

THE VALIDATION OF
PARALLEL CO-OPERATING PROCESSES

by

Clarence A. Ellis
Department of Computer Science
University of Colorado
Boulder, Colorado 80302

Report #CU-CS-065-75

April 1975

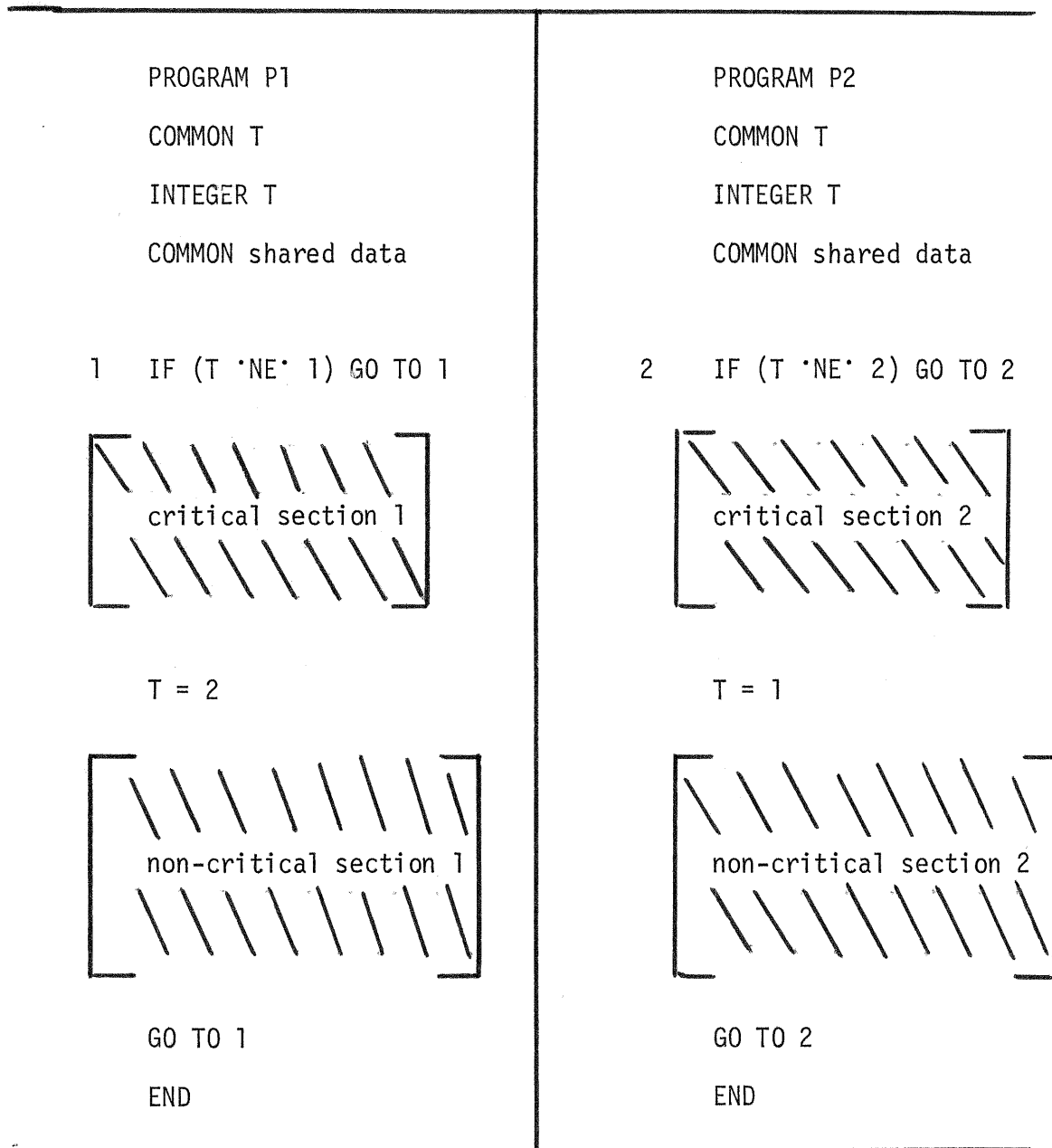
THE VALIDATION OF PARALLEL CO-OPERATING PROCESSES

Why has the state-of-the art in Operating Systems implementation not reached the quality of construction found in present day compilers, and other programs? Brinch-Hansen in his book [1] states that this situation is due to a lack of understanding of the principles common to all modern operating systems. I feel that another reason is the added complexity due to parallelism. There may be many tasks (i.e., processes) executing simultaneously: An IBM 360/370 series machine has many channels plus one or more CPUs accessing memories in parallel; A CDC 6000 series machine has complex co-operation between a central processor and 10 to 20 peripheral processors; Any multiprogramming system has more than one job in memory and competing for resources at any one time. The above illustrations of parallel co-operating processes all imply an added dimension of complexity due to parallelism which frequently turns out to be the source of a surprisingly large number of (often irreproducible) errors in present-day systems, (see Randall [2]). This paper presents a simple example of some types of errors which can occur between two (or more) programs which are simultaneously executing and attempt to synchronize. It does not matter whether these programs are running on a multiprocessor system or on a multiprogramming uniprocessor system. Next, the paper illustrates how the formalism of L systems [9-12] can be applied to this (or any) set of programs, and used to rigorously prove the correctness or incorrectness of their interaction. Furthermore, the proof procedure is effective, and thus can be mechanically carried out on a computer. This paper is a report on preliminary research so, for example, it is

not known if efficient computer techniques can be devised to make mechanical proofs feasible on realistically large sets of programs. There exists a wide range of problems similar in nature to the problem discussed here for which this formalism may be useful. These problems are not discussed herein, and remain thus far unexplored.

Consider the two FORTRAN programs, P1 and P2, shown in figure 1, and suppose that they are simultaneously executing on two processors within a multiprocessor system. The portion of each program in brackets labelled critical section is a sequence of statements which do not access any shared data. The two programs are supposed to continually cycle through their critical then their non-critical sections; but we assume that if both programs are in their critical sections at the same time, then errors can occur. Thus, the global integer variable T and some control statements are used as a synchronization mechanism by the programs to communicate and thereby insure that only one of the programs is executing code inside of its critical section at a time. The formal proof of correctness of this synchronization mechanism (and others) is the subject of this paper.

Figure 1



* Assume variable T is initially set to 1

Do the control statements in the above programs insure that at most one of the two parallel co-operating processes is within its critical section at any time? This is one instance of the mutual exclusion problem discussed by Dijkstra [4,5]. It is worthwhile to digress slightly to explain why this problem is of significance. Suppose the critical section of Fortran code in both P1 and P2 is simply $X = X+1$ where X is in common. Then the compiled machine language code for this Fortran statement, assuming a 1-address machine, might consist of three instructions:

1. load accumulator from location X ,
2. add one to accumulator,
3. store accumulator into location X .

We assume that each of the two processors, executing P1 and P2 respectively, have their own accumulators. Next, assume that X is supposed to count the number of times that critical sections are executed, so X is initially set to zero. If one processor and then the other enters and executes the critical three instructions, then the value of X will be two because of the sequence:

1	load accumulator	1	←	X has value 0
2	add 1 to accumulator	1		
3	store accumulator	1	←	X has value 1
4	load accumulator	2	←	X has value 1
5	add 1 to accumulator	2		
6	store accumulator	2	←	X has value 2

On the other hand, suppose that the processor timings are such that accumulator 2 is loaded (step 4 above) before the store accumulator 1 instruction which precedes it (step 3). Then X has value 1 because of the sequence:

1'	load accumulator	1	←	X has value 0
2'	add 1 to accumulator	1		
3' (=step 4 above)	load accumulator	2	←	X has value 0
4' (=step 3 above)	store accumulator	1	←	X has value 1
5'	add 1 to accumulator	2		
6'	store accumulator	2	←	X has value 1

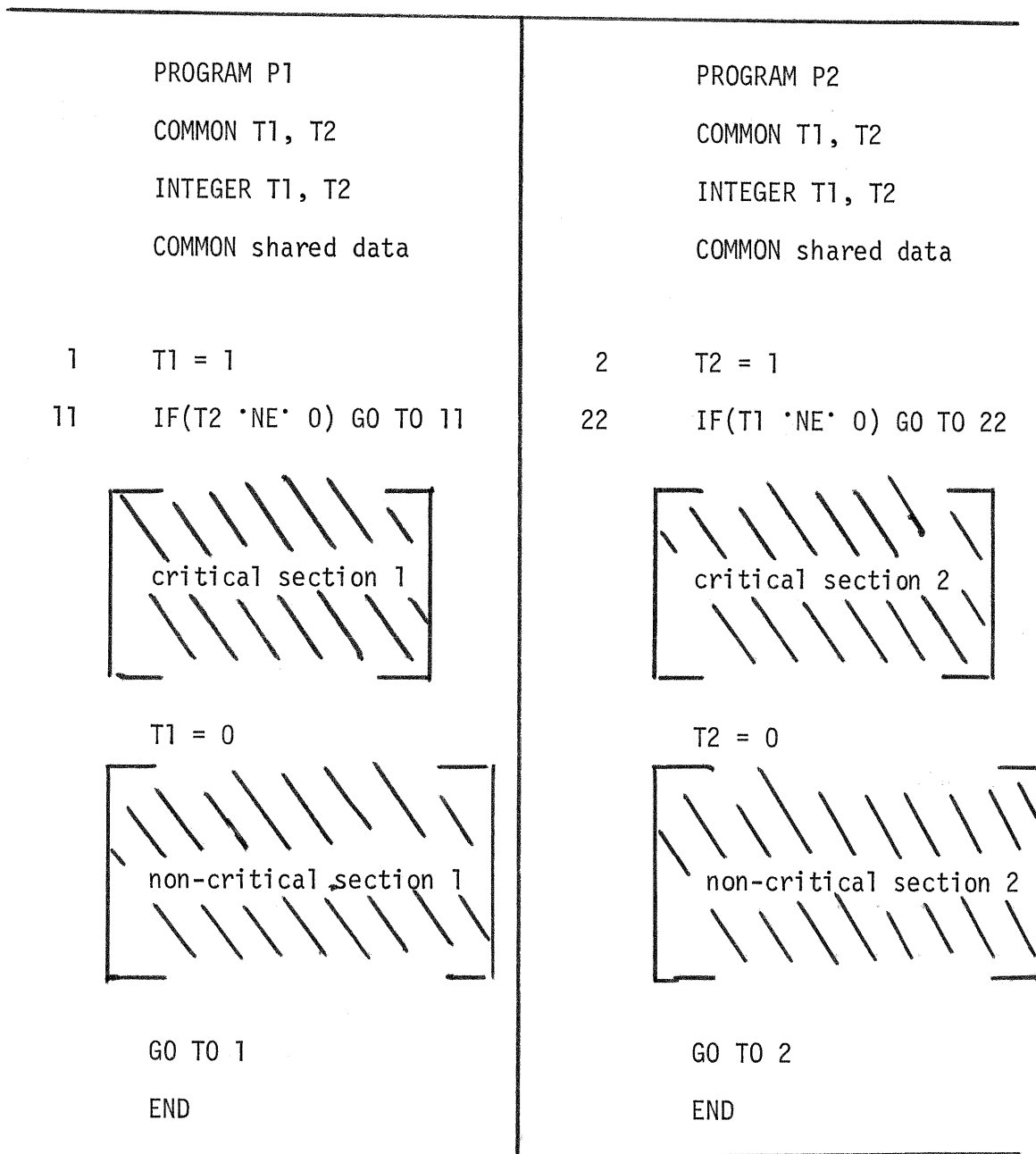
We see that a final value of 1 (incorrect) or 2 (correct) may be stored in X depending upon the sequence of events. That is, this system exhibits nondeterministic behavior [6] which is dependent upon relative times of execution of instructions by the two processors. The same type of error could occur if two jobs both access a shared data file or some other serially reusable resource. Errors of this type tend to be particularly elusive because of their time dependence. The probability of a particular time sequence of events may be low, so that the system may run for a long time before the error occurs, (e.g. Wood [3]). The system with which I worked at Bell Telephone Labs was a supposedly fail-safe system which we had thoroughly tested for several months. Soon after it was released, the system crashed due to an error similar in nature to this one. Worst of all, we had no clues to the error because the particular timing sequence causing the error was unknown and irreproducible. A way of avoiding this problem is to implement some form of mutual exclusion.

Mutual exclusion, as exhibited in the two programs of figure 1, does insure that only one processor at a time will enter its critical section so that correct count values will always be stored in X. We did not discuss the case in which steps 3 and 4 above were executed at exactly the same time. We always assume that if a variable X is simultaneously assigned two values (from two processor accumulators perhaps), this race condition will be resolved and one or the other of the values will be stored. Thus we rule out the possibility that a combination of the two results or garbage will be stored. Similarly, the effect of testing a variable and simultaneously setting the variable is that the test will indicate the results either before or after

setting the variable, but not the results with respect to some intermediate value. This criterion justifies use of the global variable T which may be simultaneously accessed in figure 1. We demand that our solutions work independent of which processor wins the race, and furthermore, we demand independence of the relative speeds of the processors. They may be the same speed, different speeds, varying speeds, or the programs may be time-sliced on a single processor in any fashion. This requirement, which is referred to in the literature [4] as speed independence, makes our solutions and our model applicable to both multiprocessor and multiprogramming systems since operations may take an arbitrary but finite amount of time to complete.

There is a problem with the program to implement mutual exclusion presented in figure 1. If a program, say P1, terminates inside of its non-critical section, then we demand that the other program must still be able to continue executing through its critical--non-critical sections loop. In figure 1, if P1 stops, it will never again set $T = 2$, so that P2 will eventually enter a nonterminating loop at statement 2 waiting for T to be set to 2. Thus we reject this solution because it does not satisfy the above requirement, which we will refer to as criterion 1, partial operability.

Figure 2

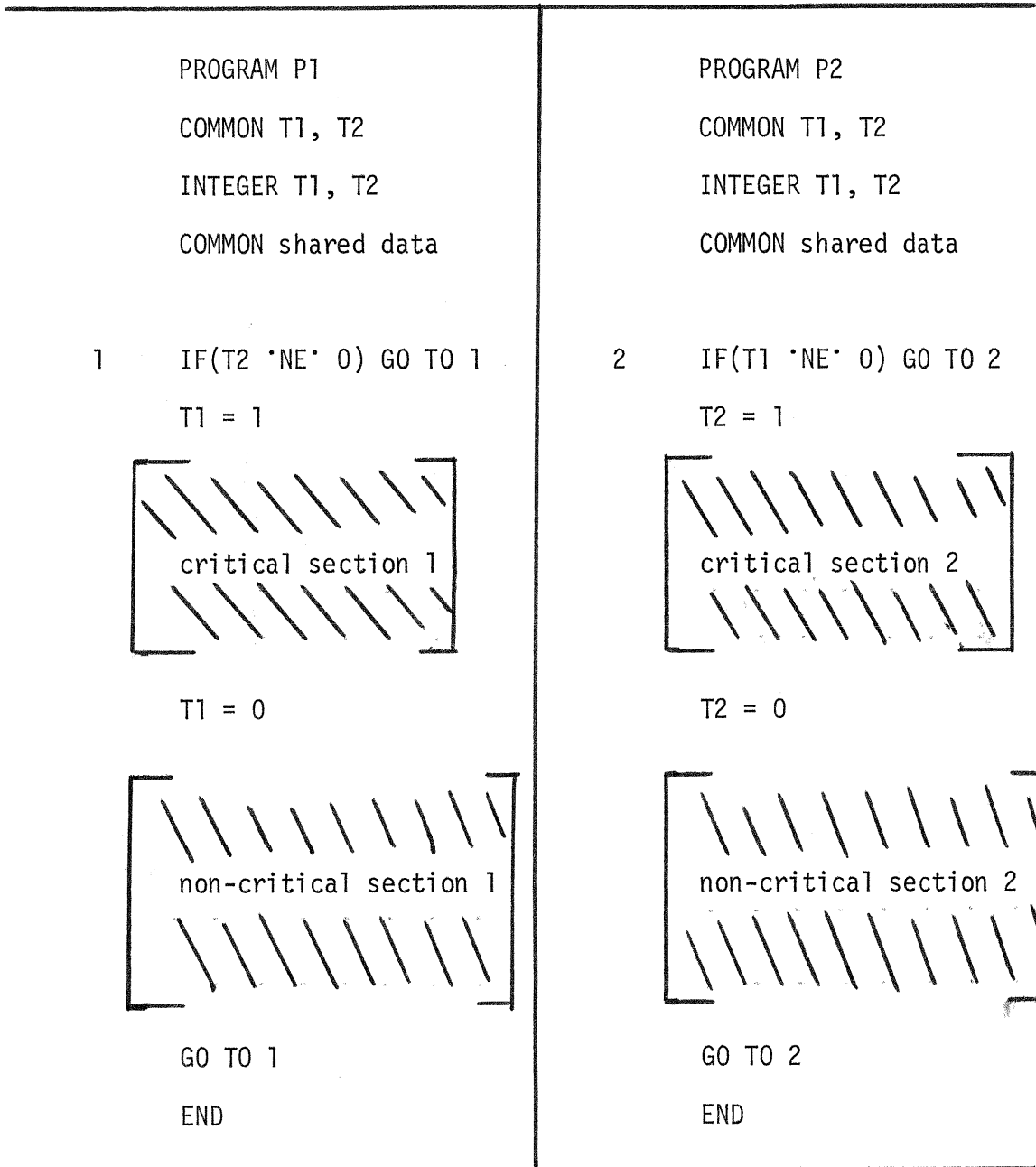


* Assume variables T1 and T2 are initially set to 0

Figure 2 offers a solution to the mutual exclusion problem which satisfies criterion 1 by introducing two variables $T1$ and $T2$ in place of the global T . $T1$ is only modified by $P1$ before and after it executes its critical section where $T1 = 1$ implies $P1$ is inside and $T1 = 0$ implies it is outside of its critical section. If $P1$ stops within its non-critical section, then $T1 = 0$. Thus $P2$, which tests $T1$ for zero at statement 22 will continue to operate correctly. The same argument holds for the case in which $P2$ stops because the programs are symmetric. The introduction of the two symmetric variables $T1$ and $T2$ has relieved one problem, but caused another: If both processes execute the same statement at (approximately) the same time within their respective programs, then when they both execute their IF tests, both will loop on this test forever. This is the condition $T1 = T2 = 1$, so statements 11 and 22 will allow neither process to enter its critical section. This behavior represents a situation of deadlock as defined by Holt [7], Coffman [8], and others. We thus reject the solution of figure 2 saying that it does not satisfy the non-deadlock criterion which we will refer to as criterion 2. Notice that it is possible to have a large, complex sequence of tests within the synchronization control section of the programs which may never allow the processes to enter their critical section, although the processes are not "stuck" on a single statement. If the programs are such that under certain speed assumptions the processes will wait forever, (even though under other assumptions one or the other will eventually enter its critical section) then we still reject this solution. This rejection is consistent with our demand for speed-independence. Justification: We want to eliminate (as much as possible) code within our operating system which will malfunction if a processor or channel is traded in

for a faster one. This problem will be dubbed the critical blocking problem, and a solution of this type will be rejected because it does not fulfill criterion 3. If, in figure 2, we put the test statements 11 and 22 before the assignment statements 1 and 2, then we avoid all critical blocking and deadlock. This solution is presented in figure 3.

Figure 3



* Assume variables T1 and T2 are initially set to 0

We (at last) seem to have found a suitable solution. Criteria 1, 2, and 3 are all satisfied. However, a closer scrutinization of this solution reveals that if statements 1 and 2 are executed simultaneously, then the statements following these will be executed and both processes will then enter their critical sections. Oh dear! We have defeated our original purpose which was to not allow both processes to enter their critical sections. We will, then, define criterion 4 as the mutual exclusion condition that two (or more) processes must not be allowed to simultaneously execute code inside of their critical sections. We must reject this third solution because it violates criterion 4. In the next section, using the formalism of L-systems [9-11], we present an abstraction of the set of programs given in figure 3 and show that criteria 1 through 4 can be mechanically verified or invalidated. This validation technique is applicable to any set of programs offered as a solution to the mutual exclusion problem. The appendix shows a correct solution to our problem.

Section 2: Application of L Systems to Mutual Exclusion

The following discussion will briefly introduce the formalism of L systems using the notation of Rozenberg [10]. It will then be shown that the question of whether a set of programs fulfill the four criteria is equivalent to certain decidable questions concerning the emptiness of particular sets of strings and sequences. Finally, it is indicated that there are a number of other related operating systems problems which are amenable to analysis via the formalism of L systems.

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

(1) Elimination of the distinction between terminal and non-terminal 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 co-operating processes. The following are a few of the definitions of Ehrenfeucht and Rozenberg [10], 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 Σ is a finite nonempty set called the alphabet of G , g is a symbol not in Σ used as an end marker, w is a word over the alphabet Σ 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 $k \in \mathbb{N}$ and $\ell \in \mathbb{N}$. (\mathbb{N} = set of natural numbers).

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

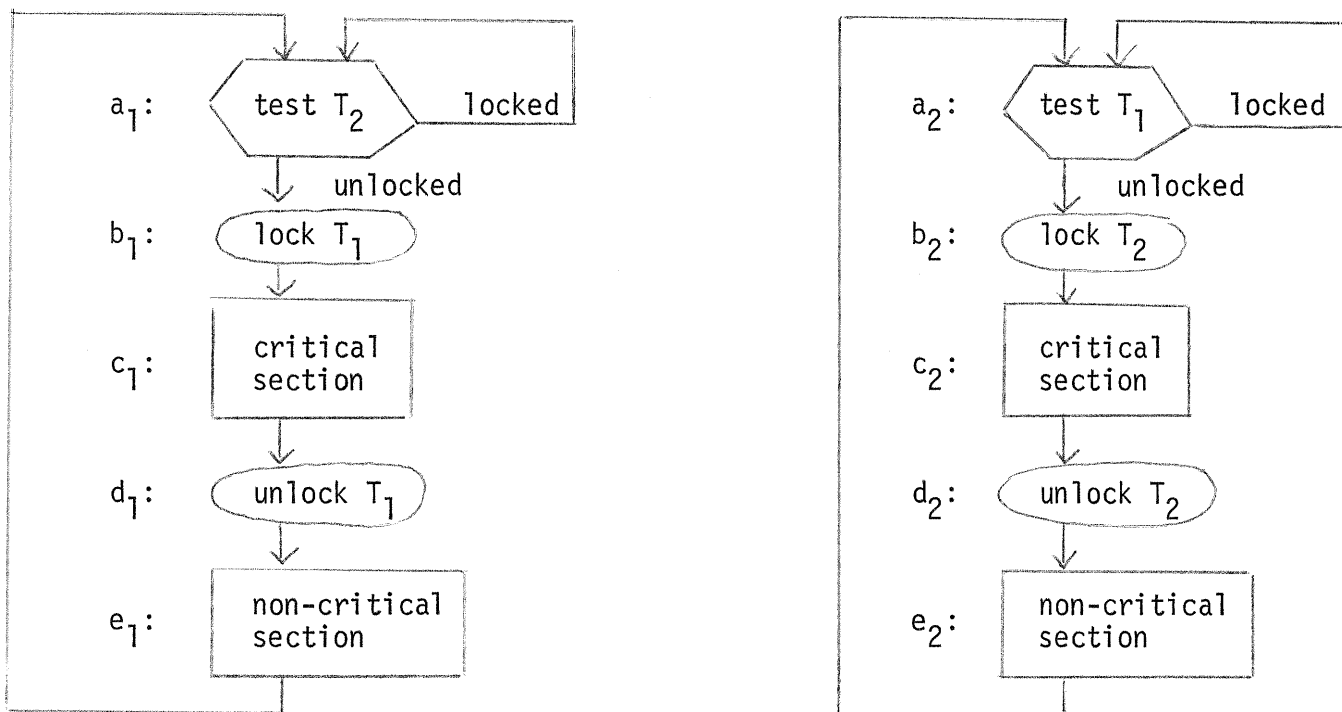
there exists a γ in Σ^* such that $\langle \alpha, a, \beta, \gamma \rangle \in P$.

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 Λ 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 as explained next.

Definition 2: Let $G = \langle \Sigma, P, g, w \rangle$ be a TIL system. Let $x = a_1 a_2 \dots a_n \in \Sigma^*$ and $y \in \Sigma^*$. We say that x directly derives y in G (written $x \xrightarrow[G]{*} y$) if $y = \gamma_1 \gamma_2 \dots \gamma_n$ for some $\gamma_1, \gamma_2, \dots, \gamma_n \in \Sigma^*$ such that there exists a table P in Σ and for every i in $\{1, 2, \dots, n\}$, P contains a production of the form $\langle \alpha_i, a_i, \beta_i \rangle \rightarrow \gamma_i$ where α_i is the last k symbols of $g^k a_1 \dots a_{i-1}$ and β_i is the first ℓ symbols of $a_{i+1} \dots a_n g^\ell$. The transitive and reflexive closure of the relation $x \xrightarrow[G]{*} y$ is denoted $x \xrightarrow[G]{*} y$, then we say that x derives y in G .

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

Returning to the mutual exclusion problem, the essence of the programs shown in figure 3 can be represented by flowcharts or graphs (as given below) where each node is given a label, a_i through e_i , corresponding to the label attached to statements in figure 3. Since we are not interested in the details of exactly what operations are performed within the critical section, this entire section is abstracted to a single node labelled c_i . Similarly all code within the non-critical section of process P_i along with the GO TO statement following it is represented by node e_i .



The state of the total system at any particular instant of time can then be captured by a string $S_1 T_1 S_2 T_2$ 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 critical region for its own exclusive use, or 0 implying that process i is not within its critical region (unlocked). The elements

of these value sets will form the alphabet of the L system model.

$\Sigma = \Sigma_1 \cup \Sigma_2 \cup \{0,1\}$. Note that by subscripting the node labels we can model asymmetric solutions involving an arbitrary number of processes although Dijkstra only considers symmetrical solutions. Elsewhere in this paper we will employ the notation $S_i(\gamma)$ to mean the value taken on by S_i within string γ . 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 process 1, by process 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 process should start along with the initial values of variables. In our example, $w = a_1 | a_2 1$.

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) $w = a_1 | a_2 1$
- (d) $P =$

	<u>Table 1</u>	<u>Table 2</u>
1	$a_1 \rightarrow a_1$	
2	$\langle \Lambda, a_1, \delta \delta 0 \rangle \rightarrow b_1$	Same as
3	$\langle \Lambda, a_1, \delta \delta 1 \rangle \rightarrow a_1$	table 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	$c_1 \rightarrow d_1$	
9	$e_1 \rightarrow e_1$	Same as
10	$e_1 \rightarrow a_1$	table 1

	<u>Table 1</u>	<u>Table 2</u>
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	$0 \rightarrow 0$	$\langle a_1 c_1 d_1 e_1, 0, \Lambda \rangle \rightarrow 0$
15	$1 \rightarrow 1$	$\langle a_1 b_1 c_1 e_1, 1, \Lambda \rangle \rightarrow 1$

The tables only give productions denoting the progress of process 1. Further productions would need to be specified to incorporate process 2. A computer program given graphs of any programs for mutual exclusion could construct the tables. It is also asserted that the program can check for correctness of a proposed solution to the mutual exclusion 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 process 1 is within flowchart box a_1 testing to see if it can enter its critical section ($S_1 = a_1$), process 2 is within its non-critical 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 state, we must select a table (let's choose table 1) and apply one production from this table to each symbol in γ . Thus if P_1 completes successful execution of its IF statement (depicted by production 2, $\langle \Lambda, a_1, \delta \delta 0 \rangle \rightarrow b_1$ and P_1 leaves its non-critical section ($e_2 \rightarrow a_2$), then the corresponding derivation is $a_1 0 e_2 0 \Rightarrow b_1 0 a_2 0$. 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 followed by any two symbols followed by a 0. 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 at one node for an arbitrary but finite number of transitions before moving to the next node.

Thus, for each symbol $\alpha \in \Sigma$, there must be a production $\alpha \rightarrow \alpha$. If in the above example, the IF statement takes a longer time to execute, then process 1 would not yet complete execution of its IF statement. This situation can be depicted by utilizing production 1 of table 1, $a_1 \rightarrow a_1$, to produce the derivation $a_1 0 e_2 0 \Rightarrow a_1 0 a_2 0$. Table 2 is provided to guarantee that the action of leaving node 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 node d_1 (see productions 11 and 12).

Given this L system G as a model of the asynchronous programs of figure 3, the question "Is it impossible for both programs to simultaneously be within their critical sections?" can be cast as the following question concerning set emptiness within the model. Condition 4: Is the set C_4 empty? $C_4 = \{\gamma \in L(G) \mid (c_1^i \in \gamma) \wedge (c_2^j \in \gamma) \text{ for some } i, j \in \mathbb{N}, i \neq j\}$. In the example just developed, this means that we consider the set of strings γ 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 $\gamma \in L(G)$). Since c_1^i means that process i is in its critical section, a string such as $\gamma = c_1^i c_2^j$ within $L(G)$ means that C_4 is non-empty. Thus, by a sequence of legal actions specified by $w \xrightarrow[G]{*} \gamma$, it is possible for both programs to enter their critical sections. 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 programs are within their critical sections. Thus criterion 4 is satisfied. In the example, there is a legitimate derivation $w = a_1 0 a_2 0 \Rightarrow b_1 0 b_2 0 \Rightarrow c_1^1 b_2 0 \Rightarrow c_1^1 c_2^1$. This L system fails to meet condition 4, so the set of programs (figure 3) must fail to meet their mutual exclusion

criterion (criterion 4). If the L system depicting the programs of figure 2 were examined it would be found that the adult language [10, section 2.4] of G is non-empty. This is precisely the necessary and sufficient condition for (total) deadlock. Thus, we can state Condition 2: Is the set A(G) empty? and the answer is 'yes' (i.e., condition 2 is satisfied) if and only if criterion 2 is satisfied with respect to the corresponding set of programs. The related criterion of no critical blocking is satisfied if and only if the following question can be answered affirmatively. Condition 3: Is the set of C_3 empty? C_3 consists of the set of infinite derivations, $w = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots$ such that

1. $(\exists j)(\exists N) (\forall n > N, c_j \notin \gamma_n)$
- and
2. $(\forall j)((S_j(\gamma_k) = \infty) \supset (\exists \ell > k \exists S_j(\gamma_\ell) \neq \infty))$

The set C_3 consists of infinite derivations because it is necessary to capture the states of the system after an arbitrarily long initial period of time during which a process may loop within its control statements before entering its critical section. Part 1 of the specification of the set C_3 $[(\exists j)(\exists N)(\forall n > N, c_j \notin \gamma_n)]$ states that condition 3 will fail if there is any derivation such that after some initial period of time (depicted by γ_0 through γ_N) the symbol $c_j \in \Sigma$ (for some j) never appears. This means that the critical section is never again entered by process j which is exactly the condition of critical blocking. Part 2 of the specification of C_3 $[(\forall j)((S_j(\gamma_k) = \infty) \supset (\exists \ell > k \exists S_j(\gamma_\ell) \neq \infty))]$ is appended to rule out the possibility that some process might take an infinitely long time to perform some operation. Recall that the productions $\infty \rightarrow \infty$ 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 process state S_j do not consist only of applications of the production

$\infty \rightarrow \infty$. This is indicated by the formal statement: if S_i takes on value ∞ in the k -th string of a derivation ($S_i(\gamma_k) = \infty$), then S_i must take on another value at some later time ($\exists \ell > k \Rightarrow S_i(\gamma_\ell) \neq \infty$). Finally, the criterion of partial operability can be examined by simply applying the previous three set emptiness questions to the following subsets of $L(G)$: For each $\gamma' \in L(G)$ containing less than n (but more than zero) symbols of the form e_j where $n = \text{number of processes}$, apply the emptiness tests to $L(G')$ where $G' = (\Sigma', P', \#, \gamma')$ where Σ' is the set of all symbols in Σ except those symbols in $\Sigma_i - e_j$ for each $e_j \in \gamma'$. P' is the restriction of P to Σ' . Thus only productions which generate valid strings of symbols over Σ' can be used. This means that $e_j \rightarrow e_j$ must always be employed. Thus, in considering blocking and deadlock, the processes which are effectively stopped should not be considered.

For any L system G formed from any finite set of mutual exclusion programs, 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 condition 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 $\gamma_0, \gamma_1, \gamma_2, \dots$ as possible members of C_3 , it is sufficient to only consider those sequences such that $\gamma_{i+1} = \gamma_i$. 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 $\gamma_i, \gamma_{i+1}, \dots, \gamma_\ell$ to γ_i for each case of $\gamma_{i-1} \neq \gamma_i = \gamma_{i+1} = \dots = \gamma_{\ell-1} = \gamma_\ell \neq \gamma_{\ell+1}$ in the original sequence. The real-time subsequence is valid because $\gamma_{i-1} \implies \gamma_i = \gamma_\ell \implies \gamma_{\ell+1}$ implies $\gamma_{i-1} \implies \gamma_i \implies \gamma_{\ell+1}$. The real-time subsequence is infinite because repetitions $\gamma_i = \gamma_{i+1} = \dots$, 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 condition 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 $\gamma \in L(G)$ which does not contain one or more $c_j \in \Sigma$. If so, then since $L(G)$ only contains a finite number of strings, there must be a (nontrivial) 'loop' in which some $\gamma_i \xrightarrow{*} \gamma_i$. This loop can be repeated indefinitely to obtain an infinite sequence excluding C_j . Finally, since $\gamma_i \in L(G)$, $w = \gamma_0 \xrightarrow{*} \gamma_i$. Thus a sequence beginning with γ_0 can be constructed. Conversely, if there is some real-time sequence in C_3 , then some $\gamma_i \in L(G)$ must be repeated in this sequence within $|L(G)| + 1$ derivation steps and there must be some c_j 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 condition 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

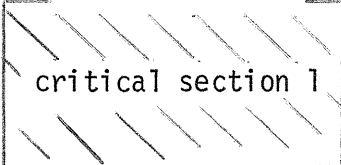
systems rather than other analytic models (such as graphs and hyper-graphs [13], suggested by Lee Osterweil). Attempts at formal proof of parallel programs have, thus far, used inductive techniques (specifying all possibilities, or assertions about programs, e.g. [14, 15, 16]), but L systems hold the promise of applying deductive reasoning to classes of programs to obtain results without enumeration of all possibilities. As Rozenberg [9] said, "undoubtedly, one of the most important [topics] in the theory of L systems is the analysis of local behavior to predict global properties". Another area deserving further study is concerned with investigation of efficient techniques for implementing a proving program for criteria 1 through 4 or extensions thereof. (This might be an appropriate thesis area for some student). It is a pleasure to terminate this paper by acknowledging Gregor Rozenberg and Andrezj Ehrenfeucht for the fascinating lectures (and aside discussions) which introduced the wonders of L systems.

APPENDIX - A CORRECT SOLUTION TO THE MUTUAL EXCLUSION PROBLEM

```

PROGRAM P1
COMMON T, T1, T2, shared data
INTEGER T, T1, T2
1   T1 = 1
11  IF (T2 .EQ. 0) GO TO 1111
    IF (T .EQ. 1) GO TO 11
    T1 = 0
111 IF (T .EQ. 2) GO TO 111
    GO TO 1

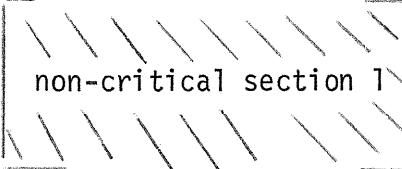
```

1111 

```

T = 2
T1 = 0

```



```

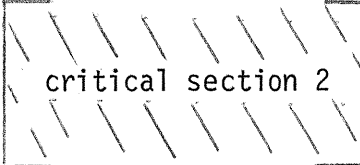
GO TO 1
END

```

```

PROGRAM P2
COMMON T, T1, T2, shared data
INTEGER T, T1, T2
2   T2 = 1
22  IF (T1 .EQ. 0) GO TO 2222
    IF (T .EQ. 2) GO TO 22
    T2 = 0
222 IF (T .EQ. 1) GO TO 222
    GO TO 2

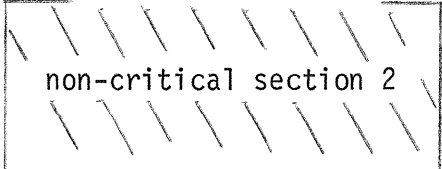
```

2222 

```

T = 1
T2 = 0

```



```

GO TO 2
END

```

* Assume variables T, T1, and T2 are initially set to 0

REFERENCES

1. Brinch-Hansen, P. , Operating Systems Principles, Prentice-Hall, New Jersey, 1973.
2. Randall, B., and Naur, P. (Ed.) Software Engineering, NATO Science Affairs Div., Brussels, 1968.
3. Wood, D. "An Example in Synchronization of Cooperating Processes: Theory and Practice" Operating Systems Review 7,3.
4. Dijkstra, E. W., "Solution of a Problem in Concurrent Programming" CACM 8,9.
5. Dijkstra, E. W., "Cooperating Sequential Processes" in Programming Languages, F. Genuys (Ed.), Academic Press, N.Y., 1968.
6. Denning, P. and Coffman, E., Operating Systems Theory, Prentice-Hall, N.J., 1973.
7. Holt, R. C., "On Deadlock in Computer Systems", Ph.D. thesis, Cornell University, Ithaca, New York, 1971.
8. Coffman, E., Elphick, M., and Shoshani, A., "System Deadlocks", Computing Surveys 3,2.
9. Rozenberg, G., and Ehrenfeucht, A., Lectures on L Systems, presented at summer 1974 colloquia.
10. Rozenberg, G., L Systems, Springer Verlag, N.Y., 1974.
11. Lindenmayer, A. and Rozenberg, G., "Developmental Systems and Languages" in 4th ACM STOC Conference, 1972.
12. Rozenberg, G., "TOL Systems and Languages", Information and Control, v. 23.
13. Osterweil, L. Private communications.
14. Knuth, D. E., "Additional Comments on a Problem in Concurrent Programming Control," CACM 9,5.
15. Habermann, A. N. "Synchronization of Communicating Processes", CACM 15,3.
16. Birman, A., "On Proving Correctness of Microprograms", IBM J. R&D 18,3 (May 1974).