M.I.T. Laboratory for Computer Science 11 May 1976
Computer Systems Research Division Request for Comments No.112

# THE DUPLICATE DATABASE PROBLEM

by

Clarence A. Ellis*
Laboratory for Computer Science
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

* The research work described herein was initiated while the author was under a visiting summer faculty appointment at the IBM T.J. Watson Research Center.

---

This note is an informal working paper of the M.I.T. Laboratory for Computer Science, Computer Systems Research Division. It should not be reproduced without the author's permission and it should not be referenced in other publications.

TABLE OF CONTENTS

## ABSTRACT

Solutions to the duplicate database problem are considered, and a new formal validation technique using the theory of L systems is developed and applied to the problem. The paper is not so much concerned with any particular solution as it is with general properties of the problem, convenient representational techniques, and formal proof procedures which are general enough to apply to this and to a number of other problems in parallel processing and synchronization.

# 1. INTRODUCTION

This paper considers the problem of maintenance of multiple copies of a database within a computer network. After an explanation of the significance of this problem, and a review of the literature concerning it, the problem is stated precisely and several solutions are presented.

This report also details experiences gained by the author through trial-and-error solutions which seemed for all intents and purposes to be correct but which contained flaws. Finally, attempts were made to generalize and relate this problem to classes of problems which arise in the operating systems area, and to utilize the most expedient and appropriate theories and models which have been developed within other spheres of computer science. Toward this end, the notation of Evaluation Nets (16) is used to describe solutions, because it proved to be a very understandable but efficient medium. Similarly, experience showed that it is extremely desirable to verify, in a rigorous manner if possible, the correctness of proposed solutions. Toward this end, the theory and formal techniques of L systems (19) were employed in a manner which seems to offer many, still unexplored, possibilities of application of existing theorems and derivation of new theorems for correctness proofs. In general, the validation of systems with parallel processing has proved to be a difficult problem. The technique presented herein to prove the correctness or incorrectness of the interaction of asynchronous parallel systems is quite general, and presents a

new application of L systems. The proof procedure using L systems is effective, and thus can be mechanically carried out on a computer. The last part of the paper describes a set of computer programs written in SNOBOL which automatically verifies the correctness of interaction of parallel processes. The paper shows the result of applying these programs to a proposed solution to the duplicate database problem.

## 2. THE PROBLEM

A computer network may be centralized, implying that all databases are kept at one central location, or distributed, implying that data and control reside at various locations throughout the network. If the distributed approach is used, it may be desirable to store multiple identical copies of the same database at many nodes of the network. This is very useful for a system which expects many queries to the database from many nodes. Queries may then be satisfied without requiring information transfer from node to node by simply examining the local copy of the database. The process of updating the database is made more complicated because information must be transferred to all nodes containing copies, and the update must be performed on all databases to keep them identical. Furthermore, update requests may occur in rapid succession or at the same time at different nodes which may cause problems such as nonidentical databases at the various nodes. One thus needs to carefully construct a controller or database manager process at each node to insure that the database update process is performed correctly and efficiently. The construction of this controller (7) is the problem addressed by this paper. The problem is formally stated below.

## Duplicate Database Problem

Problem:

Given an arbitrary computer network, N.

Given a database D which exists in multiple copies at k of the nodes of N.

Show and prove the correctness of a distributed synchronization algorithm which allows and controls the updating of D in such a way that all copies remain consistent (i.e. identical except for transient update times).

Properties of a "good" solution:

(1) Consistency-----After the system quiesces, all copies of D look the same.

(2) Speed Independence-----Solution should be independent of how long (but $< \infty$ ) transmission operations take.

(3) Functionality-----Allows arbitrary function to be transmitted and applied to all copies of D.

(4) Deadlock Free-----One or more nodes cannot get into a permanently blocked state waiting for other nodes, or waiting for a message which will never arrive.

(5) No Critical Blocking-----Each node's update request will eventually be fulfilled. Thus, there does not exist a sequence of timings such that two or more nodes can continually attempt to synchronize and fail.

(6) Homogeneous-----All nodes have essentially identical control programs.

Notice that without loss of generality, it can be assumed that all updates to the database D are initiated by a node holding a copy of D, and updates require locking or synchronizing all of all copies of D. The term database used throughout this paper may refer to a data set or other unit of information. For example, in IMS* the term database might be replaced by the IMS segment. Thus, it is not necessary that application of solution techniques here presented imply that all of a database must be locked. If a solution of simply performing an update function on the local database and then transmitting the function to all other nodes is adopted, then the property (1) of consistency may be violated as indicated by the following example.

Suppose two copies of a database containing a positive number in record 1 and a negative number in record 2 are simultaneously updated by functions $f_1$ at node A and $f_2$ at node B. $f_1$ sets record 1 to the sign (positive or negative) of record 2; $f_2$ sets record 2 to the sign of record 1. After setting the records at node A to negative values and the records at B to positive, both nodes will transmit their functions to the opposite node and perform them. When this is completed, the negative values of records 1 and 2 at node A will be inconsistent with the positive values at node B. This illustrates that the type of problem with which we are concerned falls within the realm of parallel processing and synchronization. Verification

---

* IMS (21) stands for Information Management System, an IBM program product used by many businesses.

of correctness of these systems tends to be more difficult than verification of strictly sequential programs because of added complexity due to timing considerations, and irreproducibility of errors when they occur. The problem exposed within this example would be avoided if both nodes transmitted the value obtained by their function, and the receiving node simply stored a value into a record. Johnson and Thomas have investigated the duplicate database problem making this value-transmission assumption. Notice that if the above example is modified so that $f_1$ and $f_2$ store their result in the same record, say record 3, then the value-transmission scheme leads to inconsistency. In a recent working paper (11), Johnson and Thomas have briefly discussed the general problem, and then outlined in some detail a solution for a particular type of duplicate database in which only the operations of query, assignment, creation, and deletion are allowed. There are obvious particular instances in which either value-transmission or function-transmission is untenable. Thus, property (3), functionality, requires that solutions to the problem be general in the sense that they allow arbitrary functions (or values) to be transmitted and applied to all copies of the database. The problems shown in the above example only appear when several updates occur in very close time proximity. The properties of speed independence, deadlock freedom, and no critical blocking are all concerned with correct operation under various time sequences of events. The final property (6) of homogeneity is one which may be discarded in an actual network

implementation. The notation of essentially identical control programs means that the flowcharts of the global database access control programs at each node look alike (i.e. their algorithms are the same), although the nodes may have entirely different computer systems. This property lends a bit of symmetry and elegance to solutions, allows verification to proceed by viewing the control program of a single node, and leads to possibilities of formal proof of correctness of arbitrarily large networks by induction on the number of nodes in the network.

## 3. THE SOLUTION

There are several solutions to the duplicate database problem present in the literature (11,13,22). However, there seems to be no coverage of the problem which is provably correct and which successfully fulfills all of the generality of the requirements listed in the previous section. This section presents and discusses several solutions using a uniform graphical presentation medium. Then Section 4 explains the automated verification of correctness technique applicable to these solutions.

One solution to the problem consists of the formation of a supervisor at one of the nodes to allocate updating privileges to all nodes. When a given node wishes to update the database, it requests permission from the supervisor, and if no other node is updating, permission is granted. When the given node has finished updating (at all concerned nodes), it must signal completion to the supervisor so that other nodes will then be allowed to update. The evaluation net describing the control program at the given node for this solution is shown in Figure 1.

Evaluation nets (17) are a modified form of Petri nets (10,18). They are good for explanation of solutions to our problem because they are graphical, unambiguous, and application oriented. Transition schema allow token flow via fork, join, and select transitions, among others. In Figure 1, a token on the location (i.e. circle) called IDLE means that the node is not

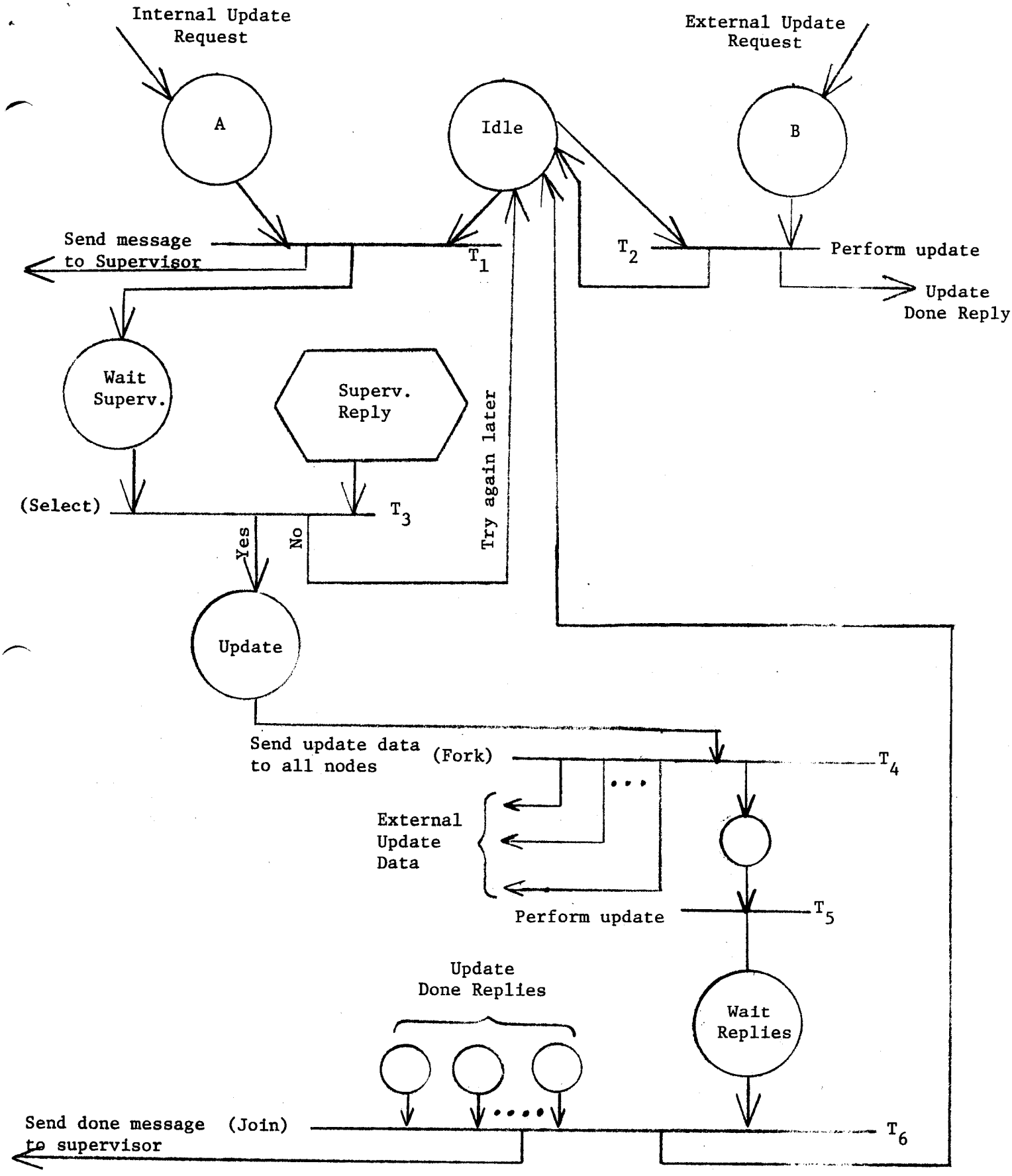
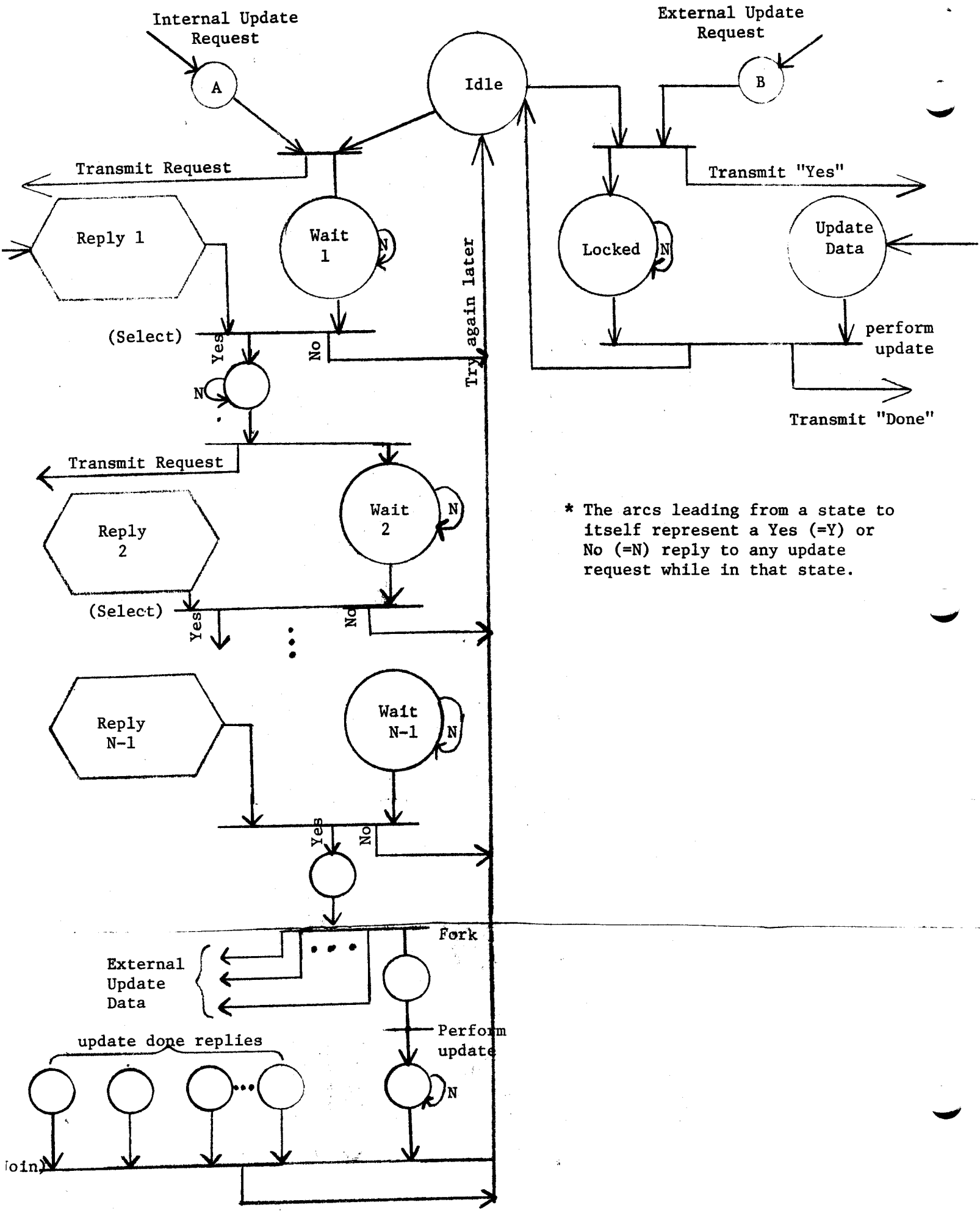



Figure 1 Centralized Solution
(Supervisor can insure fairness)

servicing any update request. An internal update request occurring from a local process is denoted by a token arriving on location A. If there are tokens on both locations A and IDLE, then the transition T1 can 'fire', removing tokens from its input locations and delivering tokens to its output locations. One of the outputs is external to Figure 1 and delivers a request token to the supervisor node. A token on the WAIT SUPERV node indicates that the local controller is waiting for a message from the supervisor node saying YES (=1) you can update, or NO (=0) you cannot update. This message is delivered by placing a 1 or 0 token onto the hexagonal node called a resolution location. Transition T3 fires when tokens are on both of its input locations. These tokens are removed and a token is placed on one or the other of the output locations depending upon the value of the token on the resolution location. In this example, if the supervisor sends a No message, then a token will be placed on IDLE meaning that the local update request was rejected and must be attempted again at a later time. The controller is thus readied for receiving other external requests. A YES message implies that it is allowed to make the local update and other nodes containing copies of the database will not interfere. The transitions following the UPDATE location illustrate a fork transition to send out the updating function or data, and a join transition to receive update done replies from all nodes. After this, the local controller signals completion to the supervisor so that others can access the database, and the controller then

returns to the IDLE state. It can be observed from this example that tokens on locations may represent states of the controller and transitions represent indivisible actions which the controller performs. Occasionally, the semantics of the actions will be written as comments beside the appropriate transition. See Figure 1. More information on evaluation nets, along with a formal definition, can be found in Nutt (16). A careful implementation of the control strategy of Figure 1 (plus a supervisor) yields a correct solution. However, this solution is not distributed, and thus fails criterion 6. The success of the whole system in this solution is critically dependent upon the supervisory node. Mullery (14) shows that this critical dependence can be avoided in distributed systems. We, thus, seek a better solution by distributing control. A straightforward approach which minimizes the amount of parallel activity requires that a controller which wants to update must signal to all involved nodes in a sequential one-by-one fashion and receive a YES reply from each before updating. This distributed solution is shown in Figure 2. This basic solution has a problem of critical blocking, in which the time sequence may be such that several nodes all attempt simultaneously to update, and all fail. Then they all wait and two or more again try at (close to) the same time, thus continually failing so that the database never gets updated. One solution to this is to assign unequal priorities to the nodes and let the highest priority node be the winner of any tie. This raises the possibility that a lower




* The arcs leading from a state to itself represent a Yes (=Y) or No (=N) reply to any update request while in that state.

priority node may be locked out, and never perform its update. The solution shown in Figure 3 uses timestamps to solve this dilemma. A clock (not necessarily identical time) is placed at each node, and is used to be sure that an earlier request is not delayed until after a later request. It is still possible that two or more requests originate at the same time so some priority is still necessary to break ties. Are priorities necessary in all solutions? No, a numbering of nodes can be implemented which is changed each time an update is performed. Are clocks and timestamps necessary? No, the function of ordering carried out by a timestamp can be performed by integer counters which simply count how long in terms of updates a node has waited since its last turn. Experience with concocting and verifying solutions indicates that flaws in solutions are not easy to find. Solutions should be speed independent, so one should consider the consequences of a node sending two messages and the second arriving before the first. This consideration may invalidate what looks like a good solution. Similarly, it is frequently dangerous if a node can send out messages and wait for answers in some cases but not wait for them in other situations. This occurs when a node requests permission to update from all nodes, receives a NO from some node, and then doesn't wait for further replies which may not return until the next update transaction much later. Properties of minimal message transfer, clarity and elegance of solution, maximal parallelism, practicality, and generality often conflict with each other. An obvious direction

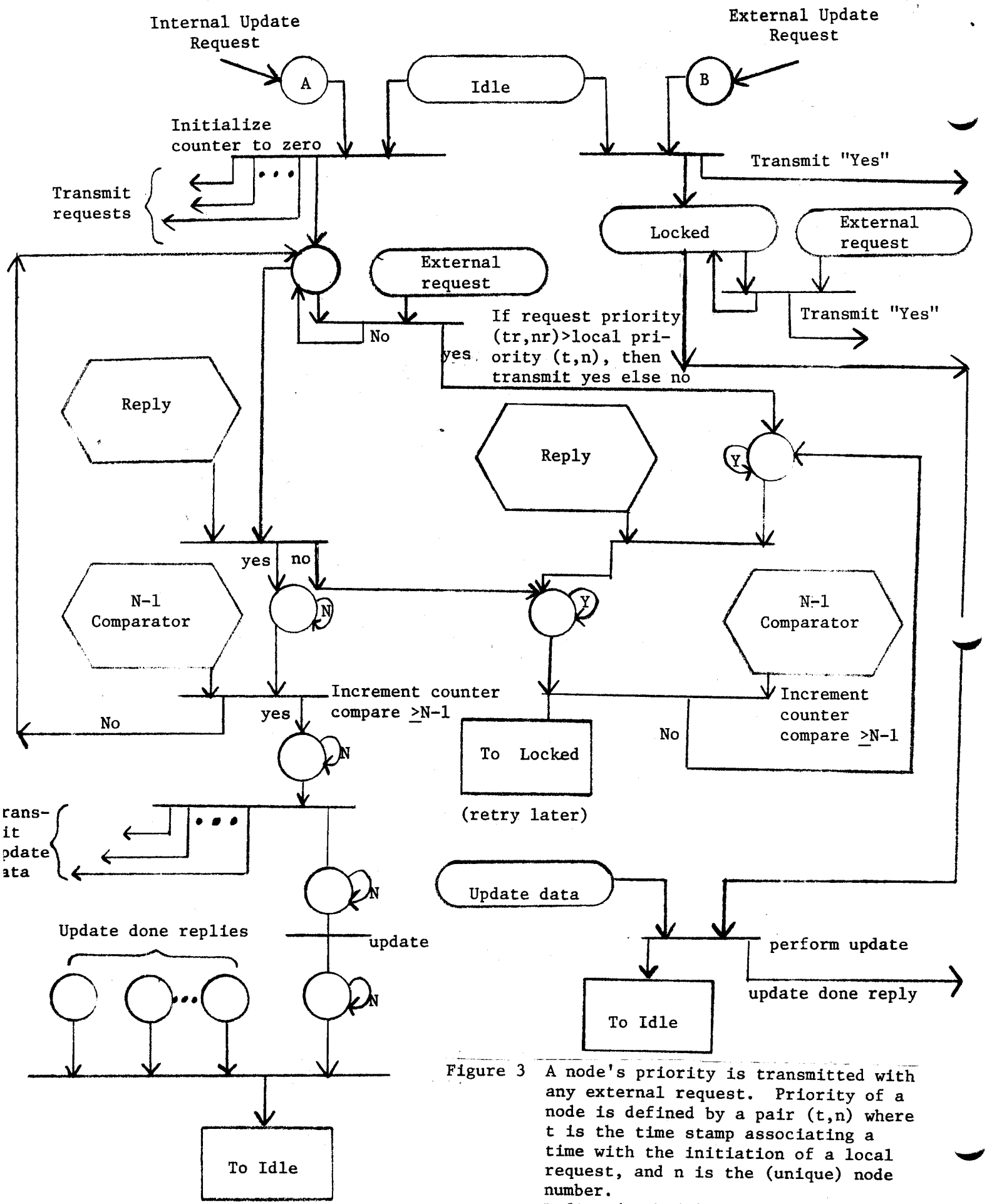



Figure 3 A node's priority is transmitted with any external request. Priority of a node is defined by a pair (t,n) where t is the time stamp associating a time with the initiation of a local request, and n is the (unique) node number.
Define (t,n)>(t',n') if and only if (t>t') or (t=t'+n>n')
(note: retry of old request keeps old time stamp)

to pursue in improving the primitive solution of Figure 2 is to attempt parallelism by sending update requests in parallel. The solution of Figure 4 does this and attempts to avoid critical blocking by forming a queue of all requests which occur near the same time. This solution fails because of the subtle problem that the various nodes may not have consistent copies of the queue. Thus, the original problem has not been solved but recast as the maintenance of multiple copies of the queue. This method offers a decrease in the amount of protocol transmission because one locking allows a whole queue of updates to take place. Finally, Figure 5 shows a solution which is very elegant, requires few transmissions, but is not built upon the premise of maximal parallelism. It has the advantage that a node need not know how many nodes, N, are in the network. It has the disadvantage of serial propagation through the network. A controller only sends messages to the next higher numbered node modulo N, and receives from the next lower node. A request to update propagates around this ring structure and is accepted when it returns to its sender. The (update + completion acknowledgement) is then done in a similar manner. Simultaneous requests are handled by storing the lower number in the higher numbered node, and every node is guaranteed to receive a turn.

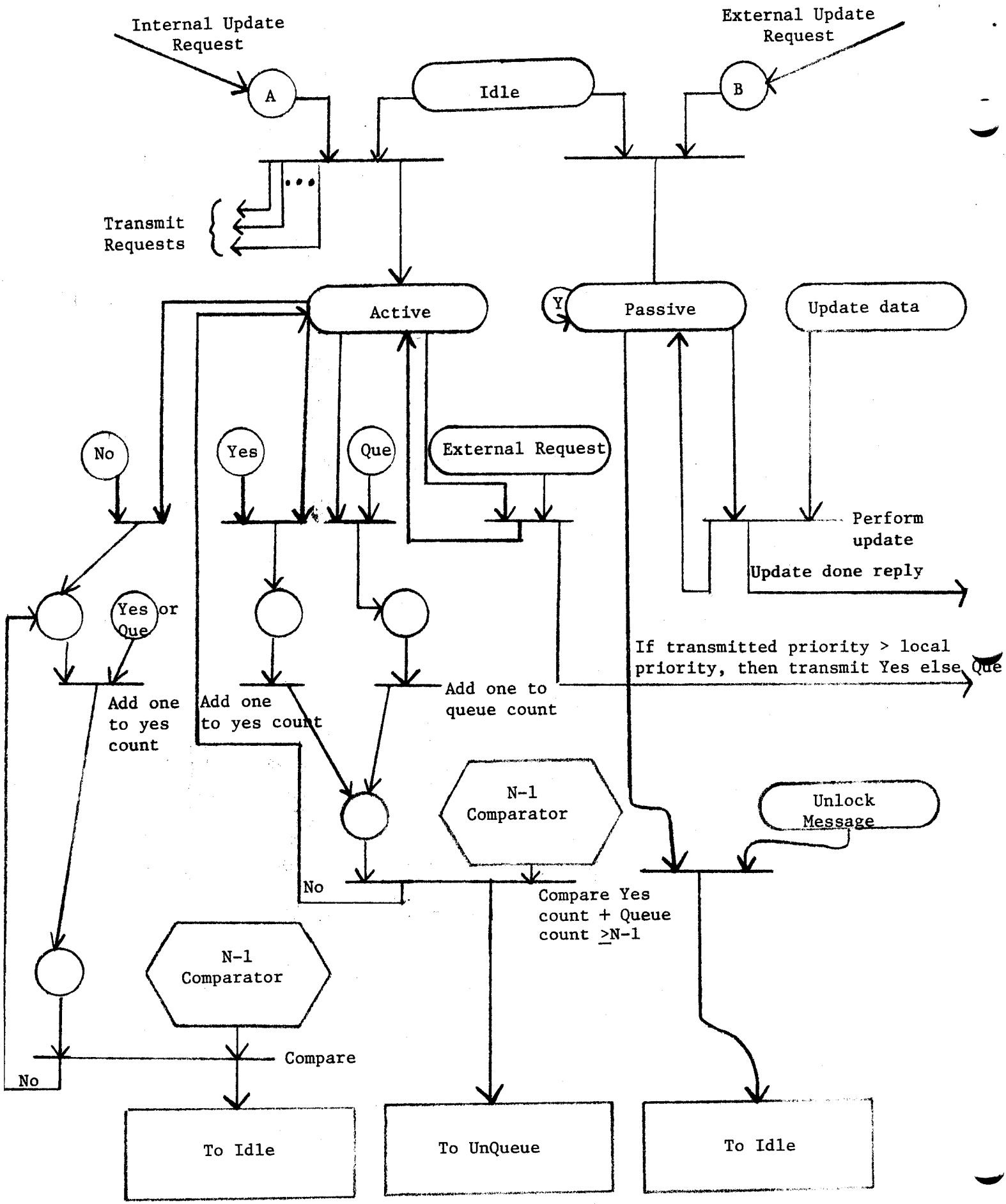
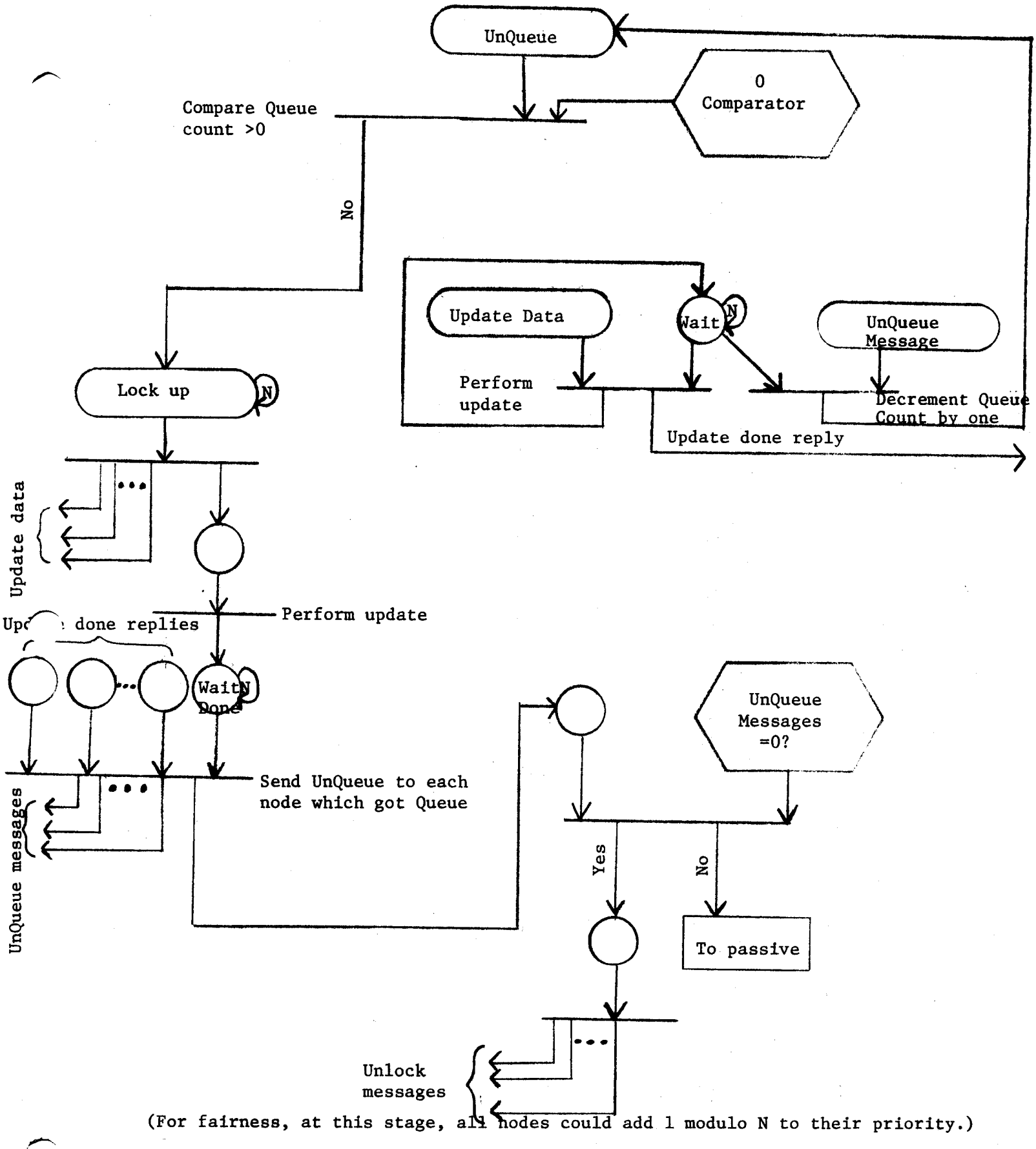



Figure 4a A Parallel Decentralized Solution




(For fairness, at this stage, all nodes could add 1 modulo N to their priority.)

Figure 4b A Parallel Decentralized Solution - continued

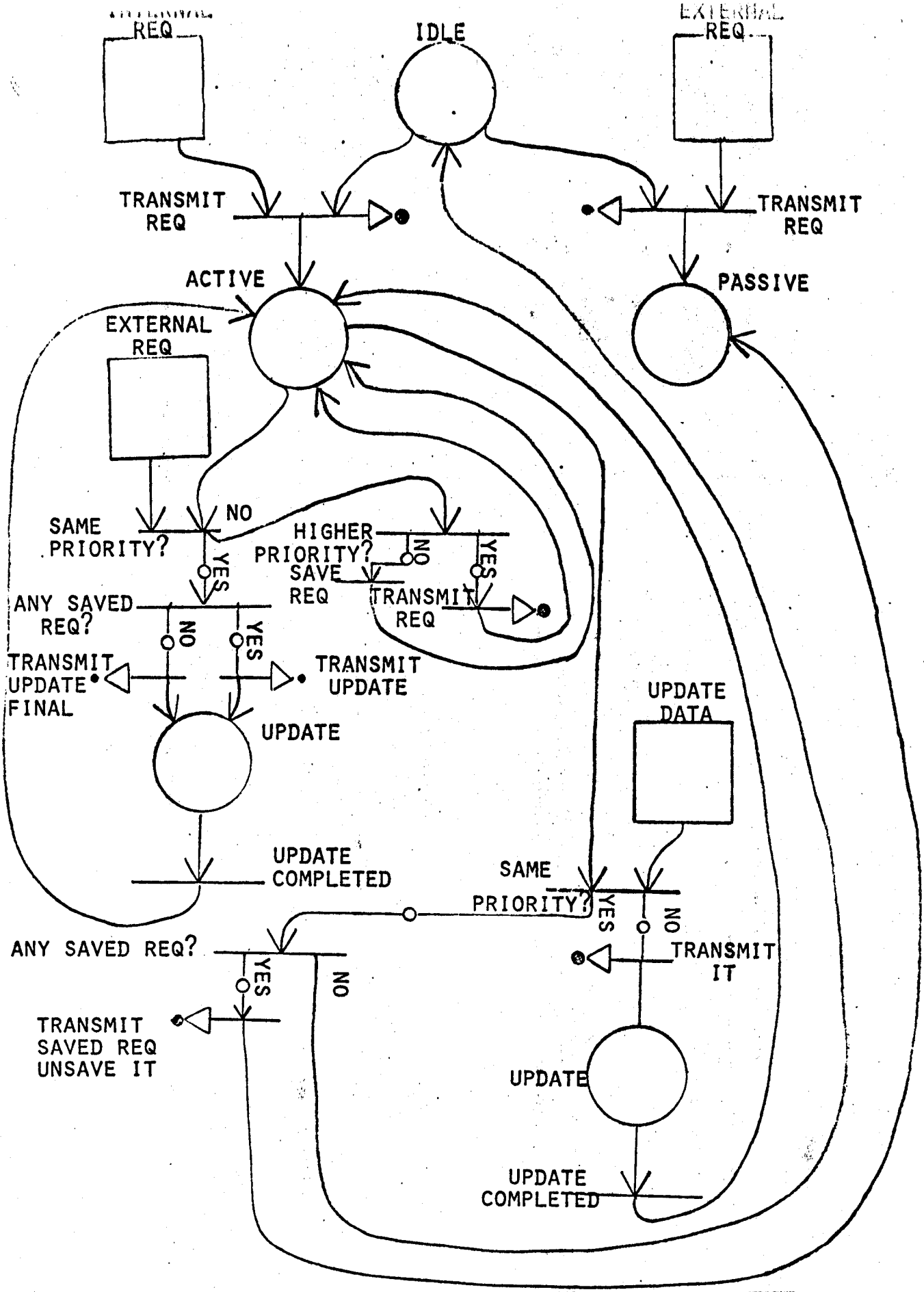



Figure 5a A Ring Structured Solution

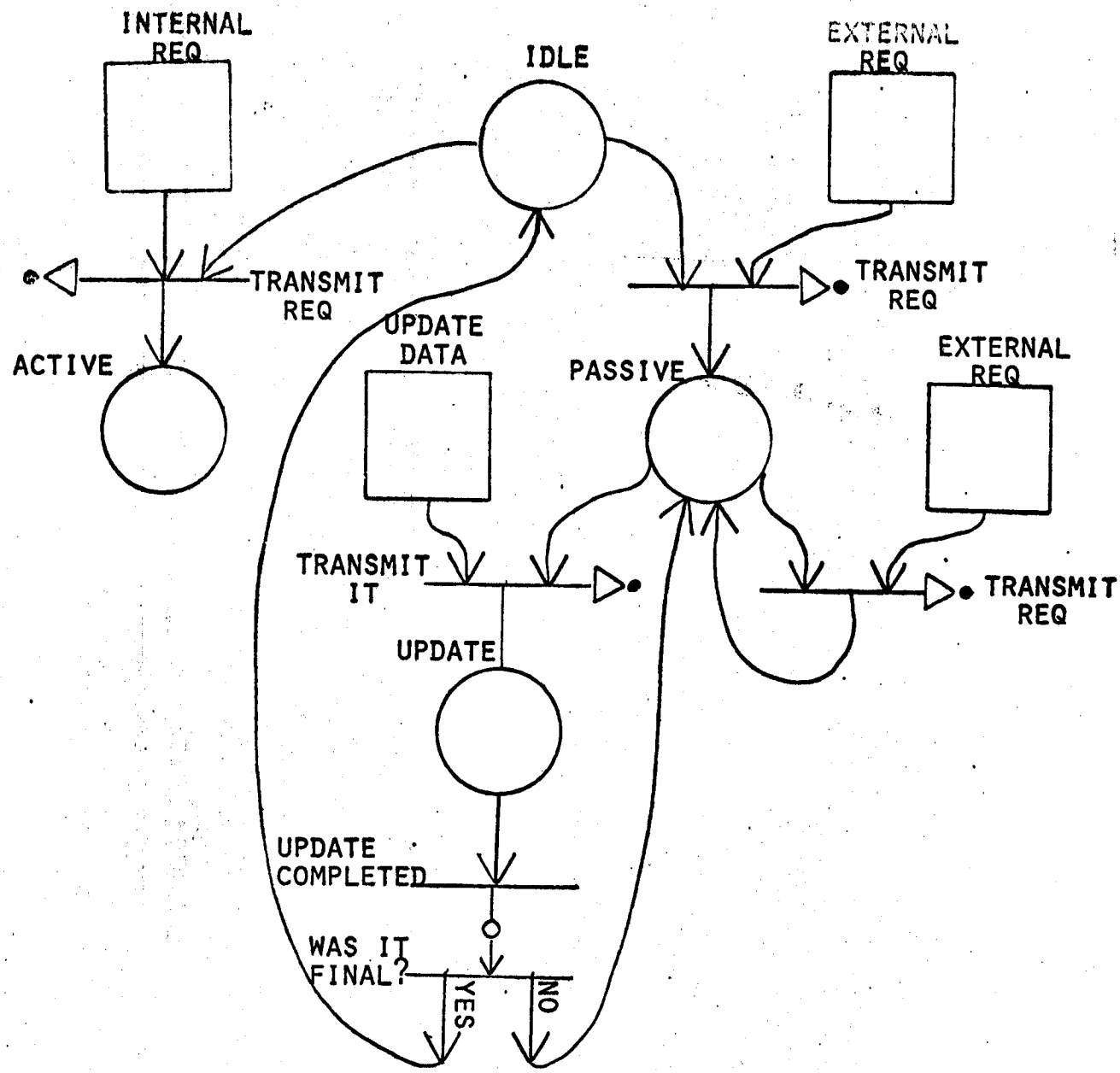



Figure 5b A Ring Structured Solution - continued

## 4. THE PROOF

The following discussion will briefly introduce the formalism of L systems using the notation of Rozenberg (20). It will then be shown that the question of whether a set of programs fulfills the criteria of consistency, speed independence, deadlock, and critical blocking is equivalent to certain decidable questions concerning the emptiness of particular sets of strings and sequences. A simple partial solution to the duplicate database problem will be used for illustrative purposes. A SNOBOL program which computes answers to these questions will be described in the Appendix.

Intuitively, L systems are similar to phrase structure grammars with the following significant alterations:

(1) Elimination of the distinction between terminal and nonterminal symbols so that all symbols in the vocabulary are assumed to be both terminal and nonterminal, and

(2) Simultaneous replacement of all symbols in a string at each derivation step.

The latter feature creates a very natural model of parallelism, and thereby motivates use of L systems for the study of asynchronous cooperating processes. The following are a few of the definitions of Ehrenfeucht and Rozenberg (19), presented in slightly simplified form.

Definition 1: An L system with tables and with interactions (abbreviated TIL system) is a construct $G = \langle \Sigma, P, g, w \rangle$, where $\Sigma$ is a finite nonempty set

called the alphabet of G, g is a symbol not in $\Sigma$ used as an end marker, w is a word over the alphabet $\Sigma$ called the axiom or initial string, and P is a finite nonempty set, such that each element $p \in P$ (called a table of G) is a finite nonempty relation satisfying the following:

$$P \subseteq \bigcup_{\substack{i,j,m,n \geq 0 \\ i+j = k \\ m+n = \ell}} \{g^i\}\ \Sigma^j \times \Sigma \times \Sigma^m \{g^n\} \times \Sigma^*$$

where $k \in N$ and $\ell \in N$. (N = set of natural numbers.)

and for every $\langle \alpha, a, \beta \rangle$ in $\bigcup_{\substack{i,j,m,n \geq 0 \\ i+j = k \\ m+n = \ell}} \{g^i\}\ \Sigma^j \times \Sigma \times \Sigma^m \{g^n\}$

there exists a $\gamma$ in $\Sigma^*$ such that $\langle \alpha, a, \beta, \gamma \rangle \in P$.

Each element of P is called a production and is usually written in the form $\langle \alpha, a, \beta \rangle \rightarrow \gamma$, or if the rule is context-free, it is written $a \rightarrow \gamma$ denoting $\langle \Lambda, a, \Lambda \rangle \rightarrow \gamma$, where $\Lambda$ means the empty string. In specifying productions in a table of a TIL system, one may omit those which cannot be used in any rewriting process which starts with the axiom of the system. This rewriting proceeds via derivations is explained next.

Definition 2: Let $G = \langle \Sigma, P, g, w \rangle$ be a TIL system.

Let $x = a_1 a_2 \ldots a_n \in \Sigma^*$ and $y \leftarrow \Sigma^*$. We say that x directly derives y in G (written $x \underset{G}{\Longrightarrow} y$) if

$y = \gamma_1\gamma_2\cdots\gamma_n$ **for some** $\gamma_1,\gamma_2,\ldots,\gamma_n \in \Sigma^*$ such that there exists a table P in $\Sigma$ **and for** every i in {1,2,...,n} , P contains a **production of** the form $\langle \alpha_i, a_i, \beta_i \rangle \rightarrow \gamma_i$ where $\alpha_i$ is the last k symbols of $g^k a_1 \ldots a_{i-1}$ and $\beta_i$ is the first $\ell$ symbols of $a_{i+1} \ldots a_n g^{\ell}$ . The transitive and reflexive closure of the relation $x \underset{G}{\Longrightarrow} y$ is denoted $x \overset{*}{\underset{G}{\Longrightarrow}} y$, then we say that x derives y in G.

Definition 3: Define the language generated by a TIL system G to be $L(G) = \{x \in \Sigma^* \mid w \overset{*}{\underset{G}{\Longrightarrow}} x\}$. A language generated by a TIL system is called an L language, or more specifically, a TIL language.

As an example, consider a network consisting of two nodes, each of which contains a copy of the duplicated database D. When a request to update is presented to node 1 from an applications process (AP), its database management process (DBMP) will perform the update and transmit the update to node 2. Before doing this, it will check whether node 2 is free (denoted by $T_2 = 0$) and if so, it will lock its own copy by setting $T_1 = 1$. After the update, it will set $T_1 = 0$ again. Note that the DBMP only changes $T_1$ which is its local variable (not duplicated at node 2). In order to test the nonlocal variable $T_2$, it must send a message to node 2 and wait for a reply. The exact protocol (which must also be performed by node 2 when it wants to update) is specified by A1 through F2.

Node i Protocol (i=1,2;j=1,2;i≠j)

A1. send a message to node j requesting update status

A2. wait for node j status reply

A3. if node j status reply is locked-for-update (Tj=1), then return to A1

A4. if node j status reply is unlocked (Tj=0), then proceed to B1

B1. change own status from unlocked to locked-for-update (set $T_i$=1)

C1. perform update on local copy of database

C2. transmit update to node j

C3. wait for update completed reply from node j

D1. change own status from locked to unlocked (set $T_i$=0)

E1. perform any non-update functions

E2. accept another update request from a local applications process, then return to A1

F1. whenever a status request message arrives from another DBMP, interrupt current processing, send reply indicating current value of $T_i$, then resume previous processing

F2. whenever an update arrives from another DBMP, interrupt current processing, perform update on local copy of database, send update completed reply, then resume previous processing.

The essence of the algorithms implied by the above protocol can be represented by flowcharts or graphs where each node is

given a label, a through e, corresponding to the label attached to statements within the protocol (Figure 6). Since we are not interested in the details of exactly what operations are performed within an update, the entire section of the protocol C1 through C3 is abstracted to a single flowchart box labelled $c_i$ which denotes an update on all copies of the database, instigated by node i. See Figure 6. Similarly, all code to perform non-update functions (protocol E1) plus the GO TO A1 statement (protocol E2) are represented by flowchart box $e_i$ and its out arc.

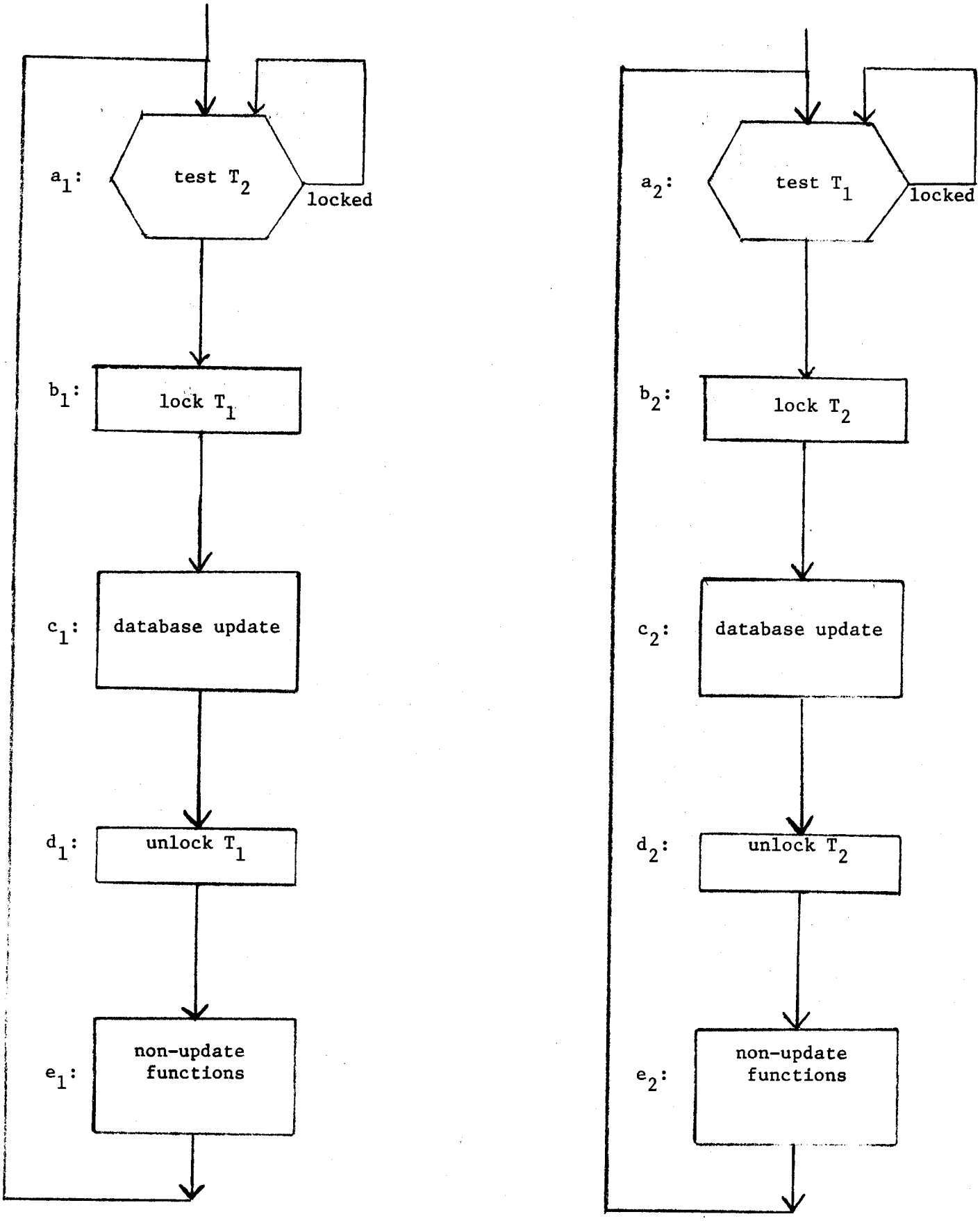



Figure 6 A 2 Node Network Solution

The state of the total system at any particular instant of time can then be captured by a string $S_1T_1S_2T_2$ where $S_i$ takes on values from the set $\Sigma_i = \{a_i,b_i,c_i,d_i,e_i\}$ denoting which box of the flowchart is currently being executed by process i. $T_i$ takes on a value of 1 meaning that process i is locking the database for its own exclusive use, or 0 implying that process i is not currently accessing the database (unlocked). The elements of these value sets will form the alphabet of the L system model. $\Sigma=\Sigma_1 U \Sigma_2 U\{0,1\}$ . Note that by subscripting the node labels we can model asymmetric solutions involving an arbitrary number of nodes although symmetrical solutions are suggested within our problem criteria. Elsewhere in this paper we will employ the notation $S_i(\gamma)$ to mean the value taken on by $S_i$ within string $\gamma$. Thus, within our example if $\gamma = a_1 0 c_2 1$, then $S_1(\gamma) = a_1$ and $S_2(\gamma) = c_2$. Given some instantaneous description, there are a finite number of possible 'next configurations' of the system which may be attained by some action by node 1, by node 2, or by both. This leads to the idea of 'rules of change' of the system which can be specified by productions within tables of the L system. Finally, the axiom w of the system specifies the flowchart box at which each node should start along with the initial values of variables. In our example, $w = a_1 0 a_2 0$. The formal specification of our system G includes

(a) $\Sigma = \{a_1, b_1, c_1, d_1, e_1, a_2, b_2, c_2, d_2, e_2, 0, 1\}$,

(b) $g = \#$,

(c) $w = a_1 0 a_2 0$

(d) P =

| | Table 1 | Table 2 |
|---|---|---|
| 1 | $a_1 \rightarrow a_1$ | |
| 2 | $\langle \Lambda, a_1, \delta\delta 0 \rangle \rightarrow b_1$ | Same as Table 1 |
| 3 | $\langle \Lambda, a_1, \delta\delta 1 \rangle \rightarrow a_1$ | |
| 4 | | $\langle \Lambda, b_1, 0 \rangle \rightarrow c_1$ |
| 5 | $b_1 \rightarrow b_1$ | $\langle b_1, 0, \Lambda \rangle \rightarrow 1$ |
| 6 | | $\langle \Lambda, b_1, 1 \rangle \rightarrow b_1$ |
| 7 | $c_1 \rightarrow c_1$ | |
| 8 | $c_1 \rightarrow d_1$ | Same as Table 1 |
| 9 | $e_1 \rightarrow e_1$ | |
| 10 | $e_1 \rightarrow a_1$ | |
| 11 | | $\langle \Lambda, d_1, 1 \rangle \rightarrow e_1$ |
| 12 | $d_1 \rightarrow d_1$ | $\langle d_1, 1, \Lambda \rangle \rightarrow 0$ |
| 13 | | $\langle \Lambda, d_1, 0 \rangle \rightarrow d_1$ |
| 14 | $0 \rightarrow 0$ | $\langle a_1 \mid c_1 \mid d_1 \mid e_1, 0, \Lambda \rangle \rightarrow 0$ |
| 15 | $1 \rightarrow 1$ | $\langle a_1 \mid b_1 \mid c_1 \mid e_1, 1, \Lambda \rangle \rightarrow 1$ |

The tables only give productions denoting the progress of node 1. Further productions would need to be specified to incorporate node 2. A computer program given graphs of any protocols for database access could construct the tables. It is also asserted that the program can check for correctness with respect to defined properties of a proposed solution to the duplicate database problem or any of a number of other synchronization problems. The basic technique for doing this is explained in the remainder of this paper. First, further interpretation of the tables is needed. Consider the case in which node 1 is in node state $a_1$, (i.e. flowchart box $a_1$) testing to see if it can update the database ($S_1 = a_1$) , node 2 is within its non-update section ($S_2 = e_2$), and $T_1 = 0$, $T_2 = 0$. This total state is represented by $\gamma = a_1 0 e_2 0$. To obtain one possible next total state, we must select a table (let's choose Table 1) and apply one production from this table to each symbol in $\gamma$. Thus, if node 1 receives a status=unlocked reply from node 2, $T_2 = 0$, it proceeds to step B1 (depicted by production 2, $\langle \Lambda, a_1, \delta\delta 0 \rangle \rightarrow b_1$) and node 2 leaves its non-update section ($e_2 \rightarrow a_2$) , then the corresponding derivation is $a_1 0 e_2 0 \Longrightarrow b_1 0 a_2 0$. This is due to the fact that the only choice of productions for 0 is $0 \rightarrow 0$. Note that $\langle \Lambda, a_1, \delta\delta 0 \rangle \rightarrow b_1$ means that $a_1$ can be replaced by $b_1$ if it is preceded by anything (left context is empty string $\Lambda$) and followed by any two symbols followed by a 0 ( $\delta$ implies a 'don't care' symbol). This derivation step shows an example of the simultaneous occurrence of two asynchronous events. Our assumption of speed independence

implies that a slow process should be able to remain in one node state for an arbitrary but finite number of transitions before moving to the next node state. Thus, for each symbol $\alpha \varepsilon \Sigma$, there must be a production $\alpha \rightarrow \alpha$. If in the above example, the transmission of message and reply takes a longer time to execute, then node 1 would not yet complete execution of flowchart box a. This situation can be depicted by utilizing production 1 of Table 1, $a_1 \rightarrow a_1$, to produce the derivation $a_1 0 e_2 \Longrightarrow a_1 0 a_2 0$. Table 2 is provided to guarantee that the action of leaving $b_1$ is coupled with the action of changing $T_1$ from 0 to 1 (see productions 4 and 5). Incorrect behavior would occur if one of these actions could take place without the other. Similarly, this table would be selected when it is time to change $T_1$ back from 1 to 0 when leaving $d_1$ (see productions 11 and 12).

Given this L system G as a model of the asynchronous programs of Figure 6, the question "Is it impossible for both programs to simultaneously perform (different) updates, thereby causing inconsistency?" can be cast as the following question concerning set emptiness within the model. Criterion 1: Is the set $C_1$ empty? $C_1 = \{\gamma \varepsilon L(G) \mid (c_i \varepsilon \gamma) \wedge (cj \varepsilon \gamma)$ for some $i,j \varepsilon N, i \neq j\}$. In the example just developed, this means that we consider the set of strings $\gamma$ containing both of the symbols $c_1$ and $c_2$, and inquire if any of the strings can be derived from $w=a_1 0 a_2 0$ (i.e. is $\gamma \varepsilon L(G)$). Since $c_i$ means that node i is updating the database, a string such as $\gamma = c_1 1 c_2 1$ within L(G) means that $C_1$ is non-empty. Thus, by a sequence of legal actions

specified by $w \overset{*}{\underset{G}{\Longrightarrow}} \gamma$, it is possible for both nodes to simultaneously update causing inconsistency. Similarly, it can be argued that an answer of 'yes' to the set inclusion question implies that there is no way to drive the system to a state such that both nodes are performing (different) updates. Thus, the protocol must be consistent. In the example, there is a legitimate derivation

$$w = a_1 0 a_2 0 \Longrightarrow b_1 0 b_2 0 \Longrightarrow c_1 1 b_2 0 \Longrightarrow c_1 1 c_2 1.$$

This L system fails to meet criterion 1, so the protocol (Figure 6) must fail to guarantee consistency. To insure consistency, we alter our protocol by putting step B1 before A1. This is depicted by the graph in Figure 7. If the L system modelling this protocol were examined, it would be found that the adult language [10, section 2.4] of G, denoted A(G), is non-empty. This is precisely the necessary and sufficient condition for (total) deadlock. Thus, we can state Criterion 2: Is the set A(G) empty? and the answer is 'yes' (i.e., criterion 2 is satisfied) if and only if the protocol is deadlock free. The related criterion of no critical blocking is satisfied if and only if the following question can be answered affirmatively. Criterion 3: Is the set $C_3$ empty? $C_3$ consists of the set of infinite derivations, $w=\gamma_0 \Longrightarrow \gamma_1 \Longrightarrow \gamma_2 \Longrightarrow \cdots$ such that

1. $(\exists_i)\,(\exists N)\ (\forall n>N, c_i \notin \gamma_n)$ and
2. $(\forall_i)\,((S_i(\gamma_k) = \alpha) \supset (\exists \ell > K \ni S_i(\gamma_\ell) \neq \alpha))$

The set $C_3$ consists of infinite derivations because it is necessary to capture the states of the system after an

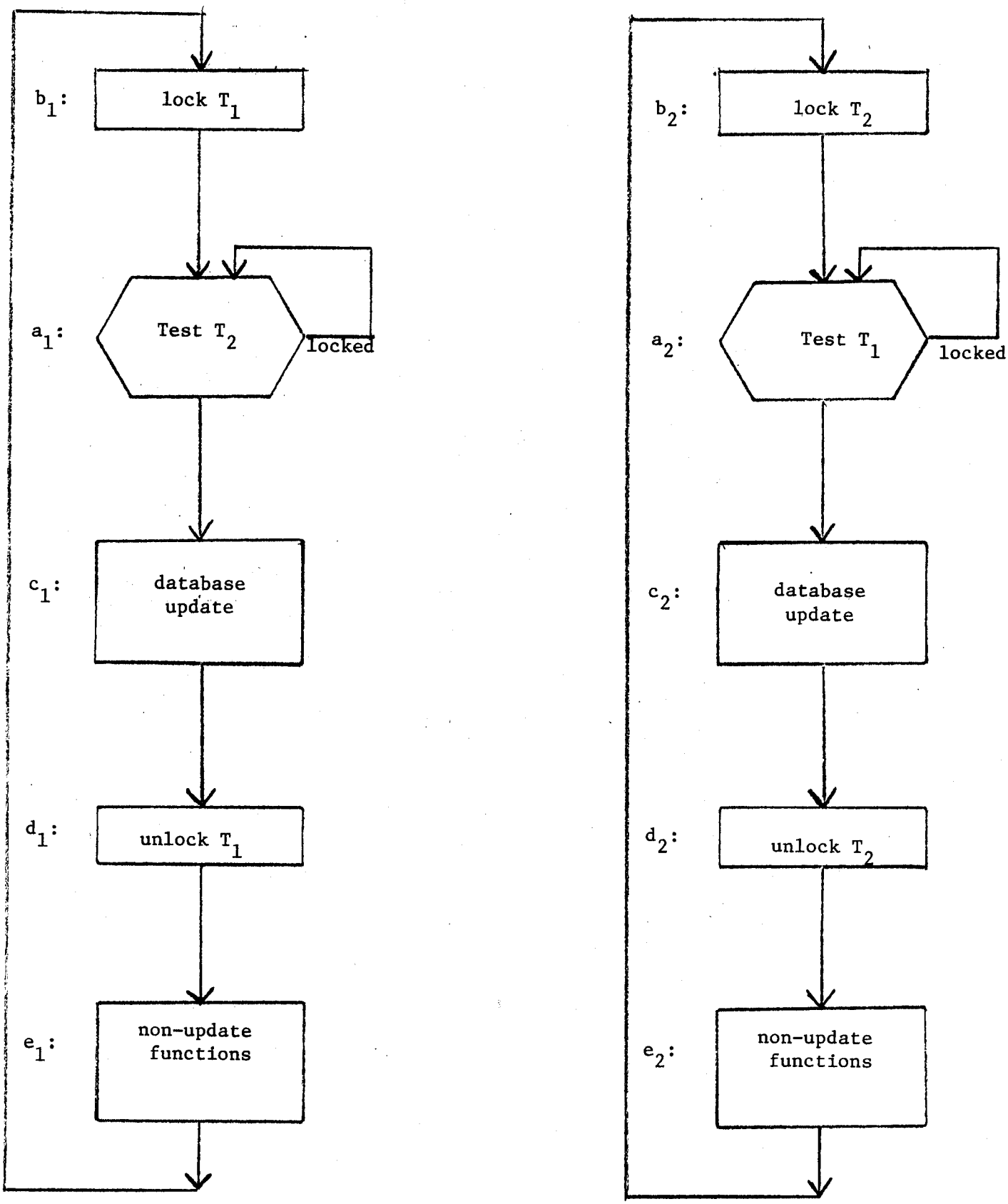



Figure 7 Altered 2 Node Network Solution

arbitrarily long initial period of time during which a node may loop within its control statements before entering its critical section. Part 1 of the specification of the set $C_3$ $[(\exists_i)(\exists N)(\forall n>N, c_i \notin \gamma_n)]$ states that criterion 3 will fail if there is any derivation such that after some initial period of time (depicted by $\gamma_0$ through $\gamma_N$) the symbol $c_i \varepsilon \Sigma$ (for some i) never appears. This means that the update section is never again entered by process i although it attempts to update. This is exactly the condition of critical blocking. Part 2 of the specification of $C_3[(\forall_i)((S_i(\gamma_k) = \alpha) \supset (\exists \ell k \ni S_i(\gamma_\ell) \neq \alpha)]$ is appended to rule out the possibility that some process might take an infinitely long time to perform some operation. Recall that the productions $\alpha \rightarrow \alpha$ were added to allow speed independence, and that operations were allowed to take arbitrary but finite amounts of time. Thus, we must only consider sequences which for each node state $S_i$ do not consist only of applications of the production $\alpha \rightarrow \alpha$. This is indicated by the formal statement: if $S_i$ takes on value $\alpha$ in the k-th string of a derivation $(S_i(\gamma_k) = \alpha)$, then $S_i$ must take on another value at some later time $(\exists \ell > k \ni S_i(\gamma_\ell) \neq \alpha)$. Finally, the criterion of partial operability (criterion 4) can be examined by simply applying the previous three set emptiness questions to the following subsets of L(G): For each $\gamma' \varepsilon L(G)$ containing less than n (but more than zero) symbols of the form $e_i$ where n = number of processes, apply all of the above emptiness tests to L(G') where $G' = (\Sigma', P', \#, \gamma')$ where $\Sigma'$ is the set of all symbols in $\Sigma$ except those symbols in

$\Sigma_i - e_i$ for each $e_i \varepsilon \gamma'$ . P' is the restriction of P to $\Sigma'$. Thus, only productions which generate valid strings of symbols over $\Sigma'$ can be used. This means that $e_i \rightarrow e_i$ must always be employed. Thus, in considering blocking and deadlock, the nodes which are effectively stopped should not be considered.

For any L system G formed from any finite set of nodes plus their protocols, the set emptiness questions are decidable, and so the four conditions can be mechanically verified to prove or disprove the correctness of a solution. A brief sketch of the argument verifying that criterion 3 is decidable will be given here. This condition is selected because it involves infinite sequences and thus is one of the least obviously decidable conditions.

Proposition: The question "Is $C_3$ empty?" can be answered after a finite number of test steps.

Justification:

a. In considering infinite sequences of strings $\gamma_0, \gamma_1, \gamma_2, \ldots$ as possible members of $C_3$, it is sufficient to only consider those sequences such that $\gamma_{i+1} \neq \gamma_i$. This is true because given any sequence, it is in $C_3$ if and only if its underlying real-time subsequence is in $C_3$. This subsequence is obtained by collapsing $\gamma_i, \gamma_{i+1}, \ldots, \gamma_\ell$ to $\gamma_i$ for each case of $\gamma_{i-1} \neq \gamma_i = \gamma_{i+1} = \ldots = \gamma_{\ell-1} = \gamma_\ell \neq \gamma_{\ell+1}$ in the original sequence. The real-time subsequence is valid because $\gamma_{i-1} \Longrightarrow \gamma_i = \gamma_\ell \Longrightarrow \gamma_{\ell+1}$ implies $\gamma_{i-1} \Longrightarrow \gamma_i \Longrightarrow \gamma_{\ell+1}$. The real-time subsequence is infinite because repetitions

$,\gamma_i = \gamma_{i+1} = \ldots,$ are guaranteed to be finite by part 2 of the specification of $C_3$. Notice that deadlocked sequences are not in the set $C_3$ because they do not fulfill the criterion for critical blocking. They are instead detected by criterion 2.

b. In considering infinite real-time sequences (i.e. $\gamma_{i+1} \neq \gamma_i$) as possible members of $C_3$, it is sufficient to only consider finite sequences of length $|L(G)| + 1$. We inquire if there is any sequence of this length beginning with any $\gamma \varepsilon L(G)$ which does not contain one or more $c_i \varepsilon \Sigma$. If so, then since L(G) only contains a finite number of strings, there must be a (nontrivial) 'loop' in which some $\gamma_i \overset{*}{\Longrightarrow} \gamma_i$. This loop can be repeated indefinitely to obtain an infinite sequence excluding $C_i$. Finally, since $\gamma_i \varepsilon L(G)$, $w = \gamma_o \overset{*}{\Longrightarrow} \gamma_i$. Thus, a sequence beginning with $\gamma_o$ can be constructed. Conversely, if there is some real-time sequence in $C_3$, then some $\gamma_i \varepsilon L(G)$ must be repeated in this sequence within $|L(G)| + 1$ derivation steps and there must be some $c_i$ not contained in any of the strings within these derivation steps. This situation will then be detected by simply looking at finite strings of length $|L(G)| + 1$.

This completes the justification of the decidability of criterion 3. One crucial factor was the finite fixed size of L(G). However, current systems allow dynamic requests for resources and dynamic creation of processes and subprocesses by any existing process. This implies that strings of L(G) would not all be of the same length. The area of dynamic systems presents some even more tantalizing problems than the one

investigated here. These dynamic systems problems further justify use of L systems. In presenting the above verification techniques, the duplicate database problem was used as an example. These techniques can be similarly applied to the solution verification of other problems in synchronization and control of asynchronous systems. A SNOBOL program which performs the verification of criteria 1 through 4 is described in the Appendix. The output of this program is shown when an L system modeling the proposed solution of Figure 7 is presented as input.

In summary, the very interesting duplicate database problem has been explored; some solutions and a verification technique have been presented; and a new tool (L systems) for network operating systems analysis has been briefly explored. Areas of future study are apparent in the further application of theorems of L systems to this problem and others. The complexity and cost-effectiveness (in terms of number of transmissions, etc.) of solutions also warrant more study. Finally, the solutions to the duplicate database problem presented need to be expanded and refined to include such features as time-outs, and error handling and recovery.

Acknowledgements

Thanks are due to the numerous talks with numerous persons at Yorktown Heights for useful discussions and a generally stimulating creative environment in which to discover and work upon the problem presented here. Thanks also to Professor Mike Hammer and Harry Forsdick of MIT who provided valuable reference papers and invaluable discussions.

Clarence A. Ellis

April 1976

BIBLIOGRAPHY


1. Brinch-Hansen, P., Operating Systems Principles, Prentice-Hall, N.J., 1973.
2. Chamberlin, Boyce, and Traiger, A Deadlock Free Scheme for Resource Locking in a Database Environment, IFIPS Conference, 1974.
3. Chandra, A.N., Howe, W.C., and Karp, D.P., Deadlock in Loosely Coupled Computing Facilities, IBM Report, Nov. 1971.
4. Coffman, E., Elphick, M., and Shoshani, A., System Deadlocks, Computing Surveys 3,2.
5. Denning, P. and Coffman, E., Operating Systems Theory, Prentice-Hall, N.J., 1973.
6. Eswaran, Gray, Lorie, and Traiger, On the Notion of Consistency and Predicate Locking in a Database Environment, IBM Report No. R.J. 1487, Dec. 1974.
7. Forsdick, H.C., A Comparison of Two Schemes that Control Multiple Updating of Data Bases, MIT, May 1975.
8. Habermann, A.N., Synchronization of Communicating Processes, CACM, 15,3.
9. Holt, A.W., et.al., Information System Theory Report, Applied Data Research Inc. Technical Report No. RADC-TR-68-305, 1968.
10. Holt, A.F. Commoner, Events and Conditions, Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, pp. 3-52, 1970.
11. Johnson, P.R. and Thomas, R.H., The Maintenance of Duplicate

Databases, Network working group RFC 677 working paper, Jan. 1975.

12. Lindenmayer, A. and Rozenberg, G., Developmental Systems and Languages, 4th ACM STOC Conference, 1972.
13. Lamport, L., Comments on Duplicate Databases, Mass. Computer Associates report.
14. Mullery, A.P., The Distributed Control of Multiple Copies of Data, IBM Report RC 3642, Dec. 1971.
15. Noe, J.D., The CDC 6400, Proc. of ACM Workshop on System Performance Evaluation, 1971.
16. Nutt, G.J., The Formulation and Application of Evaluation Nets, PhD Dissertation, Univ. of Washington, 1972.
17. Nutt, G.J., Evaluation Nets for Computer Performance Analysis, Proc. of FJCC, 1972.
18. Petri, C.A., Kommunikation mit Automaten, PhD Dissertation, Univ. of Bonn, 1962.
19. Rozenberg, G., L Systems, Springer Verlag, N.Y., 1974.
20. Rozenberg, G., TOL Systems and Languages, Information and Control. v. 23.
21. Information Management System Manual, IBM Publication LY 20-0629-4.
22. Holler, E., "Files in Computer Networks", in Proceedings of IRIA Conference on Networks, 1975, Springer Verlag.

# APPENDIX

The program listed in this appendix accepts a set of L system tables plus an initializing axiom as inputs, and performs tests of deadlock, consistency and critical blocking on the modelled system. The program uses a depth first recursive descent technique to test the set inclusion questions presented in Section 4, and prints as output a list of tests which failed along with the strings associated with each failure. The program is written in the SNOBOL language, consists of 161 source statements, and utilizes 256K bytes of memory. The program when running a typical modelled system (32 symbols, 4 tables of 26 productions each) on the IBM 360/91 requires approximately two minutes of execution time.

To use the program, data specifying the L system must be inserted immediately after the END statement. All data begins in column 1 and contains no intermediate blanks. The first card specifies the number of productions; the second card specifies an upper bound on the size of an internal stack (typically 128 is sufficient), the third specifies an axiom, and the fourth specifies the inconsistency conditions separated by commas. The cards following this contain the productions, one per card. In order to specify which productions are in which table, each group of production cards is preceded by a table card telling the number of the table for that group. The table card has the following format where XX denotes a two digit table number

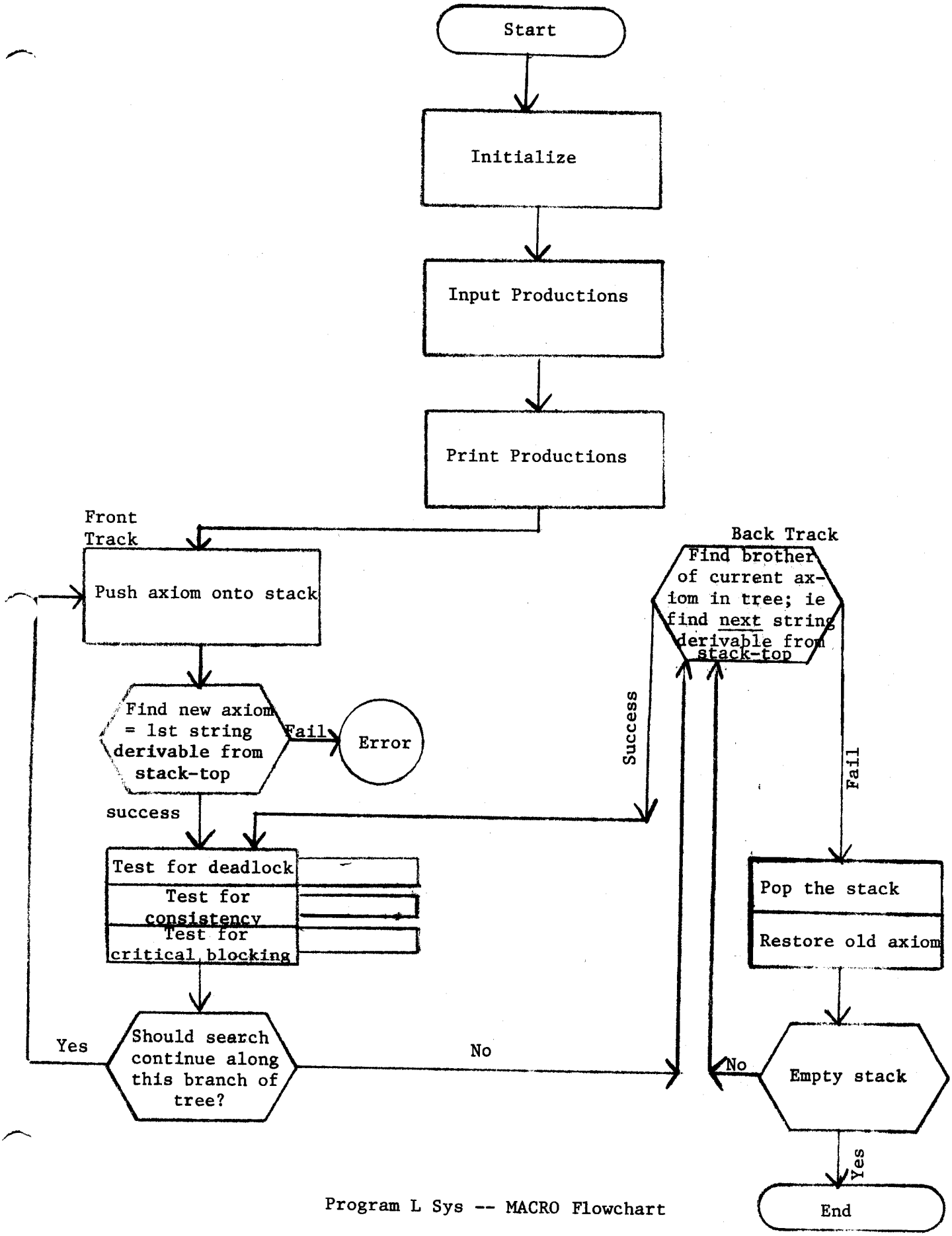
(beginning with 01), YY denotes the number of productions in the group, and b denotes a blank:

TABLEbXXbYY

A production card has the following format starting in column one with no interspersed blanks:

lc,aa,rc=W1 W2 ... WN

as denotes the symbol to be replaced; each Wi denotes a string of symbols which is a possible replacement of aa. lc and rc denote strings of symbols which are the left context and right context of aa, respectively. Under the current implementation, each symbol of the alphabet $\Sigma$, such as aa above, must be represented by three characters (e.g. A11,BO3,XK5, are three legitimate symbols). lc and/or rc may be the empty string but the commas (and no blanks) are still required. The program may halt printing an error message if incorrectly formatted data is entered. Other error messages may result from a bad L system in which there are no replacement productions for a symbol occurring in a string. Intermediate output and diagnostic trace information can be obtained by changing the first statement of the program from PRNTSWITCH = 0 to PRNTSWITCH = 2. A lesser amount of intermediate output can be obtained by setting PRNTSWITCH = 1. These outputs may be used in conjunction with the flowchart which follows plus the commented program to understand more details about the actions of the program upon a particular input data set.




Program L Sys -- MACRO Flowchart

```
*            LSYS PROGRAM
*
* THIS PROGRAM VALIDATES CONDITIONS OF PARALLEL SYSTEMS
* THE TECHNIQUE USED INVOLVES INPUTTING A SET OF TABLES
* THE TABLES PROVIDE A FORMAL SPECIFICATION OF THE SYSTEM MODELLED
* THEOREMS OF L SYSTEMS CAN BE APPLIED TO TEST THE MODELLED SYSTEM
* THE PROGRAM TESTS FOR DEADLOCK, CONSISTENCY, AND CRITICAL BLOCKING
* FURTHER TESTS (E.G. PARTIAL OPERABILITY) MAY BE INCORPORATED LATER
*
*
*
      PRNTSWITCH = 0
      MAXPNTR = 2
*   SET APARAM TO # LOOP TIMES - 1
      APARAM = 0
*
*            FUNCTIONS
      DEFINE('TREER()')
      DEFINE('COMPAR()')
      DEFINE('PRINTSTK()')
      DEFINE('MUTEX()')
      DEFINE('DEADL()')
      DEFINE('CRBLOCK()')
         DEFINE('NXTABLE(B)')
         DEFINE('NXSTRING(B)')
         DEFINE('NXCHAR(A,B)')
      OUTPUT = ' BEGUN '
*
*              KEYWORDS
      &STLIMIT = 500000
      &TRIM = 1
      &DUMP = 1
      &TRACE = 500
      LT(PRNTSWITCH,2)                    :S(NOPRNT1)
      &FTRACE = 500
NOPRNT1
      LT(PRNTSWITCH,1)                    :S(NOPRNT2)
      TRACE('CURRENT')
      TRACE('PNTR')
NOPRNT2
*
```

```
*              INITIAL INPUTS
MAINR PRODSIZE = INPUT
      PROD = TABLE(PRODSIZE)
         STKSIZE = INPUT
*  BEWARE STK SIZE ESTIMATE TOO SMALL
         STK =          ARRAY(STKSIZE)
         TABLNSTK =     ARRAY(STKSIZE)
      AXIOM = INPUT
      OUTPUT =
      OUTPUT =    'AXIOM IS ' AXIOM
*  INPUT  INCONSISTENCY  CONDITIONS
          BADS = ARRAY(8)
          INCON = INPUT
      INCO = INCON
      OUTPUT =  'INCONSISTENCY CONDITIONS ARE ' INCO
      OUTPUT =
          I = 1
CONSIS INCON BREAK(',') . BADS<I> ',' REM . INCON :F(CONSIS2)
          I = I + 1                          :(CONSIS)
CONSIS2    BADS<I> = INCON
          NUMOFCONDS = I
         OUTPUT = '      TABLES'
FRST     RDIN = INPUT                        :F(NXT)
         RDIN 'TABLE ' LEN(2) . TABLEN ' ' LEN(2) . TABLESIZE :F(NXT)
         OUTPUT = 'TABLE ' TABLEN '  SIZE ' TABLESIZE
         I = 1
MORE     RDIN = INPUT
         OUTPUT = RDIN
         RDIN BREAK(',') . LEFTCONTEXT ',' BREAK(',') . GENERATRIX
+                    RTAB(0) . REST      :F(ERR1)
         G = TABLEN GENERATRIX
         IDENT(PROD<G>)                      :F(APPEND)
         PROD<G> = LEFTCONTEXT REST
CONT     I = LT(I,TABLESIZE) I + 1          :S(MORE)F(FRST)
APPEND   PROD<G> = PROD<G> '+' LEFTCONTEXT REST :(CONT)
NXT      OUTPUT = RDIN
         TABLEMAX = TABLEN
         LT(PRNTSWITCH,2)                   :S(GOON)
      OUTPUT = ' PRODUCTIONS '
      A = CONVERT(PROD,'ARRAY')
      I = 1
NEXTPROD OUTPUT = A<I,1> ' YIELDS ' A<I,2>    :F(GOON)
         I = I + 1                          :(NEXTPROD)
ERR1     OUTPUT = 'PROD CARD IN WRONG FORMAT=' RDIN  :(END)
* CALL PRIMARY SUBROUTINE TO TRAVERSE TREE
GOON  OUTPUT = TREER()
      I = 1
LOP   OUTPUT = 'STACK(' I ')=' STK<I>
      I = LT(I,MAXPNTR) I + 1              :S(LOP)
      OUTPUT = '  AXIOM IS ' AXIOM
      OUTPUT = '  INCONSISTENCIES ARE  ' INCO
      APARAM = NE(APARAM,0) APARAM - 1 :S(MAINR)F(END)
*
```

```
*            FUNCTION BODIES
MUTEX  I = 1
MUTEXLOOP IDENT(CURRENT,BADS<I>)          :S(MUTEXOUT)
       I = LT(I,NUMOFCONDS) I + 1         :S(MUTEXLOOP)F(RETURN)
MUTEXOUT OUTPUT = ' + + + THIS SOLUTION FAILS + + +'
       PRINTSTK()
          OUTPUT = '  VIOLATION OF DATABASE CONSISTENCY VIA STRING '
+                  CURRENT                 :(END)
DEADL  DEADLOCK = EQ(B,0) 'NO'            :F(DEAD2)
       DEADLOCK = IDENT(CURRENT,STK<PNTR>) 'YES' :(RETURN)
DEAD2  DEADLOCK = DIFFER(CURRENT,STK<PNTR>) 'NO' :(RETURN)
DEADOUT DEADLOCK = IDENT(DEADLOCK,'YES') 'NO' :F(POPUP)
          OUTPUT = ' + + + THIS SOLUTION FAILS + + +'
       PRINTSTK()
 OUTPUT = 'VIOLATION OF DEADLOCK CRITERION VIA STRING ' CURRENT :(END)
CRBLOCK   :(RETURN)
*  COMPARISON FUNCTION BODY
COMPAR I = PNTR
COMPLOOP IDENT(STK<I>,CURRENT)            :S(RETURN)
       I = NE(I,1) I - 1                  :F(FRETURN)S(COMPLOOP)
*   PRINT STACK SUBROUTINE
PRINTSTK I = GE(PRNTSWITCH,2) 1           :F(RETURN)
PRLOOP   OUTPUT = I '  ' STK<I> '  ' TABLNSTK<I>
         I = LT(I,PNTR) I + 1             :S(PRLOOP)F(RETURN)
*
*
*
*
* PRIMARY SUBROUTINE BODY
TREER CURRENT = AXIOM
      PNTR = 0
* FRONT TRACK GOES DOWN TREE
FRONTT PNTR = PNTR + 1
       STK<PNTR> = CURRENT
       MAXPNTR = LT(MAXPNTR,PNTR) PNTR
         TABLNSTK<PNTR> = TABLEN
         TABLEN = 1
       B = 0
         CURRENT = NXTABLE('0')           :F(ERR3)
* CORRECTNESS TESTS
T1      MUTEX()
T2     DEADL()
T3     CRBLOCK()
* COMPARE CURRENT TO ALL IN STACK
       COMPAR()                           :F(FRONTT)
*  BACKTRACK REVERSES UP TREE
BACKT B = 1
         CURRENT = NXTABLE('1')           :S(T1)
       STCT = &STLIMIT - 10000
       GT(&STCOUNT,STCT)                  :F(STGOON)
       &STLIMIT = &STLIMIT + 10000
STGOON
       LT(PRNTSWITCH,1)                   :S(NOPRNT3)
       PRINTSTK()
NOPRNT3
                                          :(DEADOUT)
POPUP  TREER = EQ(PNTR,1) ' FINISHED '    :S(RETURN)
       CURRENT = STK<PNTR>
         TABLEN = TABLNSTK<PNTR>
       PNTR = PNTR - 1                    :(BACKT)
ERR3     OUTPUT = '  BAD GRAMMAR '  STK<PNTR>  :(END)
```

```
*
*
*
*  SECONDARY SUBROUTINES TO SEARCH TABLES
NXTABLE  TABLEN = EQ(B,0) 1
TBLOOP   NXTABLE = NXSTRING(B)          :S(RETURN)
         TABLEN = LT(TABLEN,TABLEMAX) TABLEN + 1 :F(FRETURN)
         B = 0                          :(TBLOOP)
NXSTRING FSTRING =
         POINT = 3
         LSTRING = NE(B,0) CURRENT      :S(GOBBLE)
NXS1 NXSTRING = NXSTRING NXCHAR(POINT,'0') :F(FRETURN)
         POINT = LT(POINT,SIZE(AXIOM))   POINT + 3 :S(NXS1)F(RETURN)
GOBBLE LSTRING TAB(3) REM . LSTRING
         NXSTRING = FSTRING NXCHAR(POINT,'1') LSTRING :S(RETURN)
         FSTRING = FSTRING NXCHAR(POINT,'0') :F(ER1)
         POINT = LT(POINT,SIZE(CURRENT)) POINT + 3 :S(GOBBLE)F(FRETURN)
ER1      OUTPUT = 'NO MATCH FOR ' STK<PNTR> ' AT POINTER ' POINT :(END)
NXCHAR J = A - 3
         STK<PNTR> TAB(J) . LEFT TAB(A) . ID REM . RIGHT
         CURRENT TAB(J)   TAB(A) . DI
         ID = TABLEN ID
         ID = LT(SIZE(ID),5) '0' ID
RESTART  WORK = PROD<ID>
TST WORK BREAK(',') . TRY ',' BREAK('=') . TRYY '=' REM . WORK :F(ERR5)
         LFT = LEFT
         RGHT = RIGHT
* TEST A PRODUCTIONS CONTEXT
*   LEFT
TSTL     TRY RTAB(3) . TRY RTAB(0) . TRY3 :F(TSTR)
         LFT RTAB(3) . LFT RTAB(0) . LFT3 :F(SKIP)
         IDENT(TRY3,'***')                :S(TSTL)
         IDENT(TRY3,LFT3)                 :S(TSTL)F(SKIP)
*  RIGHT
TSTR     TRYY TAB(3) . TRY3 REM . TRYY    :F(OKC)
         RGHT TAB(3) . RGHT3 REM . RGHT   :F(SKIP)
         IDENT(TRY3,'***')                :S(TSTR)
         IDENT(TRY3,RGHT3)                :S(TSTR)F(SKIP)
OKC      EQ(B)                            :F(PT2)
PT1      WORK TAB(3) . NXCHAR             :S(RETURN)F(ERR4)
PT2      WORK = WORK
         WORK BREAK('+') . WORK
         WORK ANY DI REM . WORK           :F(SKIP)
         B = IDENT(WORK) 0                :S(SKIP)
         WORK '|' LEN(3) . NXCHAR         :S(RETURN)F(ERR4)
* KEEP ON LOOKING FOR     REPLACEMENT '
SKIP     WORK BREAK('+') '+' REM . WORK   :S(TST)F(FRETURN)
ERR4     OUTPUT = ' INVALID PRODUCTION FOR ' A '  ' CURRENT :(END)
ERR5     OUTPUT = ' NO MATCH FOR CURRENT ' A '  ' CURRENT ' ' WORK :(END)
END
```

NO ERRORS DETECTED IN SOURCE PROGRAM

```
 BEGUN

AXIOM IS B11T10B22T20
INCONSIS TENCY CONDITIONS ARE C11T11C22T21,C11T10C22T20

     TABLES
TABLE 01  SIZE 40
,A11,******T20=C11
,A11,=A11
,B11,=B13
,B22,=B23
B13,T10,=T13
B23,T20,=T23
,B13,T13=A11
,B23,T23=A22
B13,T13,=T11
B23,T23,=T21
,B13,T10=B13
,B23,T20=B23
,B13,T11=B13
,B23,T21=B23
,B11,=B11
,B22,=B22
,C11,=D11|C11
,E11,=A11|E11
,D11,=D14
,D22,=D24
D14,T11,=T14
D24,T21,=T24
,D14,T14=E11
,D24,T24=E22
D14,T14,=T10
D24,T24,=T20
,D14,T10=D14
,D24,T20=D24
,D14,T11=D14
,D24,T21=D24
,D11,=D11
,D22,=D22
,T10,=T10
,T11,=T11
T10,A22,=C22
,A22,=A22
,C22,=D22|C22
,E22,=A22|E22
,T20,=T20
,T21,=T21
END OF TABLES
 + + + THIS SOLUTION FAILS + + +
VIOLATION OF DEADLOCK CRITERION VIA STRING A11T11A22T21
```