AN APPROACH TO SYSTEMS CORRECTNESS

Gilles Kahn
Computer Science Department
Stanford University
Stanford, California

Summary

First, the problem of proving the correctness of an operating system is defined. Then a simple model is presented. Several examples are given to show how this model allows derivation of proofs about small systems.

I. The Problem

1. Systems

The notion of algorithms is by no means the property of the users of computers. Examples of algorithms are frequent in everyday life: the instructions on a shampoo bottle, a recipe to make a sauce bearnaise. Systems, in the same sense as in operating systems, also surround us. We are familiar with:

a library,
a traffic light,
a department store,
the post office system,
the telephone network,
an elevator,
the OS/360.

Let us look at their basic similarities.

1.1 Users. The purpose of each of the systems considered as examples is to serve a family of users. There can be a finite number of users (e.g. students in the University) or an unbounded number of them (e.g. customers, cars, programs).

1.2 Simultaneity. The systems are composed of different parts or pieces of equipment that can work simultaneously, so that the behaviour of the system is (a priori) more than the action of a single unit with a single operation going on at a time. This simple idea is best explained by examples: all customers in a library are operating independently, and independently of the employees who check out the books, or bring back the books to the shelves. Cashiers, salespeople, accountants are all working at the same time in a department store. Card readers, line printers, adders, multipliers, memory banks, discs are all in operation simultaneously in a computer system, and the fact that a multiplier is busy does not necessarily prevent a memory bank from sending information to the arithmetic unit.

1.3 Service. Users, to obtain satisfaction, may require the service of more than one of the units that constitute the system. However, there may be constraints on the order in which the customer wants these units to serve him (e.g. he wants to park his car before buying anything, he needs to go to the bank before going to any other shop) and on what kind of service he expects from them (e.g. he wants to buy socks and ties of analogous color, he cannot buy goods worth more money than he has). Notice also that a customer cannot always hope to be served immediately and might have to wait in line.

2. Systems Description

There are two ways to describe a system; they will be called the analytic description and the functional description.

2.1 The Analytic Description. One can describe an elevator by presenting its design plans, or describe a traffic light by exhibiting the circuitry of its control box. The analytic method of describing any system consists in laying bare the anatomy of the entire mechanism: the parts the system is made of and the connections between these parts. Of course, there are several levels of description for each part and one may not want to describe the internal mechanism of the accounting machine in order to depict properly the organization of the billing service.

The analytic description of an operating system consists of the set of "systems programs" together with the hardware diagrams.

2.2 The Functional Description. The functional description of a system is its description from the point of view of the users. A shopping center is supposed to "serve" customers, a traffic light is supposed to "regulate" traffic, the postal system is supposed to "route" letters. The user of an operating system expects the system to execute a certain number of tasks for him in accordance with the claims of the user's manual.

The language of functional description of a system is always different from the language of analytic description. To the author's knowledge, there is no existing investigation of what a reasonable functional language for operating systems should look like.

2.3 Correctness. Intuitively, a system is correct if it does what it is supposed to do. An elevator works if, when a person in the elevator presses the button for floor x , the elevator will stop at floor x some time later. The (simplified) postal system works if any letter placed in a mailbox will arrive at its destination. How does one establish correctness? In the past, several methods have been used by systems designers to convince themselves of the soundness of their creation. The major approaches are listed below; their failure and/or impracticability motivate this paper.

- Experimental method: Build the system as you think it should be. Observe its behaviour and adapt to its malfunctions. Advantages: the method involves no deep thinking and is suited to systems involving human beings. Drawbacks: it relies on trust in the designing team, the adaptation period may be long and costly and the system is never completely debugged.

- Incremental method: Start with a schematic version, check it with the experimental method, progressively add new features. Advantages: at least something is produced, if not the final product! Drawbacks: this method is not adapted to fast developments, and it produces "monsters" (e.g. Fortran).

- Simulation: Build a reduced model of the system displaying what is supposed to be its main features. Operate this model under simulated

conditions, then use the experimental method. Advantages: the model can be built while the real system is being developed. Drawbacks: the method can be of very little use in establishing correctness since the model is a schematized version of the real system and probably eliminates all its delicate features.

We propose to adapt to the case of systems the techniques for proving correctness of programs originated by Floyd[3]. We will show that there are some systematic ways to analyze a system that eventually lead to a proof of its proper functioning.

II. The Approach

1. Analytic Description

We are going to use a particular analytic description of systems that is similar to the representation of programs through flowcharts: a graph. In this graph, edges will represent communication lines, and nodes will represent processes that may operate independently.

1.1 Communication Lines and Objects. A communication line is a one-way path along which objects travel. There are many examples from everyday life: a one-way road along which cars travel, a bowling lane down which a bowling ball can be sent, a mail box in which letters can be dropped, a corridor in which people walk in one direction, a bus in a computer that transmits words of information from one unit to another.

The objects traveling along a communication line are discrete, but they can be of an arbitrarily complex nature. Typical objects we will use are bits, integers, bytes, letters, interrupt signals. The simplest objects have no particular value but are only interesting by their presence or absence: they will be called "its".*/ They are similar to the "tokens" considered by Holt[5].

Communication lines will be represented by oriented edges. To indicate the presence of an object on a line we will draw a heavy circle. (Figure 1, Figure 2)

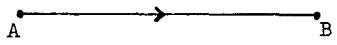


Figure 1

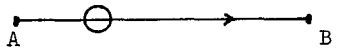


Figure 2

1.2 Multiplexers. We need a way to represent the merging of several communication lines into a single one. Objects are sent from A and B towards C (see Figure 3).and C in turn sends whatever it receives towards D . What exactly happens in C ? We make two assumptions:

(i) Time is infinitely divisible. Thus it is impossible for C to see two objects arrive on L1 and L2 exactly at the same time.

(ii) An object arriving at C on L1 or L2 is instantaneously dispatched onto L3 . Thus objects are dispatched onto L3 in their order of arrival at C .

*/ This term is due to Robin Milner.

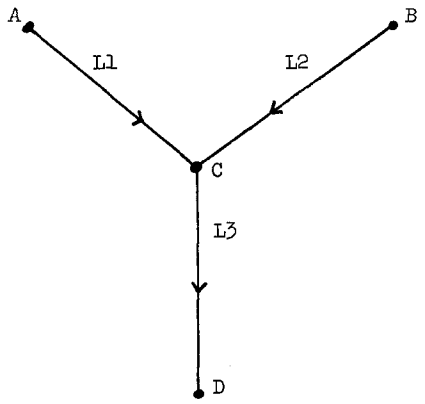


Figure 3

1.3 Queues and Capacity of the Line. A road may have several lanes, but the lines we will consider have only one lane. On a line then, objects cannot pass each other. If objects U and V are placed on the line that connects A to B at point A in this order, then U will always arrive at B before V . If objects keep pouring into the line at A and they are not removed from the line at B , a queue will form at B . We can consider it as a list (in the sense of McCarthy[6]) whose head is the first element arriving at B . The capacity of the communication line connecting A to B is the maximum length of the queue that can be formed at B . It is a physical characteristic of the line. For example, the capacity of a mailbox is the maximum number of letters that may be dropped into it, the capacity of a road linking point A to point B is the number of cars that may be bumper to bumper on this road. If a line is represented by an oriented edge, and objects on this line by circles drawn on this edge, the capacity of the line is the maximum number of circles that may be drawn on this edge. Lines we will consider will usually have capacity 1 , 2 , or infinite.

1.4 Nodes. Vertices of the graph will be of two kinds:

- Multiplexers, described above.
- Nodes, representing processes.

Nodes may represent arbitrary processes with the following characteristics:

(i) Communication between nodes is done exclusively through communication lines. The coordination of the activities of the nodes is realized by the traffic of objects along the connecting communication lines. All connections between nodes must then be explicit.

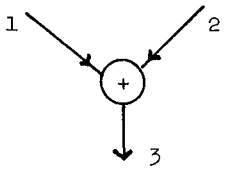
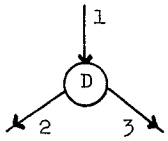
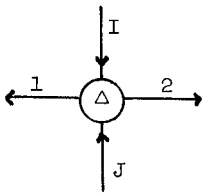
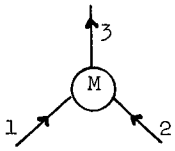
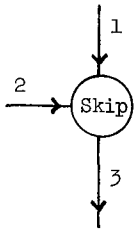
(ii) A node can be in two states: idle (waiting) or active.

(iii) If all the input queues of a node are non-empty, and if the node is idle, it will eventually make the transition from idle to active. This means that no process can "go on strike" forever.

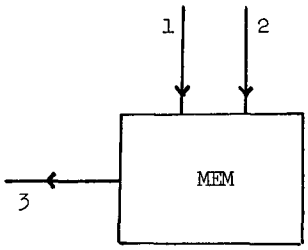
Examples of nodes are given in Figure 4.*/

*/ Note that priority scheduling can be achieved by interposing suitable nodes on communication lines.

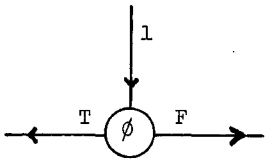
Figure 4

Node symbol		Node description
	ADDER	Waits for an input on 1 and an input on 2 , both integers; sends the sum of the inputs on 3 ; deletes the previous set of inputs. Returns to wait.
	DUPLICATOR	Waits for an input on 1 ; sends two copies of this input, one on 2 and one on 3 . Deletes the previous input and goes back to wait.
	DEMULTIPLEXER	Waits for an integer on I that can be 1 or 2 and for an 'it' on J ; sends an 'it' on the line corresponding to the input on I . Deletes previous inputs and returns to wait.
	MERGE	Waits for inputs on 1 and 2 . Sends the largest on 3 and deletes it from input line. Returns to wait.
	PULSE GENERATOR	Waits for an 'it' on 1 . Then sends continuously 'its' on 2 forever.
	SKIP	Waits for an object on 1 and a boolean on 2 . Sends on 3 the object on 1 if the input on 2 is True. Deletes inputs. Returns to wait.
	REVERSE	Waits for a list on 1 . Sends its reverse on 2 . Deletes previous input and returns to wait.

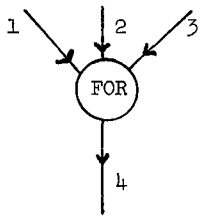
(continued on next page)



MEMORY BANK — Waits for an 'it' on 1 and an integer on 2 . Sends the quantity stored in MEM at the integer on 2 onto line 3 . Deletes inputs and returns to wait.



TEST — Waits for an object on 1 . If it verifies the predicate ∅ sends this object on T and deletes from input, else sends it on F and deletes from input. Returns to wait.



FOR — Waits for 3 integers on lines 1, 2, 3, say a , b , c . Then For i:=a step b until c send i on 4 ; returns to wait.

Figure 4

1.5 Input and Output Lines. A system communicates with the outer world via input lines and output lines. Only the extremity (resp. the origin) of an input line (resp. output line) is a vertex of the graph that represents the system.

We are now able to build systems that make sense. In Figure 5 we show a system adapted from Holt[5] to perform pipeline computations. It has two input lines I1 and I2 , two output lines O1 and O2 and 12 nodes that may be active simultaneously.

2. Functional Description

2.1 Input Condition. We feed a system by placing sequences of objects on its input lines. The input condition of the system is a condition that these sequences must verify. For example, the system in Figure 5 is designed to accept inputs on I1 and I2 that are arbitrary sequences of integers; but for the system in Figure 6, line I1 must be fed with an increasing sequence of integers, line I2 has to be presented a zero and line I3 may be fed with an arbitrary sequence of objects.

2.2 Output Condition. We expect a system to produce outputs on its output lines. The output condition of the system is a predicate relating the output sequences to the input sequences. For the system in Figure 5, let us call $\{x\}$ and $\{y\}$ the input sequences on I1 and I2 . The output condition we want is that on O1 the sequence $\{\frac{nx-my}{rx+sy}\}$ will be produced and on O2 the sequence $\{\frac{nx+my}{rx+sy}\}$.

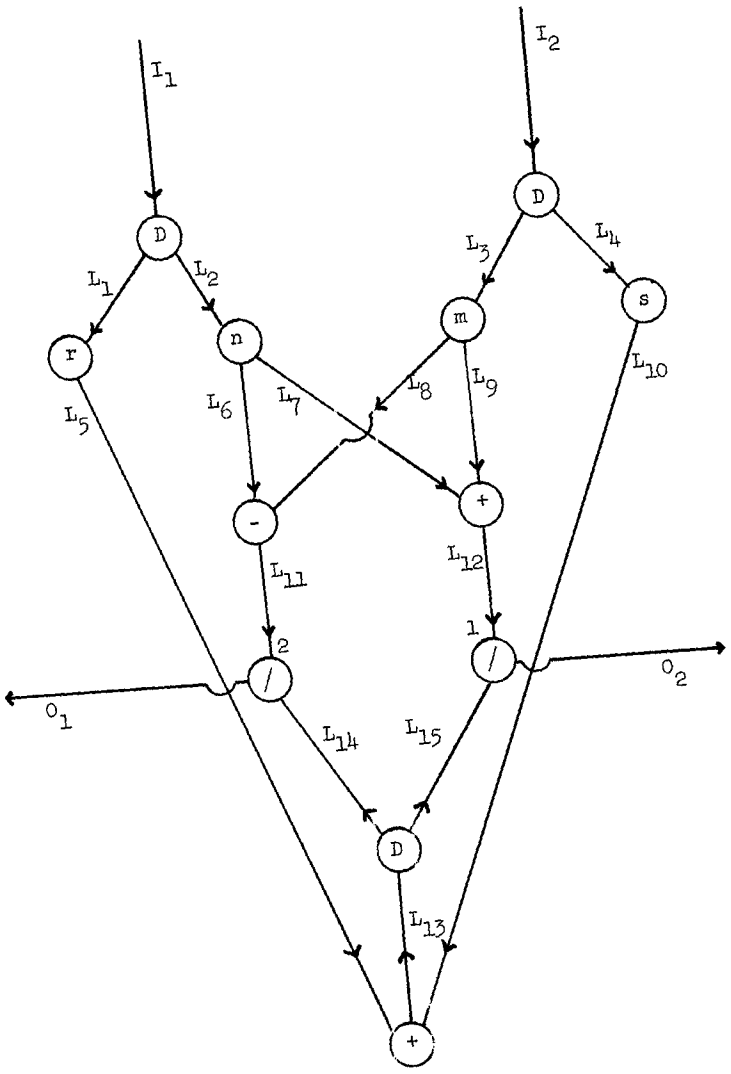
2.3 Correctness of a System. The correctness of a system is defined with respect to an input condition I and an output condition O : if a system, when fed with sequences of inputs satisfying condition I actually produces sequences of outputs satisfying condition O , it is called correct.

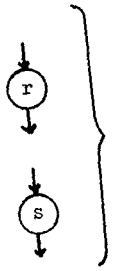
We are not prepared at this point to present a general theory of the correctness of systems. But we will show some actual proofs of correctness of small systems and exhibit some methods used in these proofs without claiming their universal validity.

3. Sample Proofs

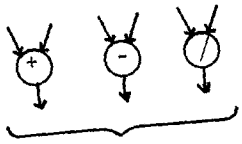
3.1 A System for Pipeline Computation. We shall analyze the system of Figure 5 which is designed to compute continuously for each pair of inputs (x,y) a pair of outputs $(\frac{nx-my}{rx+sy}, \frac{nx+my}{rx+sy})$. The system requires a sequence of integers to be fed on input lines I1 and I2 . Call these sequences $\{x\}$ and $\{y\}$. From the definition of the nodes of the system we can easily deduce the sequences of numbers that can pass through any line of the system, including the output lines O1 and O2 . Through L1 and L2 flow $\{x\}$, through L3 and L4 $\{y\}$, etc. Going down towards the output lines we see that $\{\frac{nx-my}{rx+sy}\}$ flows through O1 and $\{\frac{nx+my}{rx+sy}\}$ through O2. From the assumption that no node "goes on strike" we can deduce that the output sequences actually arrive on the output lines.

Let us assume that we do not know anything about the speed of execution of the various nodes of the system. If we may feed the system with arbitrarily long sequences of integers, then every line of the

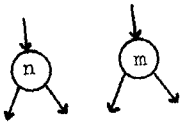




Wait for input; multiply input by r (resp. s). Delete input; return to wait.



Wait on two inputs; perform binary operation; send on output and delete inputs; return to wait.



Wait for input; multiply input by n (resp. m) . Send 2 copies on the output lines. Delete input; return to wait.

Figure 5

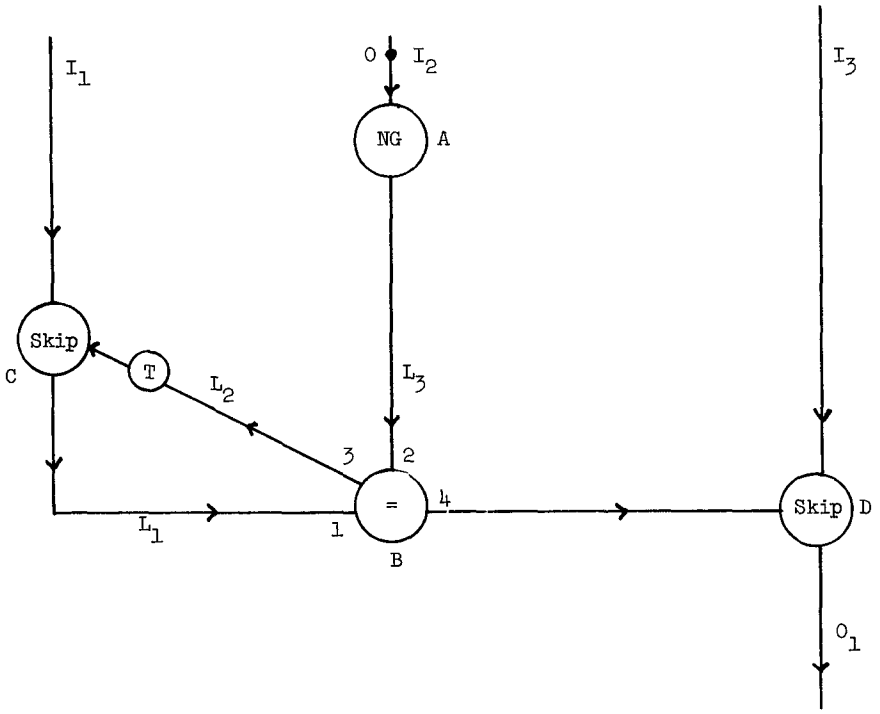
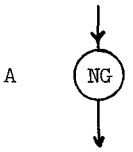
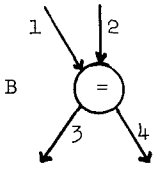


Figure 6



Waits for an input i . Then sends continuously consecutive integers starting with i .



Waits on line 1 and 2. If arguments are different it sends a False on line 4 and deletes the head of queue 2 else it sends a True on 3 and 4, deletes inputs from 1 and 2. Then it goes back to wait.

This system selects from input line I_3 the objects whose index is in I_1 and outputs them.

system must have an unbounded capacity. The proof is trivial by induction and relies on two properties of this particular system:

(i) its graph is loop-free;
(ii) the nodes of the system output infinite sequences when fed with infinite sequences.

3.2 A Merging System. Let us consider an arbitrary tree of Merge nodes (described in Figure 4), for example, the tree of Figure 7. Let us assume we feed this system by inputting to the leaves of this tree ordered sequences of positive integers followed by a 0 . We want to prove that this system performs the merging of all the input sequences, whatever their lengths are. The proof proceeds in two steps:

(i) For an individual node: By induction on the lengths of the input lines: when a Merge node is fed with decreasing sequences of positive integers followed by a zero it outputs the merge of the two input sequences with one zero missing.

(ii) For a tree of nodes: By induction on the structure of the tree.

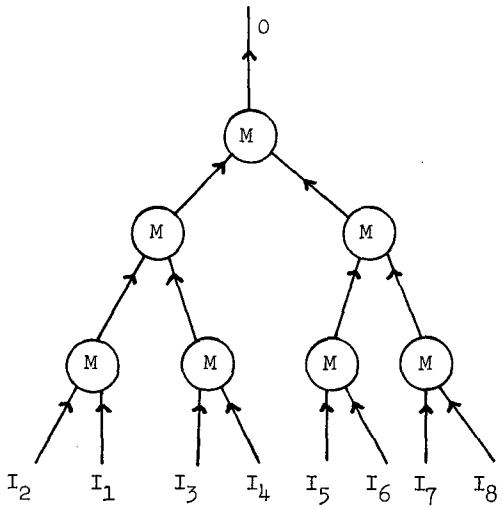


Figure 7

3.3 A First Graph With a Loop. The system on Figure 6 is designed to extract from a sequence of objects (e.g. magnetic tape records) input on I3 those objects whose position is mentioned in a list of numbers input on I1 . The system needs a sequence of increasing numbers on I1 , a zero on I2 as well as an arbitrary sequence of objects on I3 . A complete proof of its correctness is now presented.

(a) On L2 only True objects travel. Thus on L1 an initial segment of I1 passes through. On L3 , an infinite sequence of consecutive integers is placed. The sequence produced on L4 will be a sequence of True and False and a True occurs in this sequence only if its index appeared on L1 , thus was in I1 . If objects appear on O1 , they have an index in I1 .

(b) The hypothesis on I1 is now needed: By induction we can prove that all the elements of I1 will cause the creation of a True on line L4 , thus the outputs on O1 will be all the objects from I2 with index on I1.

(c) What happens if the condition on I1 is abandoned?

Either -- The first integer on I1 is negative. False objects flush through L4 , and no output is ever going to be produced.

or -- There is a first number on I1 that is smaller than its predecessor. From then on, only False objects will appear on L4 , causing the output line to dry up.

Remark: In the subgraph containing nodes B and C , there is never more than one object. Thus line L1 and L2 need only to have capacity 1 , whatever the speeds of nodes B and C are. Furthermore, it is clear that B and C cannot be active at the same time. We will generalize the remark in the next paragraph.

3.4 Mutual Exclusion.

(a) It is very simple to prevent a node from sending more objects on a line than its capacity allows. Before sending an output, the node has to wait until it receives a clearance (see Figure 8). A clearance is issued by the next node when it removes an object from the line. At the beginning, we just initialize the line with K clearances, if K is the capacity of the line.

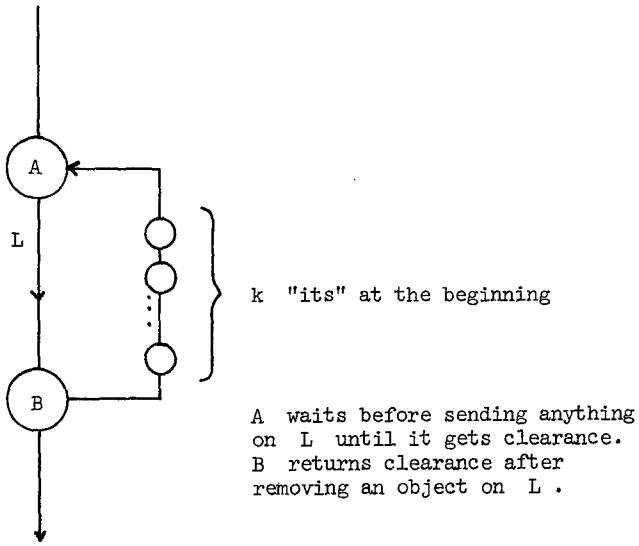


Figure 8

(b) If two nodes A and B send objects to a multiplexer C that dispatches objects towards D (see Figure 9), and the line from C to D has a capacity K , we have to provide a mechanism to prevent A and B from sending more than K objects towards C . We will issue K clearances and make each one available to the first of A or B that requests one. Clearances will be returned to the system at D when appropriate. We are led naturally to the system of Figure 10.

(c) We define a semaphore as a system with one demultiplexer and two multiplexers organized as in Figure 11. In (b) we have essentially given a constructive justification of this notion. The reader

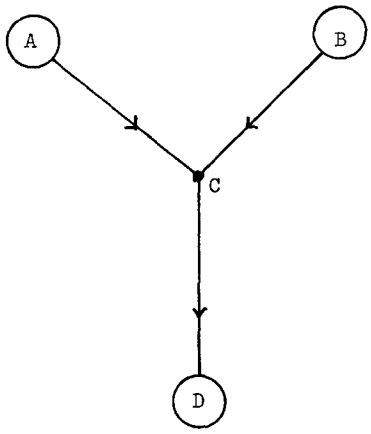


Figure 9

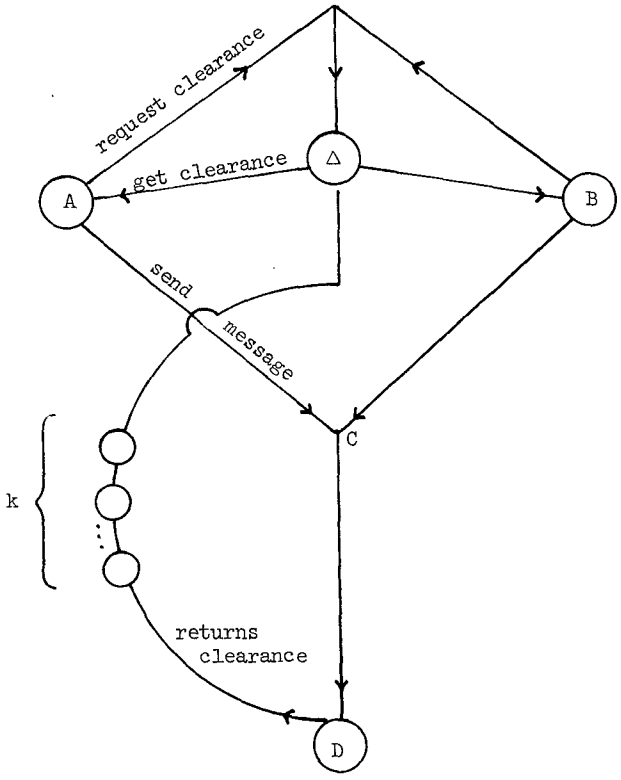


Figure 10

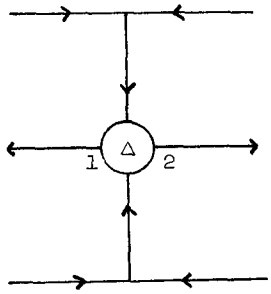


Figure 11

can convince himself that our notion of semaphore is identical to the notion of general semaphore of Dijkstra[2].

(d) In Figure 12, we have modeled the mutual exclusion problem, a generalization of the problem solved in (b). We can prove that processes T1 and T2 are mutually excluded (i.e., not active simultaneously) if an 'it' is placed on line L , since the system enclosed in dotted lines always contains one and only one object, only one of T1 and T2 can be active at any given time.

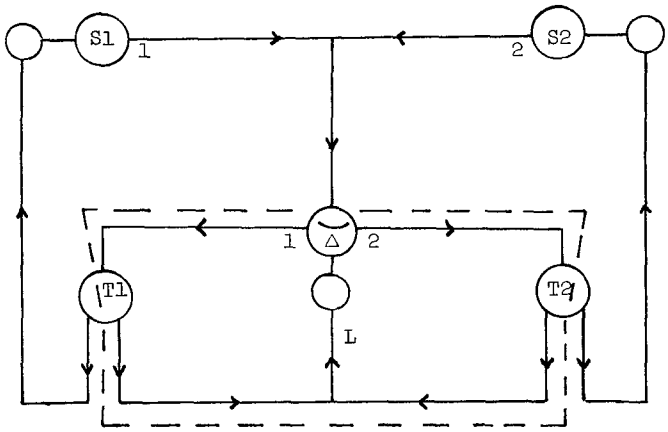


Figure 12

(e) Mutual exclusion is clearly a property that may be desired of a system and is in fact part of the intuitive idea of how a correct system must operate. However, we have defined correctness of a system as a property concerning only the output lines and not the internal nodes. To reconcile these two points of view, we simply consider that nodes communicate with the outer world only through communication lines, and that if the outer world is interested in their state, it must receive information on them via communication lines. Then a property of mutual exclusion becomes a property of certain output lines of a system.

(f) We can now suggest an approach to the most general exclusion problem: if we want to show that in a set of N processes, only n are active at anytime,

we look for an information cycle in which we could prove that there can not be more than n objects traveling, on which these N processes may be waiting.*/

III. Conclusion

In this paper, we have tried to capture the notion of discrete system, with the intent of systematizing the checking of operating systems. We have explained what a formal notion of correctness for operating systems can be. This notion subsumes the notion of correctness for a sequential program in a fairly natural manner. Simple cases have been systematically analyzed. Deadlock questions, and mutual exclusion problems have been shown tractable in a very formal way and the structure of Dijkstra's semaphores has been completely formalized.

However, we have limited our study to systems comprising a finite number of processes. We have good hope that the same approach will be fruitful in the case where the number of units working simultaneously is unbounded.

Acknowledgment

My friends Denis Seror and Mark Smith helped me considerably to formulate the ideas of this paper. I thank them wholeheartedly for their merciless criticisms.

References

[1] Adams, D. "A computation model with data flow sequencing." Ph.D. dissertation. Stanford University, Computer Science Dept. December 1968.

[2] Dijkstra, E. W. "Cooperating sequential processes." EWD123, Math. Dept., Technological University, Eindhoven, The Netherlands, 1965.

[3] Floyd, R. W. "Assigning meaning to programs." Proceedings of Symposia in Applied Mathematics. American Mathematical Society, Vol. 19, (1967), 19-32.

[4] Holt, A. W. "Final report of information system theory project." Applied Data Research, Inc., Princeton, N. J., 1968.

[5] Holt, A. W. and Commoner, F. "Events and conditions." Applied Data Research, Inc., New York, N. Y., 1970.

[6] McCarthy, J. et al. "LISP 1.5 Programming Manual." M.I.T. Computation Center, Cambridge, Mass. (1965).

[7] Seror, D. "D.C.P.L. A distributed control programming language." Ph.D. dissertation. Computer Science Dept., University of Utah, August 1970.

*/ This method was used with success to solve Dijkstra's 5 diners problem and formally prove the validity of the solution found.