?1** (*LN* + A-)) | *No* | 172, 182, 234, 238 |
| 239. | (**N?1** (A-) **M?1** (*LN* + A- *LN*)) | *No* | 173, 235, 238, 239 |
| 240. | (**N?1** (A- *LN*) **M?1** (+ A- *LN*)) | *No* | 174, 200, 237, 240, 241 |
| 241. | (**N?1** (A- *LN*) **M?1** (*LN* + A- *LN*)) | *No* | 175, 201, 239, 241 |
| 242. | (**N?1** () **N** (A-)) | *No* | 180, 224, 230, 255 |
| 243. | (**N?1** () **N** (A- *LN*)) | *No* | 181, 231, 242, 243, 256 |
| 244. | (**N?1** (A-) **N!** ()) | *Yes* | 184, 228, 245 |
| 245. | (**N?1** (A-) **N!** (*LN*)) | *Yes* | 185, 229, 244, 245 |
| 246. | (**N?1** (A- *LN*) **N!** ()) | *Yes* | 186, 244, 246, 247 |
| 247. | (**N?1** (A- *LN*) **N!** (*LN*)) | *Yes* | 187, 245, 246, 247 |
| 248. | (**N?1** (A-) **N?** ()) | *No* | 192, 242, 249, 257 |
| 249. | (**N?1** (A-) **N?** (*LN*)) | *No* | 193, 243, 248, 249, 258 |
| 250. | (**N?1** (A- *LN*) **N?** ()) | *No* | 194, 248, 250, 251, 259 |

*Table B.1: Membership Protocol Link Conditions*

| *Number* | *Link Condition* | *Is Link In Tree?* | *Successors* |
|---|---|---|---|
| 251. | (**N?1** (A- *LN*) **N?** (*LN*)) | *No* | 195, 249, 250, 251, 260 |
| 252. | (**N?1** (A-) **N?1** (A-)) | *No* | 236, 242 |
| 253. | (**N?1** (A-) **N?1** (A- *LN*)) | *No* | 200, 237, 243, 252, 253 |
| 254. | (**N?1** (A- *LN*) **N?1** (A- *LN*)) | *No* | 240, 253, 254 |
| 255. | (**N?1** () **X** (A-)) | *No* | 210, 242, 559 |
| 256. | (**N?1** () **X** (A- *LN*)) | *No* | 211, 243, 255, 256 |
| 257. | (**N?1** (A-) **X?** ()) | *No* | 214, 248, 255 |
| 258. | (**N?1** (A-) **X?** (*LN*)) | *No* | 215, 249, 256, 257, 258 |
| 259. | (**N?1** (A- *LN*) **X?** ()) | *No* | 216, 244, 246, 250 |
| 260. | (**N?1** (A- *LN*) **X?** (*LN*)) | *No* | 217, 245, 247, 251, 259, 260 |
| 261. | (**S** () **M** ()) | *Yes* | 262, 263, 285, 293 |
| 262. | (**S** () **M** (*LN*)) | *Yes* | 261, 262, 264, 286, 294 |
| 263. | (**S** (*LN*) **M** ()) | *Yes* | 261, 263, 264, 289, 297 |
| 264. | (**S** (*LN*) **M** (*LN*)) | *Yes* | 262, 263, 264, 290, 298 |
| 265. | (**S** (L+) **M?** ()) | *Yes* | 261, 266, 267 |
| 266. | (**S** (L+) **M?** (*LN*)) | *Yes* | 262, 265, 266, 268 |
| 267. | (**S** (*LN* L+) **M?** ()) | *Yes* | 263, 267, 268 |
| 268. | (**S** (*LN* L+) **M?** (*LN*)) | *Yes* | 264, 267, 268 |
| 269. | (**S** () **N!** (-)) | *Yes* | 244, 271, 273 |
| 270. | (**S** () **N!** (- *LN*)) | *Yes* | 269, 270, 272, 274 |
| 271. | (**S** () **N!** (*LN* -)) | *Yes* | 245, 271, 275 |
| 272. | (**S** () **N!** (*LN* - *LN*)) | *Yes* | 271, 272, 276 |
| 273. | (**S** (*LN*) **N!** (-)) | *Yes* | 246, 269, 273, 275 |
| 274. | (**S** (*LN*) **N!** (- *LN*)) | *Yes* | 270, 273, 274, 276 |
| 275. | (**S** (*LN*) **N!** (*LN* -)) | *Yes* | 247, 271, 275 |
| 276. | (**S** (*LN*) **N!** (*LN* - *LN*)) | *Yes* | 272, 275, 276 |
| 277. | (**S** () **N?** (-)) | *No* | 248, 279, 281, 293 |
| 278. | (**S** () **N?** (- *LN*)) | *No* | 277, 278, 280, 282, 294 |
| 279. | (**S** () **N?** (*LN* -)) | *No* | 249, 279, 283, 295 |
| 280. | (**S** () **N?** (*LN* - *LN*)) | *No* | 279, 280, 284, 296 |
| 281. | (**S** (*LN*) **N?** (-)) | *No* | 250, 277, 281, 283, 297 |
| 282. | (**S** (*LN*) **N?** (- *LN*)) | *No* | 278, 281, 282, 284, 298 |
| 283. | (**S** (*LN*) **N?** (*LN* -)) | *No* | 251, 279, 283, 299 |
| 284. | (**S** (*LN*) **N?** (*LN* - *LN*)) | *No* | 280, 283, 284, 300 |
| 285. | (**S** () **S** (+)) | *Yes* | 261, 287, 289 |
| 286. | (**S** () **S** (+ *LN*)) | *Yes* | 285, 286, 288, 290 |
| 287. | (**S** () **S** (*LN* +)) | *Yes* | 263, 287, 291 |
| 288. | (**S** () **S** (*LN* + *LN*)) | *Yes* | 287, 288, 292 |
| 289. | (**S** (*LN*) **S** (+)) | *Yes* | 262, 285, 289, 291 |
| 290. | (**S** (*LN*) **S** (+ *LN*)) | *Yes* | 286, 289, 290, 292 |
| 291. | (**S** (*LN*) **S** (*LN* +)) | *Yes* | 264, 287, 291 |
| 292. | (**S** (*LN*) **S** (*LN* + *LN*)) | *Yes* | 288, 291, 292 |
| 293. | (**S** () **X?** (-)) | *No* | 257, 277, 297 |
| 294. | (**S** () **X?** (- *LN*)) | *No* | 278, 293, 294, 298 |
| 295. | (**S** () **X?** (*LN* -)) | *No* | 258, 279, 299 |
| 296. | (**S** () **X?** (*LN* - *LN*)) | *No* | 280, 295, 296, 300 |
| 297. | (**S** (*LN*) **X?** (-)) | *No* | 259, 269, 273, 281, 297 |
| 298. | (**S** (*LN*) **X?** (- *LN*)) | *No* | 270, 274, 282, 297, 298 |
| 299. | (**S** (*LN*) **X?** (*LN* -)) | *No* | 260, 271, 275, 283, 299 |
| 300. | (**S** (*LN*) **X?** (*LN* - *LN*)) | *No* | 272, 276, 284, 299, 300 |

*Table B.1: Membership Protocol Link Conditions*

| Number | Link Condition | Is Link In Tree? | Successors |
|---|---|---|---|
| 301. | (S? (A+) L ()) | Yes | 302, 305, 310, 333 |
| 302. | (S? (A+) L (LN)) | Yes | 301, 302, 306, 311, 336 |
| 303. | (S? (A+ LN) L ()) | Yes | 301, 303, 304, 307, 316 |
| 304. | (S? (A+ LN) L (LN)) | Yes | 302, 303, 304, 308, 317 |
| 305. | (S? (LN A+) L ()) | Yes | 305, 306, 322, 345 |
| 306. | (S? (LN A+) L (LN)) | Yes | 305, 306, 323, 348 |
| 307. | (S? (LN A+ LN) L ()) | Yes | 305, 307, 308, 328 |
| 308. | (S? (LN A+ LN) L (LN)) | Yes | 306, 307, 308, 329 |
| 309. | (S? (A+) L? ()) | Yes | 312, 321, 333 |
| 310. | (S? (A+) L? (-)) | Yes | 309, 313, 322, 334 |
| 311. | (S? (A+) L? (- LN)) | Yes | 310, 311, 314, 323, 335 |
| 312. | (S? (A+) L? (LN)) | Yes | 309, 312, 324, 336 |
| 313. | (S? (A+) L? (LN -)) | Yes | 312, 313, 325, 337 |
| 314. | (S? (A+) L? (LN - LN)) | Yes | 313, 314, 326, 338 |
| 315. | (S? (A+ LN) L? ()) | Yes | 309, 315, 318, 327 |
| 316. | (S? (A+ LN) L? (-)) | Yes | 310, 315, 316, 319, 328 |
| 317. | (S? (A+ LN) L? (- LN)) | Yes | 311, 316, 317, 320, 329 |
| 318. | (S? (A+ LN) L? (LN)) | Yes | 312, 315, 318, 330 |
| 319. | (S? (A+ LN) L? (LN -)) | Yes | 313, 318, 319, 331 |
| 320. | (S? (A+ LN) L? (LN - LN)) | Yes | 314, 319, 320, 332 |
| 321. | (S? (LN A+) L? ()) | Yes | 321, 324, 345 |
| 322. | (S? (LN A+) L? (-)) | Yes | 321, 322, 325, 346 |
| 323. | (S? (LN A+) L? (- LN)) | Yes | 322, 323, 326, 347 |
| 324. | (S? (LN A+) L? (LN)) | Yes | 321, 324, 348 |
| 325. | (S? (LN A+) L? (LN -)) | Yes | 324, 325, 349 |
| 326. | (S? (LN A+) L? (LN - LN)) | Yes | 325, 326, 350 |
| 327. | (S? (LN A+ LN) L? ()) | Yes | 321, 327, 330 |
| 328. | (S? (LN A+ LN) L? (-)) | Yes | 322, 327, 328, 331 |
| 329. | (S? (LN A+ LN) L? (- LN)) | Yes | 323, 328, 329, 332 |
| 330. | (S? (LN A+ LN) L? (LN)) | Yes | 324, 327, 330 |
| 331. | (S? (LN A+ LN) L? (LN -)) | Yes | 325, 330, 331 |
| 332. | (S? (LN A+ LN) L? (LN - LN)) | Yes | 326, 331, 332 |
| 333. | (S? () M (A+)) | Yes | 261, 339, 345, 461, 509 |
| 334. | (S? () M (A+ -)) | Yes | 333, 340, 346, 462, 510 |
| 335. | (S? () M (A+ - LN)) | Yes | 334, 335, 341, 347, 463, 511 |
| 336. | (S? () M (A+ LN)) | Yes | 333, 336, 342, 348, 464, 512 |
| 337. | (S? () M (A+ LN -)) | Yes | 336, 343, 349, 465, 513 |
| 338. | (S? () M (A+ LN - LN)) | Yes | 337, 338, 344, 350, 466, 514 |
| 339. | (S? () M (LN A+)) | Yes | 262, 339, 351, 467, 515 |
| 340. | (S? () M (LN A+ -)) | Yes | 339, 340, 352, 468, 516 |
| 341. | (S? () M (LN A+ - LN)) | Yes | 340, 341, 353, 469, 517 |
| 342. | (S? () M (LN A+ LN)) | Yes | 339, 342, 354, 470, 518 |
| 343. | (S? () M (LN A+ LN -)) | Yes | 342, 343, 355, 471, 519 |
| 344. | (S? () M (LN A+ LN - LN)) | Yes | 343, 344, 356, 472, 520 |
| 345. | (S? (LN) M (A+)) | Yes | 263, 333, 345, 351, 485, 533 |
| 346. | (S? (LN) M (A+ -)) | Yes | 334, 345, 346, 352, 486, 534 |
| 347. | (S? (LN) M (A+ - LN)) | Yes | 335, 346, 347, 353, 487, 535 |
| 348. | (S? (LN) M (A+ LN)) | Yes | 336, 345, 348, 354, 488, 536 |
| 349. | (S? (LN) M (A+ LN -)) | Yes | 337, 348, 349, 355, 489, 537 |
| 350. | (S? (LN) M (A+ LN - LN)) | Yes | 338, 349, 350, 356, 490, 538 |

*Table B.1: Membership Protocol Link Conditions*

| *Number* | *Link Condition* | *Is Link In Tree?* | *Successors* |
|---|---|---|---|
| 351. | (**S?** (*LN*) **M** (*LN* A+)) | *Yes* | 264, 339, 351, 491, 539 |
| 352. | (**S?** (*LN*) **M** (*LN* A+ -)) | *Yes* | 340, 351, 352, 492, 540 |
| 353. | (**S?** (*LN*) **M** (*LN* A+ - *LN*)) | *Yes* | 341, 352, 353, 493, 541 |
| 354. | (**S?** (*LN*) **M** (*LN* A+ *LN*)) | *Yes* | 342, 351, 354, 494, 542 |
| 355. | (**S?** (*LN*) **M** (*LN* A+ *LN* -)) | *Yes* | 343, 354, 355, 495, 543 |
| 356. | (**S?** (*LN*) **M** (*LN* A+ *LN* - *LN*)) | *Yes* | 344, 355, 356, 496, 544 |
| 357. | (**S?** (A+ +) **M?** ()) | *Yes* | 301, 358, 361 |
| 358. | (**S?** (A+ +) **M?** (*LN*)) | *Yes* | 302, 357, 358, 362 |
| 359. | (**S?** (A+ *LN* +) **M?** ()) | *Yes* | 303, 360, 363 |
| 360. | (**S?** (A+ *LN* +) **M?** (*LN*)) | *Yes* | 304, 359, 360, 364 |
| 361. | (**S?** (*LN* A+ +) **M?** ()) | *Yes* | 305, 361, 362 |
| 362. | (**S?** (*LN* A+ +) **M?** (*LN*)) | *Yes* | 306, 361, 362 |
| 363. | (**S?** (*LN* A+ *LN* +) **M?** ()) | *Yes* | 307, 363, 364 |
| 364. | (**S?** (*LN* A+ *LN* +) **M?** (*LN*)) | *Yes* | 308, 363, 364 |
| 365. | (**S?** () **N!** (- A+)) | *Yes* | 269, 377, 389 |
| 366. | (**S?** () **N!** (- A+ -)) | *Yes* | 365, 378, 390 |
| 367. | (**S?** () **N!** (- A+ - *LN*)) | *Yes* | 366, 367, 379, 391 |
| 368. | (**S?** () **N!** (- A+ *LN*)) | *Yes* | 365, 368, 380, 392 |
| 369. | (**S?** () **N!** (- A+ *LN* -)) | *Yes* | 368, 381, 393 |
| 370. | (**S?** () **N!** (- A+ *LN* - *LN*)) | *Yes* | 369, 370, 382, 394 |
| 371. | (**S?** () **N!** (- *LN* A+)) | *Yes* | 270, 383, 395 |
| 372. | (**S?** () **N!** (- *LN* A+ -)) | *Yes* | 371, 384, 396 |
| 373. | (**S?** () **N!** (- *LN* A+ - *LN*)) | *Yes* | 372, 373, 385, 397 |
| 374. | (**S?** () **N!** (- *LN* A+ *LN*)) | *Yes* | 371, 374, 386, 398 |
| 375. | (**S?** () **N!** (- *LN* A+ *LN* -)) | *Yes* | 374, 387, 399 |
| 376. | (**S?** () **N!** (- *LN* A+ *LN* - *LN*)) | *Yes* | 375, 376, 388, 400 |
| 377. | (**S?** () **N!** (*LN* - A+)) | *Yes* | 271, 377, 401 |
| 378. | (**S?** () **N!** (*LN* - A+ -)) | *Yes* | 377, 378, 402 |
| 379. | (**S?** () **N!** (*LN* - A+ - *LN*)) | *Yes* | 378, 379, 403 |
| 380. | (**S?** () **N!** (*LN* - A+ *LN*)) | *Yes* | 377, 380, 404 |
| 381. | (**S?** () **N!** (*LN* - A+ *LN* -)) | *Yes* | 380, 381, 405 |
| 382. | (**S?** () **N!** (*LN* - A+ *LN* - *LN*)) | *Yes* | 381, 382, 406 |
| 383. | (**S?** () **N!** (*LN* - *LN* A+)) | *Yes* | 272, 383, 407 |
| 384. | (**S?** () **N!** (*LN* - *LN* A+ -)) | *Yes* | 383, 384, 408 |
| 385. | (**S?** () **N!** (*LN* - *LN* A+ - *LN*)) | *Yes* | 384, 385, 409 |
| 386. | (**S?** () **N!** (*LN* - *LN* A+ *LN*)) | *Yes* | 383, 386, 410 |
| 387. | (**S?** () **N!** (*LN* - *LN* A+ *LN* -)) | *Yes* | 386, 387, 411 |
| 388. | (**S?** () **N!** (*LN* - *LN* A+ *LN* - *LN*)) | *Yes* | 387, 388, 412 |
| 389. | (**S?** (*LN*) **N!** (- A+)) | *Yes* | 273, 365, 389, 401 |
| 390. | (**S?** (*LN*) **N!** (- A+ -)) | *Yes* | 366, 389, 390, 402 |
| 391. | (**S?** (*LN*) **N!** (- A+ - *LN*)) | *Yes* | 367, 390, 391, 403 |
| 392. | (**S?** (*LN*) **N!** (- A+ *LN*)) | *Yes* | 368, 389, 392, 404 |
| 393. | (**S?** (*LN*) **N!** (- A+ *LN* -)) | *Yes* | 369, 392, 393, 405 |
| 394. | (**S?** (*LN*) **N!** (- A+ *LN* - *LN*)) | *Yes* | 370, 393, 394, 406 |
| 395. | (**S?** (*LN*) **N!** (- *LN* A+)) | *Yes* | 274, 371, 395, 407 |
| 396. | (**S?** (*LN*) **N!** (- *LN* A+ -)) | *Yes* | 372, 395, 396, 408 |
| 397. | (**S?** (*LN*) **N!** (- *LN* A+ - *LN*)) | *Yes* | 373, 396, 397, 409 |
| 398. | (**S?** (*LN*) **N!** (- *LN* A+ *LN*)) | *Yes* | 374, 395, 398, 410 |
| 399. | (**S?** (*LN*) **N!** (- *LN* A+ *LN* -)) | *Yes* | 375, 398, 399, 411 |
| 400. | (**S?** (*LN*) **N!** (- *LN* A+ *LN* - *LN*)) | *Yes* | 376, 399, 400, 412 |

*Table B.1: Membership Protocol Link Conditions*

| Number | Link Condition | Is Link<br>In Tree? | Successors |
|---|---|---|---|
| 401. | (S? (*LN*) N! (*LN* - A+)) | Yes | 275, 377, 401 |
| 402. | (S? (*LN*) N! (*LN* - A+ -)) | Yes | 378, 401, 402 |
| 403. | (S? (*LN*) N! (*LN* - A+ - *LN*)) | Yes | 379, 402, 403 |
| 404. | (S? (*LN*) N! (*LN* - A+ *LN*)) | Yes | 380, 401, 404 |
| 405. | (S? (*LN*) N! (*LN* - A+ *LN* -)) | Yes | 381, 404, 405 |
| 406. | (S? (*LN*) N! (*LN* - A+ *LN* - *LN*)) | Yes | 382, 405, 406 |
| 407. | (S? (*LN*) N! (*LN* - *LN* A+)) | Yes | 276, 383, 407 |
| 408. | (S? (*LN*) N! (*LN* - *LN* A+ -)) | Yes | 384, 407, 408 |
| 409. | (S? (*LN*) N! (*LN* - *LN* A+ - *LN*)) | Yes | 385, 408, 409 |
| 410. | (S? (*LN*) N! (*LN* - *LN* A+ *LN*)) | Yes | 386, 407, 410 |
| 411. | (S? (*LN*) N! (*LN* - *LN* A+ *LN* -)) | Yes | 387, 410, 411 |
| 412. | (S? (*LN*) N! (*LN* - *LN* A+ *LN* - *LN*)) | Yes | 388, 411, 412 |
| 413. | (S? () N? (- A+)) | No | 277, 425, 437, 509 |
| 414. | (S? () N? (- A+ -)) | No | 413, 426, 438, 510 |
| 415. | (S? () N? (- A+ - *LN*)) | No | 414, 415, 427, 439, 511 |
| 416. | (S? () N? (- A+ *LN*)) | No | 413, 416, 428, 440, 512 |
| 417. | (S? () N? (- A+ *LN* -)) | No | 416, 429, 441, 513 |
| 418. | (S? () N? (- A+ *LN* - *LN*)) | No | 417, 418, 430, 442, 514 |
| 419. | (S? () N? (- *LN* A+)) | No | 278, 431, 443, 515 |
| 420. | (S? () N? (- *LN* A+ -)) | No | 419, 432, 444, 516 |
| 421. | (S? () N? (- *LN* A+ - *LN*)) | No | 420, 421, 433, 445, 517 |
| 422. | (S? () N? (- *LN* A+ *LN*)) | No | 419, 422, 434, 446, 518 |
| 423. | (S? () N? (- *LN* A+ *LN* -)) | No | 422, 435, 447, 519 |
| 424. | (S? () N? (- *LN* A+ *LN* - *LN*)) | No | 423, 424, 436, 448, 520 |
| 425. | (S? () N? (*LN* - A+)) | No | 279, 425, 449, 521 |
| 426. | (S? () N? (*LN* - A+ -)) | No | 425, 426, 450, 522 |
| 427. | (S? () N? (*LN* - A+ - *LN*)) | No | 426, 427, 451, 523 |
| 428. | (S? () N? (*LN* - A+ *LN*)) | No | 425, 428, 452, 524 |
| 429. | (S? () N? (*LN* - A+ *LN* -)) | No | 428, 429, 453, 525 |
| 430. | (S? () N? (*LN* - A+ *LN* - *LN*)) | No | 429, 430, 454, 526 |
| 431. | (S? () N? (*LN* - *LN* A+)) | No | 280, 431, 455, 527 |
| 432. | (S? () N? (*LN* - *LN* A+ -)) | No | 431, 432, 456, 528 |
| 433. | (S? () N? (*LN* - *LN* A+ - *LN*)) | No | 432, 433, 457, 529 |
| 434. | (S? () N? (*LN* - *LN* A+ *LN*)) | No | 431, 434, 458, 530 |
| 435. | (S? () N? (*LN* - *LN* A+ *LN* -)) | No | 434, 435, 459, 531 |
| 436. | (S? () N? (*LN* - *LN* A+ *LN* - *LN*)) | No | 435, 436, 460, 532 |
| 437. | (S? (*LN*) N? (- A+)) | No | 281, 413, 437, 449, 533 |
| 438. | (S? (*LN*) N? (- A+ -)) | No | 414, 437, 438, 450, 534 |
| 439. | (S? (*LN*) N? (- A+ - *LN*)) | No | 415, 438, 439, 451, 535 |
| 440. | (S? (*LN*) N? (- A+ *LN*)) | No | 416, 437, 440, 452, 536 |
| 441. | (S? (*LN*) N? (- A+ *LN* -)) | No | 417, 440, 441, 453, 537 |
| 442. | (S? (*LN*) N? (- A+ *LN* - *LN*)) | No | 418, 441, 442, 454, 538 |
| 443. | (S? (*LN*) N? (- *LN* A+)) | No | 282, 419, 443, 455, 539 |
| 444. | (S? (*LN*) N? (- *LN* A+ -)) | No | 420, 443, 444, 456, 540 |
| 445. | (S? (*LN*) N? (- *LN* A+ - *LN*)) | No | 421, 444, 445, 457, 541 |
| 446. | (S? (*LN*) N? (- *LN* A+ *LN*)) | No | 422, 443, 446, 458, 542 |
| 447. | (S? (*LN*) N? (- *LN* A+ *LN* -)) | No | 423, 446, 447, 459, 543 |
| 448. | (S? (*LN*) N? (- *LN* A+ *LN* - *LN*)) | No | 424, 447, 448, 460, 544 |
| 449. | (S? (*LN*) N? (*LN* - A+)) | No | 283, 425, 449, 545 |
| 450. | (S? (*LN*) N? (*LN* - A+ -)) | No | 426, 449, 450, 546 |

*Table B.1: Membership Protocol Link Conditions*

| *Number* | *Link Condition* | *Is Link In Tree?* | *Successors* |
|---|---|---|---|
| 451. | (S? (LN) N? (LN - A+ - LN)) | No | 427, 450, 451, 547 |
| 452. | (S? (LN) N? (LN - A+ LN)) | No | 428, 449, 452, 548 |
| 453. | (S? (LN) N? (LN - A+ LN -)) | No | 429, 452, 453, 549 |
| 454. | (S? (LN) N? (LN - A+ LN - LN)) | No | 430, 453, 454, 550 |
| 455. | (S? (LN) N? (LN - LN A+)) | No | 284, 431, 455, 551 |
| 456. | (S? (LN) N? (LN - LN A+ -)) | No | 432, 455, 456, 552 |
| 457. | (S? (LN) N? (LN - LN A+ - LN)) | No | 433, 456, 457, 553 |
| 458. | (S? (LN) N? (LN - LN A+ LN)) | No | 434, 455, 458, 554 |
| 459. | (S? (LN) N? (LN - LN A+ LN -)) | No | 435, 458, 459, 555 |
| 460. | (S? (LN) N? (LN - LN A+ LN - LN)) | No | 436, 459, 460, 556 |
| 461. | (S? () S (+ A+)) | Yes | 285, 473, 485 |
| 462. | (S? () S (+ A+ -)) | Yes | 461, 474, 486 |
| 463. | (S? () S (+ A+ - LN)) | Yes | 462, 463, 475, 487 |
| 464. | (S? () S (+ A+ LN)) | Yes | 461, 464, 476, 488 |
| 465. | (S? () S (+ A+ LN -)) | Yes | 464, 477, 489 |
| 466. | (S? () S (+ A+ LN - LN)) | Yes | 465, 466, 478, 490 |
| 467. | (S? () S (+ LN A+)) | Yes | 286, 479, 491 |
| 468. | (S? () S (+ LN A+ -)) | Yes | 467, 480, 492 |
| 469. | (S? () S (+ LN A+ - LN)) | Yes | 468, 469, 481, 493 |
| 470. | (S? () S (+ LN A+ LN)) | Yes | 467, 470, 482, 494 |
| 471. | (S? () S (+ LN A+ LN -)) | Yes | 470, 483, 495 |
| 472. | (S? () S (+ LN A+ LN - LN)) | Yes | 471, 472, 484, 496 |
| 473. | (S? () S (LN + A+)) | Yes | 287, 473, 497 |
| 474. | (S? () S (LN + A+ -)) | Yes | 473, 474, 498 |
| 475. | (S? () S (LN + A+ - LN)) | Yes | 474, 475, 499 |
| 476. | (S? () S (LN + A+ LN)) | Yes | 473, 476, 500 |
| 477. | (S? () S (LN + A+ LN -)) | Yes | 476, 477, 501 |
| 478. | (S? () S (LN + A+ LN - LN)) | Yes | 477, 478, 502 |
| 479. | (S? () S (LN + LN A+)) | Yes | 288, 479, 503 |
| 480. | (S? () S (LN + LN A+ -)) | Yes | 479, 480, 504 |
| 481. | (S? () S (LN + LN A+ - LN)) | Yes | 480, 481, 505 |
| 482. | (S? () S (LN + LN A+ LN)) | Yes | 479, 482, 506 |
| 483. | (S? () S (LN + LN A+ LN -)) | Yes | 482, 483, 507 |
| 484. | (S? () S (LN + LN A+ LN - LN)) | Yes | 483, 484, 508 |
| 485. | (S? (LN) S (+ A+)) | Yes | 289, 461, 485, 497 |
| 486. | (S? (LN) S (+ A+ -)) | Yes | 462, 485, 486, 498 |
| 487. | (S? (LN) S (+ A+ - LN)) | Yes | 463, 486, 487, 499 |
| 488. | (S? (LN) S (+ A+ LN)) | Yes | 464, 485, 488, 500 |
| 489. | (S? (LN) S (+ A+ LN -)) | Yes | 465, 488, 489, 501 |
| 490. | (S? (LN) S (+ A+ LN - LN)) | Yes | 466, 489, 490, 502 |
| 491. | (S? (LN) S (+ LN A+)) | Yes | 290, 467, 491, 503 |
| 492. | (S? (LN) S (+ LN A+ -)) | Yes | 468, 491, 492, 504 |
| 493. | (S? (LN) S (+ LN A+ - LN)) | Yes | 469, 492, 493, 505 |
| 494. | (S? (LN) S (+ LN A+ LN)) | Yes | 470, 491, 494, 506 |
| 495. | (S? (LN) S (+ LN A+ LN -)) | Yes | 471, 494, 495, 507 |
| 496. | (S? (LN) S (+ LN A+ LN - LN)) | Yes | 472, 495, 496, 508 |
| 497. | (S? (LN) S (LN + A+)) | Yes | 291, 473, 497 |
| 498. | (S? (LN) S (LN + A+ -)) | Yes | 474, 497, 498 |
| 499. | (S? (LN) S (LN + A+ - LN)) | Yes | 475, 498, 499 |
| 500. | (S? (LN) S (LN + A+ LN)) | Yes | 476, 497, 500 |

*Table B.1: Membership Protocol Link Conditions*

| | | *Is Link* | |
|---|---|---|---|
| *Number* | *Link Condition* | *In Tree?* | *Successors* |
| 501. | (**S?** (*LN*) **S** (*LN* + A+ *LN* -)) | *Yes* | 477, 500, 501 |
| 502. | (**S?** (*LN*) **S** (*LN* + A+ *LN* - *LN*)) | *Yes* | 478, 501, 502 |
| 503. | (**S?** (*LN*) **S** (*LN* + *LN* A+)) | *Yes* | 292, 479, 503 |
| 504. | (**S?** (*LN*) **S** (*LN* + *LN* A+ -)) | *Yes* | 480, 503, 504 |
| 505. | (**S?** (*LN*) **S** (*LN* + *LN* A+ - *LN*)) | *Yes* | 481, 504, 505 |
| 506. | (**S?** (*LN*) **S** (*LN* + *LN* A+ *LN*)) | *Yes* | 482, 503, 506 |
| 507. | (**S?** (*LN*) **S** (*LN* + *LN* A+ *LN* -)) | *Yes* | 483, 506, 507 |
| 508. | (**S?** (*LN*) **S** (*LN* + *LN* A+ *LN* - *LN*)) | *Yes* | 484, 507, 508 |
| 509. | (**S?** () **X?** (- A+)) | *No* | 293, 413, 533 |
| 510. | (**S?** () **X?** (- A+ -)) | *No* | 414, 509, 534 |
| 511. | (**S?** () **X?** (- A+ - *LN*)) | *No* | 415, 510, 511, 535 |
| 512. | (**S?** () **X?** (- A+ *LN*)) | *No* | 416, 509, 512, 536 |
| 513. | (**S?** () **X?** (- A+ *LN* -)) | *No* | 417, 512, 537 |
| 514. | (**S?** () **X?** (- A+ *LN* - *LN*)) | *No* | 418, 513, 514, 538 |
| 515. | (**S?** () **X?** (- *LN* A+)) | *No* | 294, 419, 539 |
| 516. | (**S?** () **X?** (- *LN* A+ -)) | *No* | 420, 515, 540 |
| 517. | (**S?** () **X?** (- *LN* A+ - *LN*)) | *No* | 421, 516, 517, 541 |
| 518. | (**S?** () **X?** (- *LN* A+ *LN*)) | *No* | 422, 515, 518, 542 |
| 519. | (**S?** () **X?** (- *LN* A+ *LN* -)) | *No* | 423, 518, 543 |
| 520. | (**S?** () **X?** (- *LN* A+ *LN* - *LN*)) | *No* | 424, 519, 520, 544 |
| 521. | (**S?** () **X?** (*LN* - A+)) | *No* | 295, 425, 545 |
| 522. | (**S?** () **X?** (*LN* - A+ -)) | *No* | 426, 521, 546 |
| 523. | (**S?** () **X?** (*LN* - A+ - *LN*)) | *No* | 427, 522, 523, 547 |
| 524. | (**S?** () **X?** (*LN* - A+ *LN*)) | *No* | 428, 521, 524, 548 |
| 525. | (**S?** () **X?** (*LN* - A+ *LN* -)) | *No* | 429, 524, 549 |
| 526. | (**S?** () **X?** (*LN* - A+ *LN* - *LN*)) | *No* | 430, 525, 526, 550 |
| 527. | (**S?** () **X?** (*LN* - *LN* A+)) | *No* | 296, 431, 551 |
| 528. | (**S?** () **X?** (*LN* - *LN* A+ -)) | *No* | 432, 527, 552 |
| 529. | (**S?** () **X?** (*LN* - *LN* A+ - *LN*)) | *No* | 433, 528, 529, 553 |
| 530. | (**S?** () **X?** (*LN* - *LN* A+ *LN*)) | *No* | 434, 527, 530, 554 |
| 531. | (**S?** () **X?** (*LN* - *LN* A+ *LN* -)) | *No* | 435, 530, 555 |
| 532. | (**S?** () **X?** (*LN* - *LN* A+ *LN* - *LN*)) | *No* | 436, 531, 532, 556 |
| 533. | (**S?** (*LN*) **X?** (- A+)) | *No* | 297, 365, 389, 437, 533 |
| 534. | (**S?** (*LN*) **X?** (- A+ -)) | *No* | 366, 390, 438, 533, 534 |
| 535. | (**S?** (*LN*) **X?** (- A+ - *LN*)) | *No* | 367, 391, 439, 534, 535 |
| 536. | (**S?** (*LN*) **X?** (- A+ *LN*)) | *No* | 368, 392, 440, 533, 536 |
| 537. | (**S?** (*LN*) **X?** (- A+ *LN* -)) | *No* | 369, 393, 441, 536, 537 |
| 538. | (**S?** (*LN*) **X?** (- A+ *LN* - *LN*)) | *No* | 370, 394, 442, 537, 538 |
| 539. | (**S?** (*LN*) **X?** (- *LN* A+)) | *No* | 298, 371, 395, 443, 539 |
| 540. | (**S?** (*LN*) **X?** (- *LN* A+ -)) | *No* | 372, 396, 444, 539, 540 |
| 541. | (**S?** (*LN*) **X?** (- *LN* A+ - *LN*)) | *No* | 373, 397, 445, 540, 541 |
| 542. | (**S?** (*LN*) **X?** (- *LN* A+ *LN*)) | *No* | 374, 398, 446, 539, 542 |
| 543. | (**S?** (*LN*) **X?** (- *LN* A+ *LN* -)) | *No* | 375, 399, 447, 542, 543 |
| 544. | (**S?** (*LN*) **X?** (- *LN* A+ *LN* - *LN*)) | *No* | 376, 400, 448, 543, 544 |
| 545. | (**S?** (*LN*) **X?** (*LN* - A+)) | *No* | 299, 377, 401, 449, 545 |
| 546. | (**S?** (*LN*) **X?** (*LN* - A+ -)) | *No* | 378, 402, 450, 545, 546 |
| 547. | (**S?** (*LN*) **X?** (*LN* - A+ - *LN*)) | *No* | 379, 403, 451, 546, 547 |
| 548. | (**S?** (*LN*) **X?** (*LN* - A+ *LN*)) | *No* | 380, 404, 452, 545, 548 |
| 549. | (**S?** (*LN*) **X?** (*LN* - A+ *LN* -)) | *No* | 381, 405, 453, 548, 549 |
| 550. | (**S?** (*LN*) **X?** (*LN* - A+ *LN* - *LN*)) | *No* | 382, 406, 454, 549, 550 |

*Table B.1: Membership Protocol Link Conditions*

| Number | Link Condition | Is Link<br>In Tree? | Successors |
|---|---|---|---|
| 551. | (**S**? (*LN*) **X**? (*LN* - *LN* **A**+)) | *No* | 300, 383, 407, 455, 551 |
| 552. | (**S**? (*LN*) **X**? (*LN* - *LN* **A**+ -)) | *No* | 384, 408, 456, 551, 552 |
| 553. | (**S**? (*LN*) **X**? (*LN* - *LN* **A**+ - *LN*)) | *No* | 385, 409, 457, 552, 553 |
| 554. | (**S**? (*LN*) **X**? (*LN* - *LN* **A**+ *LN*)) | *No* | 386, 410, 458, 551, 554 |
| 555. | (**S**? (*LN*) **X**? (*LN* - *LN* **A**+ *LN* -)) | *No* | 387, 411, 459, 554, 555 |
| 556. | (**S**? (*LN*) **X**? (*LN* - *LN* **A**+ *LN* - *LN*)) | *No* | 388, 412, 460, 555, 556 |
| 557. | (**X** () **M**? (+)) | *No* | 222, 265, 558 |
| 558. | (**X** () **M**? (*LN* +)) | *No* | 223, 266, 558 |
| 559. | (**X** () **N** ()) | *No* | 224, 557, 560 |
| 560. | (**X** () **X** ()) | *No* | 559 |

We can prove directly from Table B.1 that our protocol has the desired closure and consistency properties alluded to at the beginning of this section. For closure, we note that there is no configuration in the table in which a processor is presented with a message it cannot handle in its current state. For consistency, we note that the only quiescent link conditions are our original five:

(**X** () **X** ())  (**X** () **N** ())  (**N** () **N** ())  (**L** () **L** ())  (**S** () **M** ())

and that each of these has the desired status of the link as being in or out of the reference tree.

## B.2: Global Properties of the Membership Protocol

Determining additional properties of the membership protocol requires combining the data presented in this section with a little graph theory. For our purposes, we make the following definitions:

**Definition B.2:**

An *undirected graph* is an ordered pair $(N,A)$, where $N$ is a subset of the set $\mathbf{N}$ of *nodes*, and $A$ is a set of *arcs* $\{m,n\}$ such that $m,n \in N$.

Intuitively, for every arc drawn between nodes $m$ and $n$ in an undirected graph $(N,A)$, $A$ contains a doubleton set $\{m,n\}$.

**Definition B.3:**

A node $n$ is a *leaf node* of the undirected graph $G = (N,A)$ iff $n \in N$ and there is exactly one node $n'$ such that $\{n,n'\} \in A$. A node $n \in N$ is an *isolated node* of $G$ iff $n \notin a$ for any $a \in A$.

The concept of a leaf node is the familiar one of a node with exactly one arc attached; an isolated node is a node with no arcs attached to it.

**Definition B.4:**

A *reference tree transformation* of an undirected graph $G = (N,A)$ is one of

1. the original graph $G$, or
2. a new graph $G' = (N \cup \{n\}, A \cup \{\{m,n\}\})$ where $m$ and $n$ are nodes such that $m \in N$ and $n \notin N$, or
3. a new graph $G' = (N-\{n\}, A-\{\{m,n\}\})$ where $m,n \in N$, $n$ is a leaf node of $G$, and $\{m,n\} \in A$.

Intuitively, a reference tree transformation of a graph $G$ is, if $G$ is expanded, a graph with a new node such that the new node is a leaf node and is connected to some other node already in $G$, and if $G$ is contracted, a new graph minus one of the leaf nodes in $G$ (and also minus the associated arc). The purpose of defining reference tree transformations is to introduce

**Lemma B.5:**

If the set of processors and links that are members of a reference tree is viewed as an undirected graph, the membership protocol performs only reference tree transformations on reference trees.

**proof** follows from the information presented in Table B.1, with some elaborations. Our strategy is to examine the transformation effected by each kind of transition from one link condition to another.

*case* 1: No change occurs in the membership status of processors or links. This is a valid reference tree transformation, by clause 1 of Definition B.4.

*case* 2: One of the processors undergoes the transition **X:R+/L+:S** or **X?:***LN*/**:N!**. In this case both the link and the processor undergoing the transition join the reference tree. In all cases where this happens, the processor at the other end of the link is already in the reference tree, making this an instance of clause 2 of Definition B.4, assuming that no other links join the reference tree at the same time. This assumption is valid because the other link states of the processor joining the reference tree will simultaneously undergo either the transition **X:/:N** or the transition **X?:/:N?**, neither of which causes any of these links to join the reference tree.

*case* 3: One of the processors undergoes the transition **X:/:N** or **X?:/:N?**. In this case the processor joins the reference tree but the link does not. If some other link attached to the processor simultaneously joins the reference tree, as in case 2, above, then the resulting transformation is a valid reference tree transformation. It would be possible, though, to imagine a situation in which a processor spontaneously decided to join an object's reference tree, say, by undergoing the transition **X:/:N** for all attached links. This would not be a reference tree transformation because a processor would join the "tree" without any corresponding link also joining. This scenario is ruled out, however, by the restrictions on the applicability of these transitions discussed at the head of this appendix.

*case* 4: One of the processors undergoes the transition **M:/-:X?**. In this case both the link and the processor undergoing the transition leave the reference tree. This transition is only allowed if the processor's state for every other link is either **N** or **N?**, which (as can be determined from examination of Table B.1) ensures that no other link attached to that processor is in the reference tree. Hence the processor undergoing the transition is a leaf node of the reference tree, making this a reference tree transformation of the kind described in clause 3 of Definition B.4.

*case* 5: One of the processors undergoes the transition **N:/:X** or **N?:/:X?**. This can only happen if on some other link the processor is undergoing a transition of the type discussed in case 4. Therefore this case, too, satisfies clause 3 of Definition B.4.

These five cases exhaust the kinds of changes that occur between link conditions that appear as successors of each other in Table B.1.

□

We now proceed to develop some more terminology.

**Definition B.6:**

A *path* from node $n$ to node $n'$ in an undirected graph $G = (N,A)$ is a set $P = \{a_0,a_1,\ldots,a_k\} \subseteq A$ such that

1. there exists a sequence $m_0,m_1,\ldots,m_{k+1} \in N$ such that for $0 \leq i \leq k$, $a_i = \{m_i,m_{i+1}\}$;
2. if $i \neq j$, then $a_i \neq a_j$; and
3. $m_0 = n$ and $m_{k+1} = n'$.

A path between two nodes is thus a set of arcs, with no arc used more than once, which connects the two nodes.

**Definition B.7:**

An undirected graph $G = (N,A)$ is *connected* iff for any $m,n \in N$, where $m \neq n$, there exists a path from $m$ to $n$ in $G$.

**Definition B.8:**

An undirected graph $G = (N,A)$ is *acyclic* iff for any $m,n \in N$, where $m \neq n$, there is at most one path from $m$ to $n$ in $G$.

Some subsidiary lemmas help support the main result of this section:

**Lemma B.9:**

If an undirected graph $G = (N,A)$ has the property that $|N| = 1+|A|$, then either $G$ has a leaf node or $G$ has an isolated node.

**proof** by contradiction: Assume $G$ has no leaf node or isolated node. Then for every node $n \in N$, there are at least two arcs $a \in A$ such that $n \in a$. Since all arcs are doubleton sets, the total number of arcs must be at least as great as the total number of nodes. Thus $|N| \leq |A|$, contradicting the assumption that $|N| = 1+|A|$.

**Lemma B.10:**

If an undirected graph $G = (N,A)$ is connected and $|N| = 1+|A|$, then the graph $G$ is acyclic.

**proof** by induction on $|A|$:

*basis*: $|A| = 0$. Then there is only one node in $G$ and Definition B.8 is satisfied vacuously.

*induction*: Assume the lemma for $|A| = k$. If $|A| = k+1$ then $G$ has at least two nodes. Since $G$ is connected, there must be a path between any node in $G$ and any other. No node in $G$ can be an isolated node, for there can be no path from an isolated node to any other node. Therefore, by Lemma B.9, $G$ must have at least one leaf node $n$. Let $\{m,n\} \in A$ be the one arc connecting to node $n$. Then the graph $G' = (N-\{n\},A-\{\{m,n\}\})$ is the graph $G$ minus the leaf node $n$ and its connecting arc. By the induction hypothesis, $G'$ is acyclic. Since $n$ is a leaf node of $G$, the only paths in $G$ that include the arc $\{m,n\}$ are those that start or end on $n$. In order for $G$ to be acyclic, there must be at most one path between any pair $j,k$, of distinct nodes in $G$. Two cases can be distinguished.

*case 1*: $j,k \in G'$. Then $j \neq n$ and $k \neq n$, hence no path from $j$ to $k$ in $G$ could include the arc $\{m,n\}$. Thus any path between $j$ and $k$ in $G$ is also a path between $j$ and $k$ in $G'$. By the induction hypothesis, there can be at most one of these.

*case 2*: $j = n$ or $k = n$. Without loss of generality, we can assume that $j = n$. Then any path from $j$ to $k$ in $G$ must begin with the arc $\{m,n\}$. If $k = m$, this arc is the unique path from $j$ to $k$. If $k \neq m$, a path from $j$ to $k$ must be the union of $\{m,n\}$ and a path from $m$ to $k$ in $G$. Since $m,k \in G'$, any path from $m$ to $k$ in $G$ is also in $G'$. By the induction hypothesis, there is at most one path from $m$ to $k$ in $G'$. Hence there is at most one path from $j$ to $k$ in $G$.

□

**Lemma B.11:**

*If a graph $G = (N,A)$ has the property that $|N| = 1+|A|$, then any reference tree transformation $G' = (N',A')$ of $G$ has the property that $|N'| = 1+|A'|$.*

**proof**: Each of the three cases of Definition B.4 adds or subtracts the same number of elements to or from $N$ and $A$, thus keeping constant the difference between their cardinalities.

□

**Lemma B.12:**

If a graph $G = (N,A)$ is connected, then any reference tree transformation $G' = (N',A')$ of $G$ is also connected.

**proof by cases**:

case 1: $G' = G$. Then $G'$ is obviously connected if $G$ is.

case 2: $G' = (N \cup \{n\}, A \cup \{\{m,n\}\})$ where $m$ and $n$ are nodes such that $m \in N$ and $n \notin N$. Then $G'$ is connected if there exists a path in $G'$ between every two distinct nodes $j,k$ in $G'$. If $j,k \in N$ then there exists a path from $j$ to $k$ using only arcs in $G$, all of which are also present in $G'$. Otherwise either $j = n$ or $k = n$. Without loss of generality we may assume $j = n$. Then, if $k = m$, $\{\{m,n\}\}$ is a path from $j$ to $k$ in $G'$. Otherwise, if $P$ is a path from $m$ to $k$ in $G$, then $P \cup \{\{m,n\}\}$ is a path from $j$ to $k$ in $G$. Hence $G'$ is connected if $G$ is.

case 3: $G' = (N-\{n\}, A-\{\{m,n\}\})$ where $m,n \in N$, $n$ is a leaf node of $G$, and $\{m,n\} \in A$. Since $n$ is a leaf node of $G$, no path between nodes $j$ and $k$ in $G$, where $j \neq n$ and $k \neq n$, could include the arc $\{m,n\}$. Thus any such path between nodes $j$ and $k$ in $G$ is also a path between $j$ and $k$ in $G'$. Since *any* pair of nodes in $G'$ falls into this category, and since $G$ is connected, $G'$ must be connected.

□

**Lemma B.13:**

If a graph $G = (N,A)$ is connected and has the property that $|N| = 1+|A|$, the result $G' = (N',A')$ of performing any number of successive reference tree transformations on $G$ is also connected and has the property that $|N'| = 1+|A'|$.

**proof** by induction on the number $k$ of reference tree transformations applied:

*basis*: $k = 0$. Then $G = G'$ and the lemma is trivially true.

*induction*: Assume the lemma for $k = n$ successive reference tree transformations. To show the result for $k+1$ transformations, consider the graph $G'' = (N'',A'')$ resulting at the end of the first $k$ transformations in the sequence. By the hypothesis, $G''$ is connected and has the property that $|N''| = 1+|A''|$. Then one more transformation on $G''$ will produce $G'$. By Lemmas B.11 and B.12, $G'$ is connected and has the property that $|N'| = 1+|A'|$.

□

**Theorem B.14:**

Any number of successive *reference tree transformations* on the initial graph $G = (\{n\},\{\})$ will always yield a connected, acyclic graph.

**proof**: $G = (N,A)$ is connected and has the property that $|N| = 1+|A|$. Thus by Lemma B.13 any graph $G' = (N',A')$ produced from $G$ by any number of successive *reference tree transformations* is connected and has the property that $|N'| = 1+|A'|$. By Lemma B.10, this means that $G'$ is acyclic as well as being connected.

□

Since every object's reference tree starts out including just the single processor where the object was created, Theorem B.14 assures us that no reference tree will ever become disconnected or come to include cycles.

### B.3: Summary

This appendix first recapitulated the workings of the reference tree membership protocol, including the various restrictions on the applicability of some of the state transitions. This protocol was then analyzed for the local properties of closure and consistency, and the global properties of resistance to disconnection and resistance to forming cycles. Conclusions of the analysis were as follows:

- closure — every message that can arrive at a processor in a particular state can be handled by the processor in that state.

- consistency — our assignment of link status (as being in or out of the reference tree) to link conditions is consistent both with our assumptions about when processors enter and leave the tree and with the link status desired to accompany each quiescent link condition (condition in which no messages are in transit).

- resistance to disconnection — reference trees do not become disconnected.

- resistance to forming cycles — cycles do not form in reference trees.

## OFFICIAL DISTRIBUTION LIST

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office/Boston
Building 114, Section D
666 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office/Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office/Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area
715 Broadway - 5th floor
New York, N. Y. 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375
6 copies

Assistant Chief for Technology
Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Naval Ocean Systems Center, Code 91
Headquarters-Computer Sciences &
Simulation Department
San Diego, CA 92152
Mr. Lloyd Z. Maudlin
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Math Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D. C. 20374
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division
(OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Captain Richard L. Martin, USN
Commanding Officer
USS Francis Marion (LPA-249)
FPO New York, N. Y. 09501
1 copy