The University of Wisconsin
Computer Sciences Department
1210 West Dayton Street
Madison, Wisconsin 53706

EFFICIENT ANALYSIS OF THE PROCESS STRUCTURES OF FORMALLY DEFINED COMPLEXES OF INTERACTING DIGITAL SYSTEMS

by

Pamela Z. Smith
D. R. Fitzwater

Computer Sciences Technical Report #219

August 1974

EFFICIENT ANALYSIS OF THE PROCESS STRUCTURES OF FORMALLY DEFINED COMPLEXES OF INTERACTING DIGITAL SYSTEMS

by

Pamela Z. Smith

and

D. R. Fitzwater

ABSTRACT

Basic processes are presented as uniquely suitable structural units for the initial analysis of a formally defined complex of interacting digital systems. An algorithm is given for decomposing a complex into its basic processes. The algorithm proceeds iteratively from characterizations of lesser detail to those of greater, so that analysis of any part of the system can be curtailed when the level of detail reached is sufficient. Furthermore, a modest restriction on the forms of antecedents and consequents in the systems will allow many of the operations to be performed by very efficient string manipulation algorithms. An example of this system analysis technique is worked out, and it is argued that even large designs could be studied using it.

I. INTRODUCTION

The technique presented in this report is proposed as a first step in the analysis of a formally defined complex of interacting digital systems, in the formal definition universe described in [1] and extended in [2]. It is based on the analytic tools, and the definition of a process, found in [3]. Its purpose is to provide the designer with an efficient algorithm for discovering the overall process structure implicit in his system design.

We feel that the value of such a technique is obvious. Process structure is widely recognized to be the key to understanding computations, and so a general look at a design's process structure must be an appropriate beginning to any effort toward understanding or verifying its parts. The fact that we have an efficient algorithm for doing so means that computers can be used to analyze the largest designs: this is why we can claim to have a practical design tool.

The goal of the algorithm is to find a partition of a system complex into basic processes, the properties of which are discussed in Section II. Because it is formulated as a top-down iterative procedure, analysis of any part of the system can be curtailed when it is detailed enough; the designer gets only as much information as he can afford, and is willing to look at. This is intrinsically more efficient and flexible than the usual bottom-up approach, as will be shown in Section III.

Top-down or not, system analysis requires extensive computation. Certain operations on regular languages are especially prominent in our technique, and so it should mean significant savings in computation time if they can be optimized. We will show that when certain simple rules are observed in the construction of antecedents and consequents, the languages involved will belong to a subclass of the regular languages called delimited pattern languages. The properties of this class, as developed in Section IV, allow many of the most common language operations to be performed by fast string manipulation algorithms.

Finally, an example of this technique is given, in hopes of showing that what is theoretically sound is also intuitively reasonable, and can contribute to understanding of computational structures.

II. BASIC PROCESSES

A finite process structure (fps) for a system is a finite directed graph whose nodes are disjoint regular languages to which process states of the system may belong. Each arc is associated with a production of the system, and indicates the possible generation of process states from states in the preceding σ . Fps's for isolated systems can be extended to graphs describing system complexes. Given a system complex and an fps describing it, a process is defined as a set of regular languages, one for each system in the complex, such that:

(1) the regular language for a system is a union of nodes in the fps for that system,

(2) all process states in a system belong to the process if and only if they belong to the regular language corresponding to that system, and

(3) no arcs cross the process boundary except interactions (multiple-component arcs with at least one origin node on either side of the process boundary).

These matters are explained in detail in [3].

Basic processes are processes according to the above definition (although a process is formally a set of regular languages, we will refer to it more often as a set of fps nodes--an equivalent notion). We make them the target of initial system analysis because the division of any fps into basic processes is a unique partition, and because basic processes are the smallest processes that are independently analyzable. The exact meaning of this will be discussed later, after we have presented the unique partitioning of an fps, which begins with its modules.

Definition:

A module is a maximal set of nodes in an fps such that each node in the set is connected to every other node in the set by a path composed completely of l-arcs, where l-arcs are interpreted as non-directional connectors.

A module can be found simply by starting with one node in an fps, adding every node connected to that one by a l-arc, then every node connected to those nodes by l-arcs, etc., until no new nodes can be added. Figure 1 shows an fps divided into modules.

Modules have three properties of interest to us: (1) any module is contained within one system, (2) a module is contained within any process it overlaps, and (3) the division of any fps into modules is a unique partition. Property (1) holds because only (n+m)-arcs connect nodes in different systems. Property (2) holds because no process boundary could be drawn dividing a module into two parts: the two parts of the module must be connected by at least one l-arc, but a l-arc cannot cross a process boundary. Property (3) is implied by the fact that we can view the nodes of an fps as a set of objects, and non-directional connection between two nodes by l-arcs as a relation. Since this relation holds between all nodes of the same module, it is reflexive, symmetric, and transitive, and is therefore an equivalence relation. It is well known that any equivalence relation defines a unique partition of the set on which it is defined.

Modules are building-blocks for processes, because any process is a union of modules. Since the division of an fps into modules ia a unique partition, if we give an

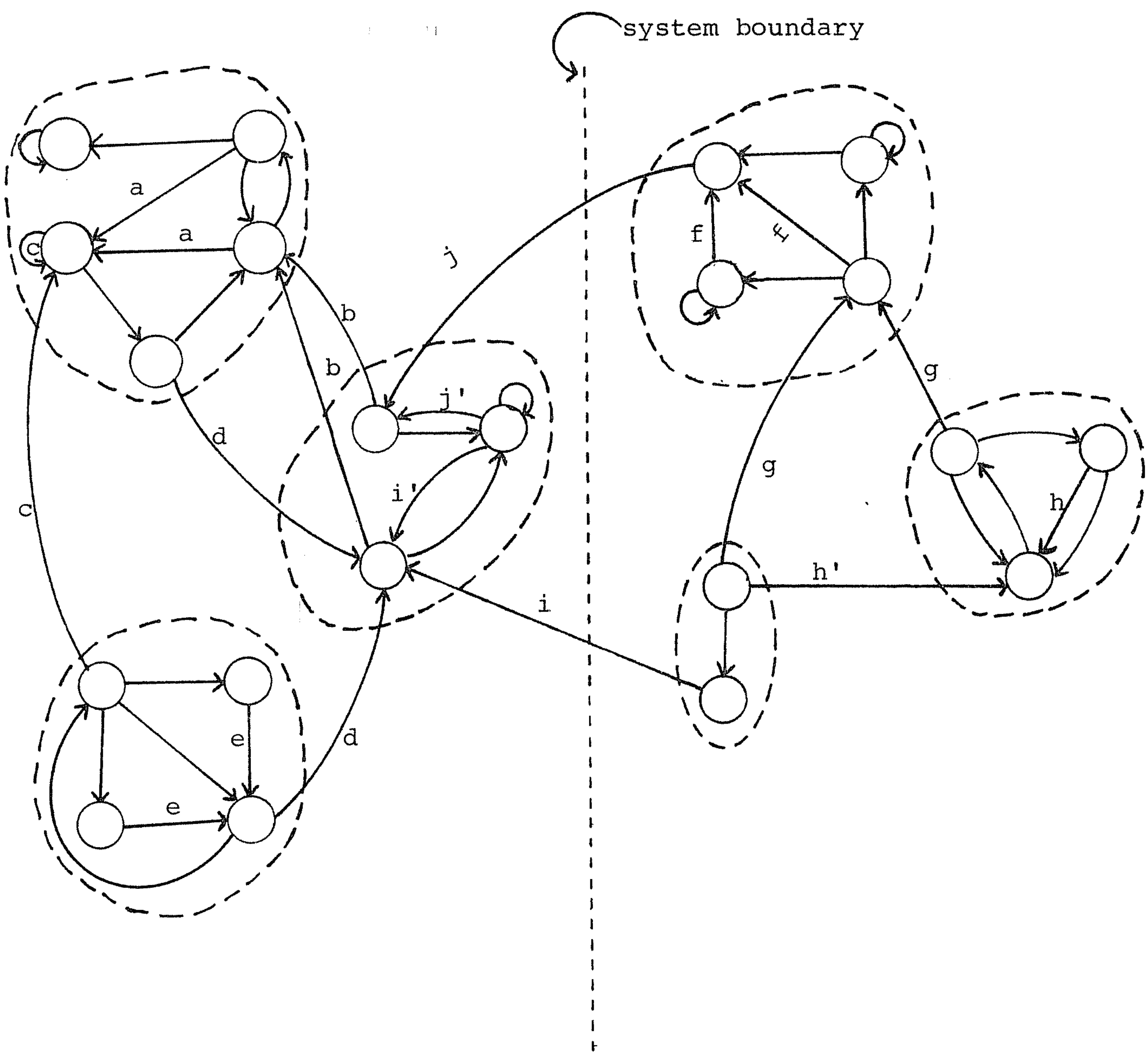


FIGURE 1

algorithm for grouping together certain modules, and say that any resultant set of one or more modules is a basic process, the implication is that the division into basic processes is a unique partition.

Definition:

Basic processes are the structural units of an fps for a system complex formed by the following algorithm:

Let the fps be partitioned into modules, and let each module belong to a distinct set. If there is an n-arc with all of its origin nodes in one set and its destination node in another set, then those two sets are united. Two sets are also united if the destination node of an (n+m)-arc is in one of them and all the origin nodes of the primed components are in the other. These rules are applied repeatedly until they can no longer be applied, and then the resultant sets are the basic processes.

Figure 2 shows the same fps as in Figure 1 with the basic processes marked. The most obvious property of basic processes is that they exist entirely within single systems. We know that modules do not contain system boundaries. Only (n+m)-arcs cross system boundaries, but their primed components cannot, so there is no means by which modules in different systems could be grouped.

Are these really processes, i.e. do only interactions cross the process boundaries? 1-arcs cannot cross them because they are all contained in modules. N-arcs can cross them only if they have origin nodes in different processes, but arcs obeying that rule are interactions by definition. (N+m)-arcs can cross them only if there are primed components with origin nodes in different processes,

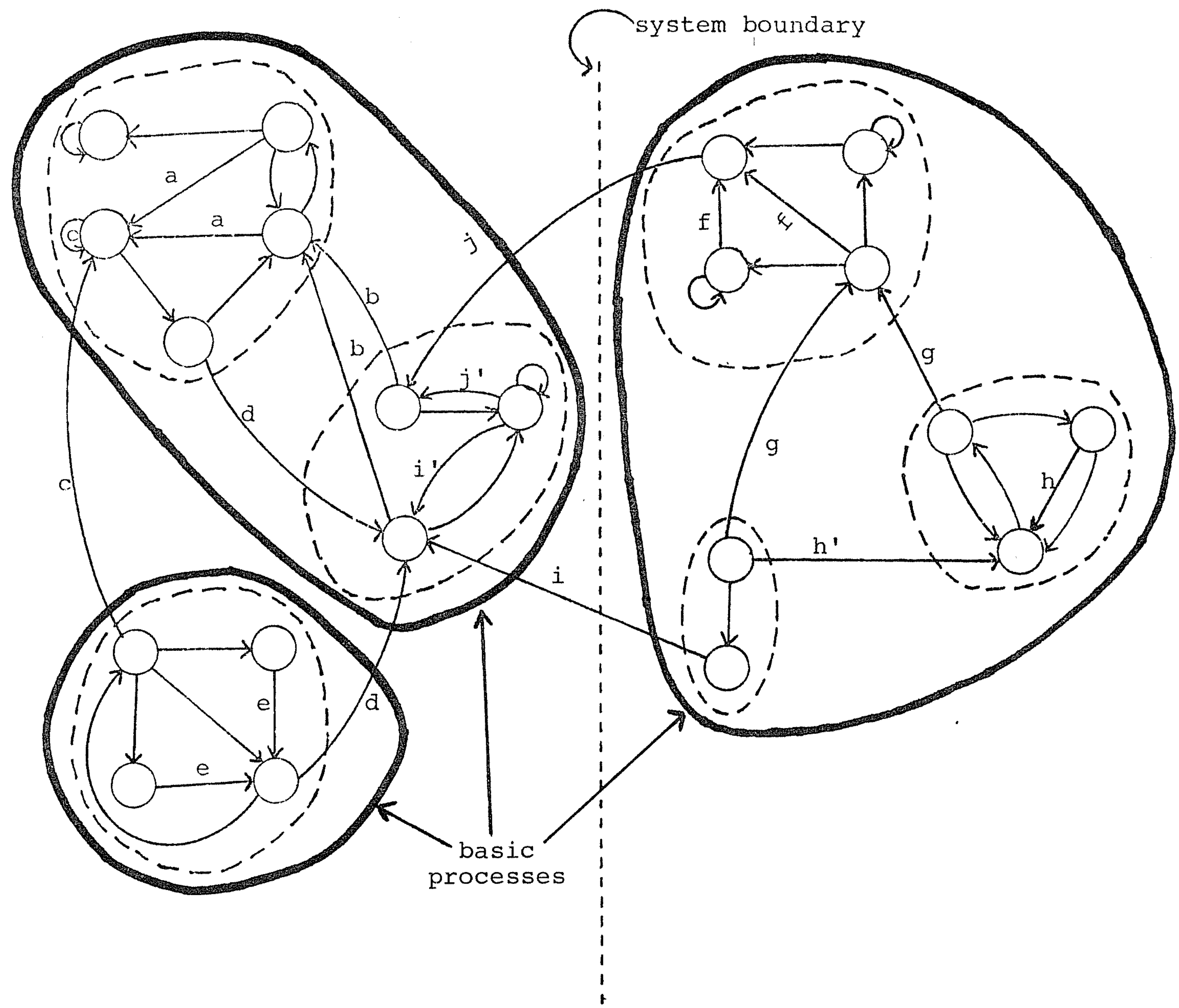


FIGURE 2

or unprimed components originating outside the destination process while all the primed components originate within it, and these are interactions also.

We have shown that basic processes comply with the process definition and that they partition an fps uniquely. They were defined so as to make the uniqueness obvious, but they are really just the smallest processes wholly within single systems whose boundaries cannot be crossed by any of the primed components of an (n+m)-arc, if all their origin nodes are in the same process. It is these properties that make basic processes independently analyzable.

By "independently analyzable" we mean that the process can be studied as a solitary synchronous entity, whose interface with the rest of the system complex can be specified as sets of inputs and outputs. More formally, we will show that all the state information deducible from the fps of an independently analyzable process can be expressed in the form of a finite state machine. It is obvious why basic processes have to exist completely within one system--asynchronous interactions do not yield much to analysis, nor do they fit into finite state machines.

All the origin nodes of primed components in an (n+m)-arc interact via some other m-arc to produce a successor process state which then acts as a channel name for the acceptance of a message, and disappears. If all these origin nodes are in the same process, then their destination node (i.e. the channel name node) must also be in that process. Consider what happens when other origin nodes (of unprimed components) of the (n+m)-arc, and its destination node, lie outside the process containing the origin nodes of the primed components: because of the interaction of process states, some or all of which are not in the process, the channel name will disappear from the process! This is a side effect which would keep that process from being independently analyzable, and so we do not allow it to occur between basic processes.

Figures 3a and 3b are illustrations of this situation. In Figure 3a, node C has successor D, and language D contains a channel name for the acceptance of a message in B . Whenever such a message is generated on the same system step in which a state in language C generates a suitable channel name, the channel name will disappear, for no reason which is detectable from the right-hand module alone. The arc labeled (a) is a legitimate interaction, and so a process boundary could be drawn where the module boundary is, but a basic process boundary could not be drawn there. Figure 3b is more complex, but the situation is much the same. The fate of a process state in language J cannot be determined without knowledge of process states in language E, so the module on the right is not independently analyzable.

A basic process does interact with its environment through its input and output arcs.

Definition:

An input arc of a group of fps nodes is an arc whose destination node is within the group, but having at least one origin node outside the group.

Definition:

An output arc of a group of fps nodes is an arc whose destination node is outside the group, but having at least one origin node within the group.

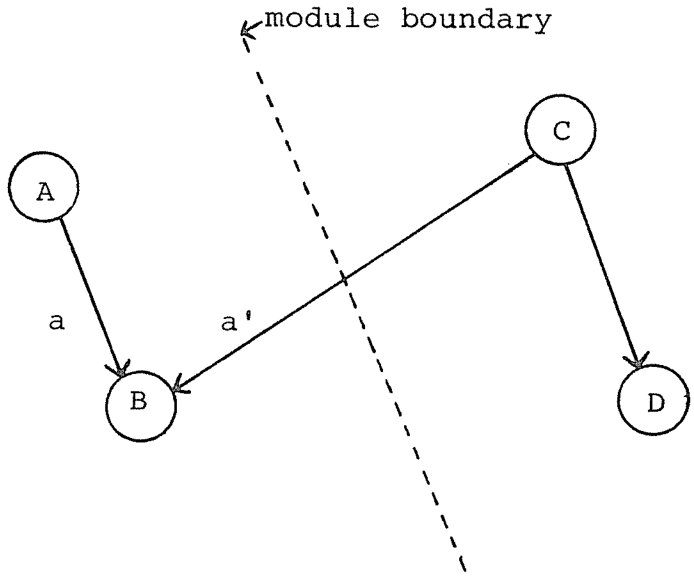


FIGURE 3a

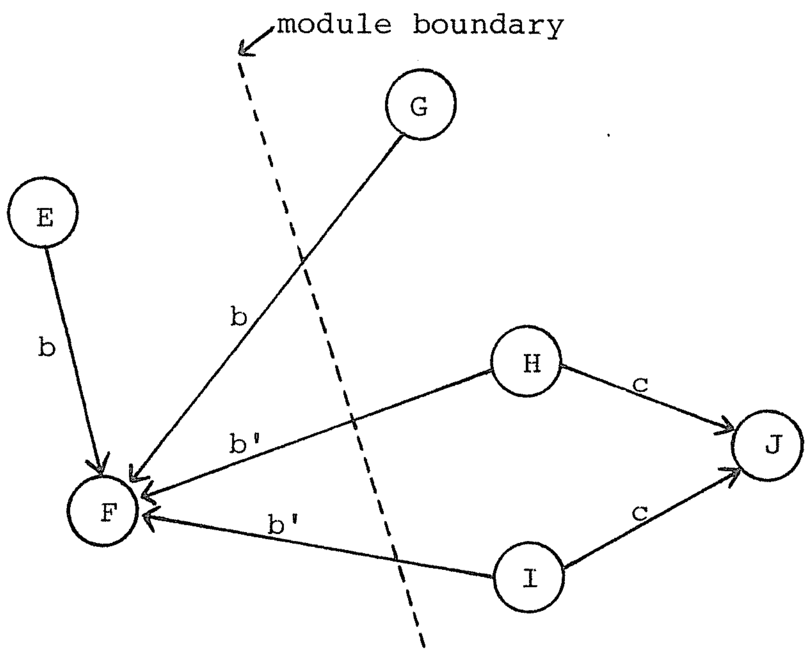


FIGURE 3b

We will show later how these are related to the inputs and outputs of the finite state machine representing a basic process. We bring them up now to show why the situation in Figure 4 does not preclude drawing a basic process boundary.

In describing the action of the right-hand process, we would say that E is an input node, and that there is input on arc (a) when there is a process state in node language A. The actual appearance of a state in language E depends on the states in nodes B and C, but if a channel name from D disappears, it will happen as an effect of conditions known in the basic process on the right: that there are states in languages B and C, and there is input on arc (a) .

This is not at all the same as if we had said, in reference to Figure 3a, that there is output from C on (a'), because the effect on states in D still depends on the presence of states in A, and that information does not belong to the module on the right.

Another situation which is handled suitably by the definition is that in which only one of several primed components crosses a basic process boundary, as in Figure 5. The possible disappearance of a process state belonging to node E can be predicted from the status of node D.

We have shown that basic processes are synchronous and do not suffer side effects; now we will show that their state sequences can be interpreted as the states of a finite state machine (under the abstraction from process states to regular languages, of course). We feel that this is a powerful result concerning the possibility of factoring system analysis, especially because composition and other manipulations of finite state machines are so well understood. The concept is not yet practical because of the combinatorics involved in constructing these machines, but there is always room for design constraints and optimizations--we should not assume that it will remain out of reach forever.

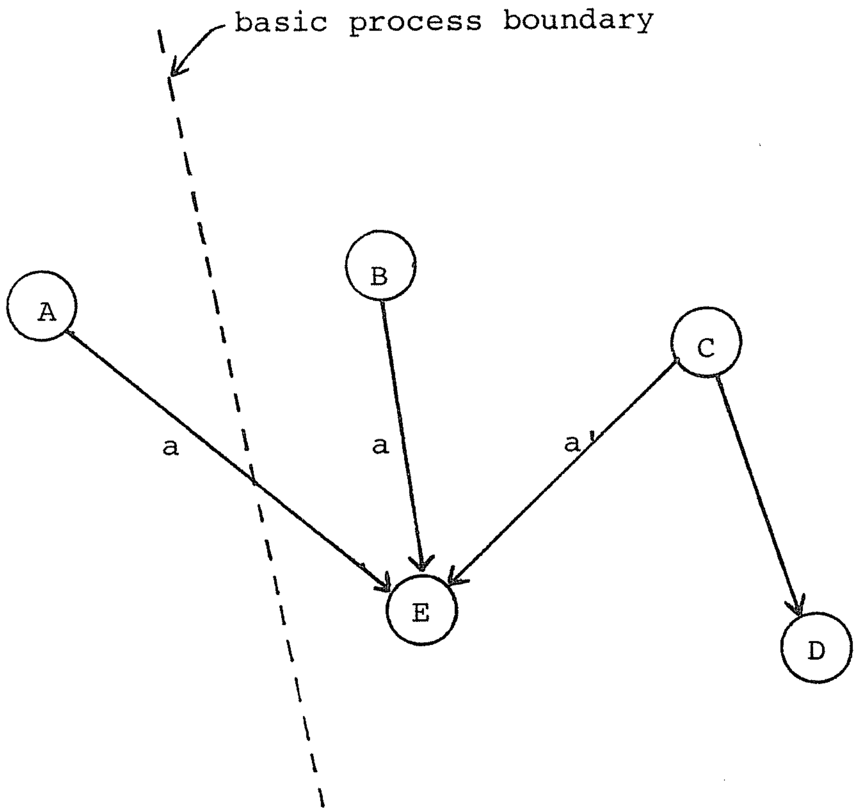


FIGURE 4

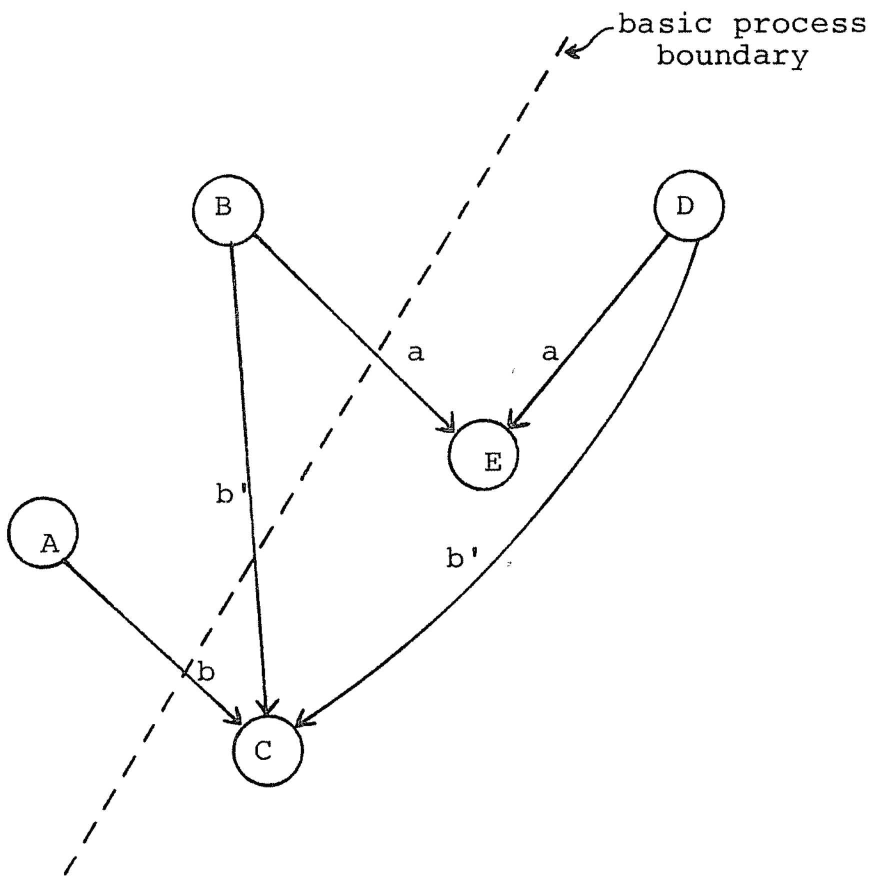


FIGURE 5

To define an fsm representing a basic process we need an fps for the process, including the interactions crossing the process boundary. For input interactions all we need to know is the names of the arcs and their destination nodes; for outputs we need to know the names of the output arcs and their sets of origin nodes within the process. Let the fps in Figure 6 serve as our example: the basic process has input to node A on arc (c), and outputs from C and D on arc (b), and from C on arc (d).

If $S = \{A,B,C,D\}$ is the set of all nodes of the fps in the process, the states of the fsm are the members of $\mathcal{P}(S)$ (the power set of S). A state $\langle A,C \rangle$ for instance, is interpreted as meaning that there is one or more process state in σ contained in node A and also in C, and none in nodes B or D. The initial state of the process machine is the set of all nodes whose intersection with the initial σ is non-empty. In our example the initial state is $\langle D \rangle$.

The transition function describes how the state of the process can change during one system step, as defined in [1]. It is calculated straightforwardly from the information in the fps. For instance, if the fps shows a 2-arc with origin nodes A and B and destination C, then any state containing the nodes A and B may have successor nodes containing C. We say "may" because, as long as the arcs are not known to be deterministic, we must assume that they are not; the result is a different successor state for each possibility and a highly non-deterministic machine. The effect on the fsm would be the same if the 2-arc were a (1+1)-arc instead.

If I is the set of input arcs to the process, then $\mathcal{P}(I)$ is the set of inputs to the fsm. The meaning of an input is this: for every arc in the subset, all the external conditions are satisfied so that the arc could represent a

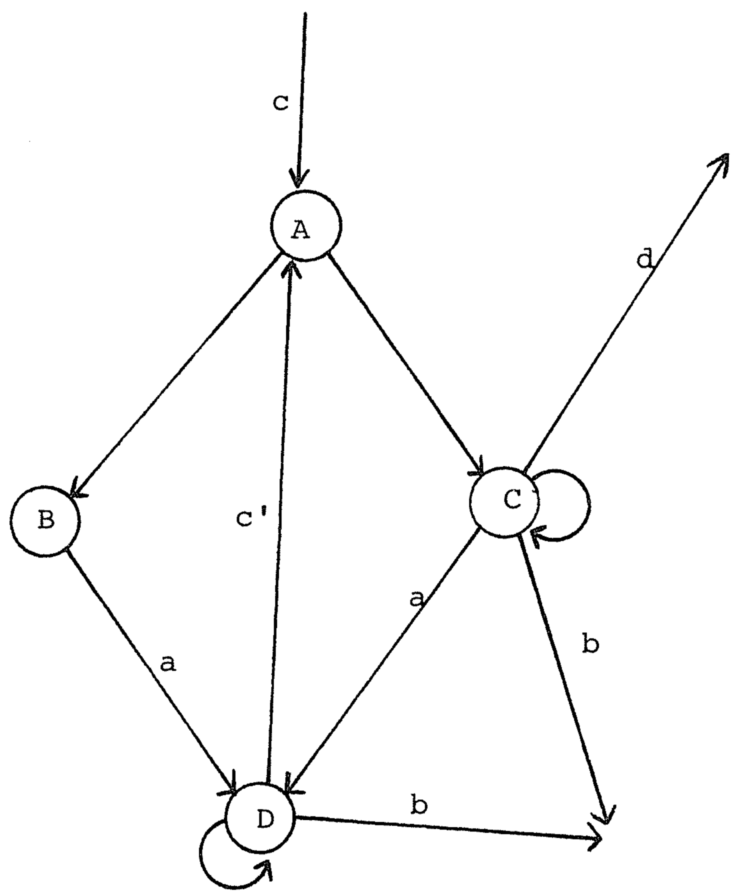


FIGURE 6

transformation which will actually take place in this system step--and for every arc not in the subset, the external conditions are not satisfied. What do we mean by external conditions? For each intra-system component of the interaction, σ must contain a process state belonging to the language of its origin node. Each inter-system component is associated with the sending of a message, and that message must be in σ'' of the system whose basic process we are modeling before it is seized at the end of this step.

If O is the set of output arcs to the process, then $P(O)$ is the set of outputs of the fsm. The output associated with each state is the set of output arcs all of whose origin nodes in this process are present in that state.

Figure 7 gives a tabular description of the fsm for the process in Figure 6. The non-determinism can get out of hand quickly! The machine can be reduced to ten or fewer states, but the breadth of non-determinism still makes analysis unrewarding.

The best way of cutting these machines down to a useful size is by making use of any obtainable information about deterministic arcs. Figure 8 shows the fsm derived by assuming that all the arcs in the fps are deterministic, an obvious improvement. The only non-determinism remaining arises because it is not known, when the machine is in a state containing D, whether σ contains one or more than one process state belonging to that language. This kind of information could also be incorporated into any analysis algorithm where its cost seemed justified.

STATE	OUTPUT	TRANSITIONS ON ϕ	TRANSITIONS ON (C)
ϕ	ϕ	ϕ	ϕ
<A>	ϕ	ϕ,<B>,<C>,<B,C>	ϕ,<B>,<C>,<B,C>
<B>	ϕ	ϕ	ϕ
<C>	<(d)>	ϕ,<C>	ϕ,<C>
<D>	ϕ	ϕ,<D>	ϕ,<A>,<D>,<A,D>
<A,B>	ϕ	ϕ,<B>,<C>,<B,C>	ϕ,<B>,<C>,<B,C>
<A,C>	<(d)>	ϕ,<B>,<C>,<B,C>	ϕ,<B>,<C>,<B,C>
<A,D>	ϕ	ϕ,<B>,<C>,<B,C>, <D>,<B,D>,<C,D>, <B,C,D>	ϕ,<B>,<C>,<B,C>, <D>,<B,D>,<C,D>, <B,C,D>,<A>,<A,B>, <A,C>,<A,B,C>,<A,D>, <A,B,D>,<A,C,D>, <A,B,C,D>
<B,C>	<(d)>	ϕ,<C>,<D>,<C,D>	ϕ,<C>,<D>,<C,D>
<B,D>	ϕ	ϕ,<D>	ϕ,<A>,<D>,<A,D>
<C,D>	<(b),(d)>	ϕ,<C>,<D>,<C,D>	ϕ,<A>,<D>,<A,D>,<C>, <A,C>,<C,D>,<A,C,D>
<A,B,C>	<(d)>	same as <A,D>	ϕ,<B>,<C>,<B,C>,<D>, <B,D>,<C,D>,<B,C,D>
<A,B,D>	ϕ	same as <A,D>	same as <A,D>
<A,C,D>	<(b),(d)>	same as <A,D>	same as <A,D>
<B,C,D>	<(b),(d)>	ϕ,<C>,<D>,<C,D>	same as <C,D>
<A,B,C,D>	<(b),(d)>	same as <A,B,C>	same as <A,D>

FIGURE 7

STATE	OUTPUT	TRANSITIONS ON ϕ	TRANSITIONS ON (C)
<A>	ϕ	<B,C>	<B,C>
<D>	ϕ	<D>	<A,D>,<A>
<A,C>	<(d)>	<B,C>	<B,C>
<A,D>	ϕ	<B,C,D>	<A,B,C>,<A,B,C,D>
<B,C>	<(d)>	<C,D>	<C,D>
<C,D>	<(b),(d)>	<C,D>	<A,C>,<A,C,D>
<A,B,C>	<(d)>	<B,C,D>	<B,C,D>
<A,C,D>	<(b),(d)>	<B,C,D>	<A,B,C>,<A,B,C,D>
<B,C,D>	<(b),(d)>	<C,D>	<A,C,D>
<A,B,C,D>	<(b),(d)>	<B,C,D>	<A,B,C,D>

FIGURE 8

III. THE TOP-DOWN ANALYSIS ALGORITHM

For the reasons given in [3], we want to confine ourselves to finite process structures in the complete lattice whose join is the canonical fps. We will consider the basic process structure of the canonical fps for a system complex to be the "correct" process structure. Although it is always obtainable by computing the canonical fps, our goal here is to arrive at it by successive approximations, so that computing full detail is unnecessary.

Each system in the complex can be partitioned into basic processes independently of the other systems. Since the lattice ordering of the fps's for a single system is also independent of those for other systems in the complex, we need only consider single systems from now on.

The strategy is to begin with the fps whose single node is the total language for the system (the meet of the canonical lattice), and take successive steps upward through the canonical lattice. A single step is taken by splitting a single node of an fps into two nodes, yielding an fps of higher resolution, and therefore closer to the canonical fps. The process can be terminated whenever the system structure it yields is good enough. We call this top-down analysis because it begins with a gross characterization of the system which is gradually refined.

What makes this approach worthwhile is that any changes in the basic process structure of the system's fps resulting from splitting a node will be local to the basic process in which the node is split, so only that process can change, and only those arcs touching the affected node need be re-computed. The rest of the fps and its basic process structure is as good as ever. If this were not true, top-down analysis would be out of the question, because the whole structure would have to be re-computed after each step!

This notion is formalized in the following theorem.

Theorem:

If a new fps is obtained from an old fps by splitting one node, then the only possible change in the basic process structure from the old fps to the new is that the basic process to which the split node belonged may separate into two or more basic processes.

Proof:

Since a basic process partition is uniquely associated with each fps, and the operations (steps through the lattice of fps's) of splitting a node and merging two nodes are inverses of each other, it follows that if we take an fps, merge two nodes and find the basic processes for the new fps, then split the merged node into its original components, the resultant fps and basic process structure must be the same as we started with.

Consider what happens to the basic process structure of an fps when two nodes are merged. If those two nodes are in the same process, the structure does not change at all. If they are in different processes those processes will merge, and may suck other processes into the merger as well.

When we split a node we are performing the inverse operation of this, and so the process containing the split node may or may not separate into several processes. In either case, the other processes will not be affected.

Q.E.D.

It may be desirable to take larger steps through the lattice, by splitting more than one node at a time. The theorem generalizes to say that only processes containing split nodes need be re-examined.

Although the theorem holds for any node split, only certain splits will result in fps's that are still in the canonical lattice. It is not difficult to confine ourselves to these, however, because the language distinctions occurring naturally in this type of analysis always yield fps's in the canonical lattice.

The total language is the union of all the languages L_i which are antecedent, consequent, or accepted message languages from the SR. The nodes of the canonical fps are the intersections of these languages L_i, constructed so that each node language is contained in any L_i it overlaps.

Definition:

A canonical language of an SR is any union of node languages from the canonical fps for the SR.

Any fps all of whose nodes are canonical languages belongs to the canonical lattice (because the lattice consists of the set of all distinct combinations or groupings of the nodes of the canonical fps), and so any split of a node into two parts, both of which are canonical languages, preserves membership in the canonical lattice.

Theorem:

If the total language of an SR is the union of the members of $\{L_1, L_2, \ldots, L_n\}$, where each L_i is an antecedent, consequent, or accepted message language of the SR, then each L_i is a canonical language of the SR, as is any union, intersection, or complement with respect to the total language, of canonical languages.

<u>Proof</u>:

(1) Any L_i is a canonical language.

By construction, any node language in the canonical fps can be expressed as an intersection of languages, one of which is L_i or $\overline{L_i}$. Each node which has L_i in its expression is contained in L_i , so if $\{p_1,p_2,\ldots,p_s\}$ is the set of all such nodes, $p_1 \cup p_2 \cup \ldots \cup p_s$ is contained in L_i . L_i must be contained in $p_1 \cup p_2 \cup \ldots \cup p_s$, because some union of nodes of the canonical fps must cover L_i , but all of the nodes not in $\{p_1,p_2,\ldots,p_s\}$ are in $\overline{L_i}$. Therefore $L_i = p_1 \cup p_2 \cup \ldots \cup p_s$, and L_i is a canonical language.

(2) If L is a canonical language, so is $\bar{L}$.

L is a union of some of the nodes in the canonical fps, so $\bar{L}$ is the union of all those nodes in the canonical fps which are not contained in L .

(3) If L_1 and L_2 are canonical languages, so is $L_1 \cup L_2$. Obvious.

(4) If L_1 and L_2 are canonical languages, so is $L_1 \cap L_2$.

Each of L_1 and L_2 is a union of disjoint nodes of the canonical fps, so $L_1 \cap L_2$ is non-empty if and only if there is some set of nodes of the canonical fps all of which are contained in both L_1 and

L_2 . In this case $L_1 \cap L_2$ is the union of the nodes in this set.

Q. E. D.

It is hard to imagine how one would derive a useful node language which was not a canonical language, according to this theorem. Certainly no standard analysis algorithm would do so, because the antecedent and consequent languages of the SR are the only starting points it would have.

Exactly which node is to be split at any step in the analysis is left to the designer, and may depend on heuristic methods or special needs for detail. Its choice may be the result of a procedure or an interactive session. In any case it is necessary to guide the search enough so that successive steps illuminate more of the process structure--it is useless to split nodes within a basic process of the canonical fps, because the process structure will never change.

At each stage of the analysis we have an fps in which every node larger than a node of the canonical fps is potentially divisible. Ideally we would test an interesting node to discover whether it was contained in a basic process of the canonical fps, in which case we would leave it alone. There is no way to do this except by calculating the basic process structure of the canonical fps, but it would be too general a solution to the guidance problem anyhow.

Assuming the designer has provided a procedure for proposing node splits, all we need to know is whether or not the specifically proposed split will cause the basic process in which it resides to split. If it will not, the cause of defining process structure is not immediately served, and it is probably best to ask the procedure for another proposal.

Because the domain of possible change is limited to the basic process, it may not be too expensive to simply re-evaluate that part of the process structure. Figure 9a shows a basic process and a proposed node split; Figure 9b shows that portion of the fps after the intra-basic-process arcs touching the split node have been re-computed, and the relationships between the nodes of the former basic process have been re-assessed. Here the node split caused its module to decompose, and one basic process separated into three.

To cut the cost further, we can define more easily tested necessary conditions on node splits for them to cause changes in basic process structure. Hopefully, testing these conditions will screen out most of the futile node splits, and only a few will require the more complete computation. They are based on the fact that two nodes (the split halves of an original node) cannot be in different basic processes unless they are in different modules.

An intermediate test of this nature would call for the re-computation of all the 1-arcs touching the split node, followed by a check to see if its module still held together. This is illustrated in Figures 10a and 10b; in 10a the node split fails the test, and in 10b it passes.

A test which is even more rapid and more local (but screens out fewer futile splits) only calls for examination of the 1-arcs actually touching the node to be split. Unless the following three conditions are satisfied, the two split halves must be in the same module, and a basic process split is impossible.

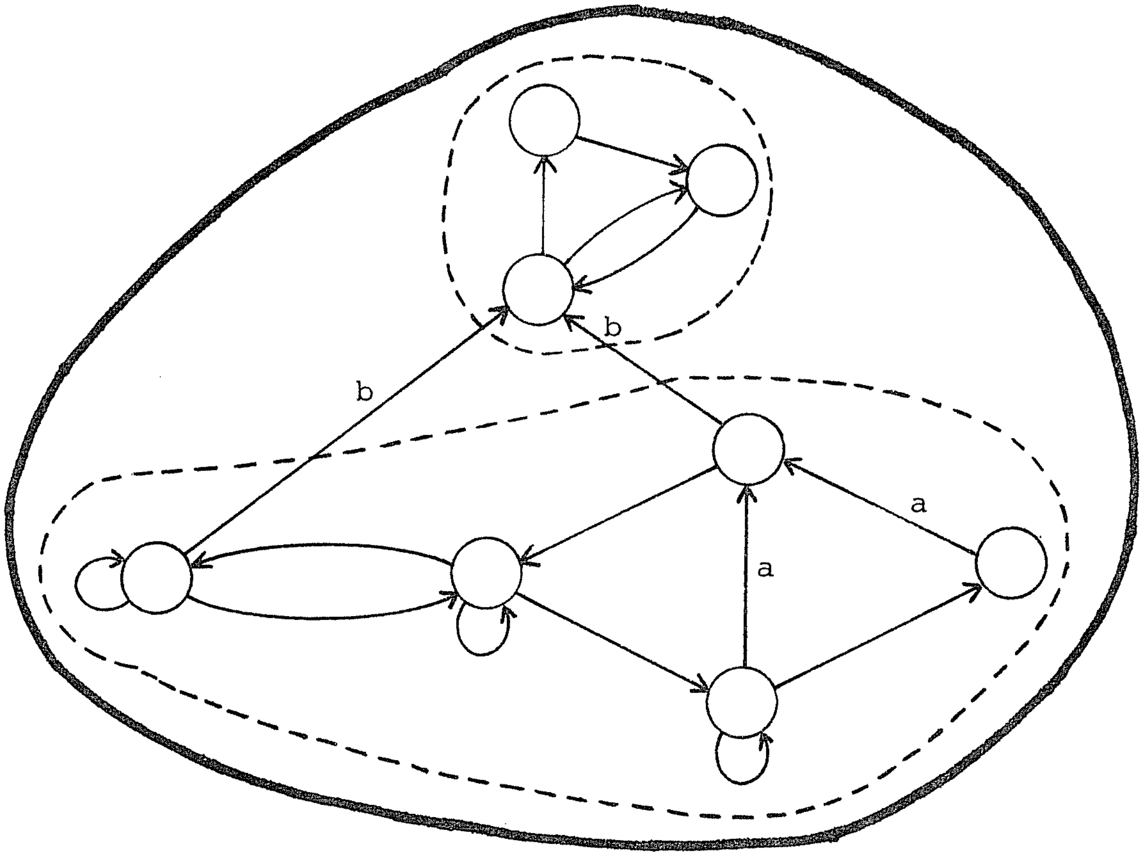


FIGURE 9a

--- module boundary

—— basic process boundary

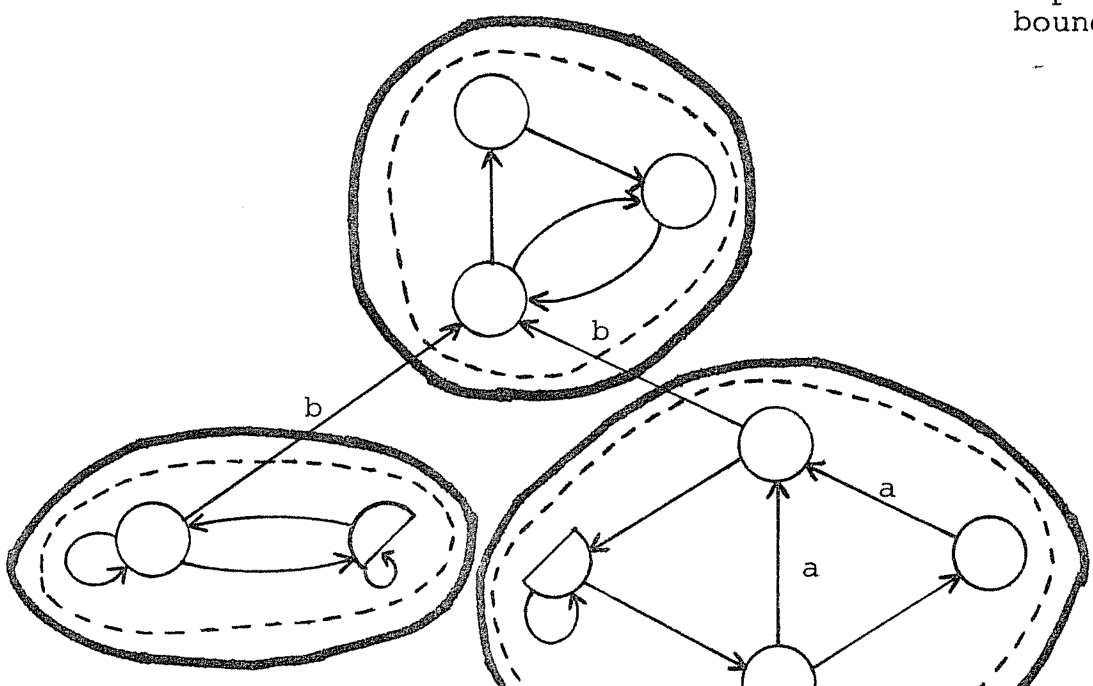


FIGURE 9b

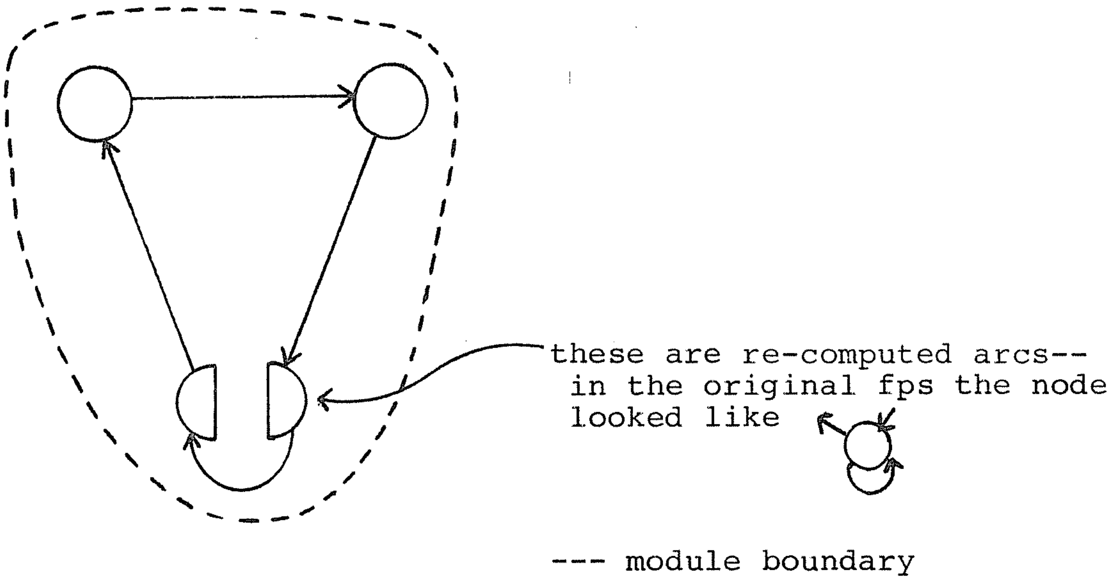


FIGURE 10a

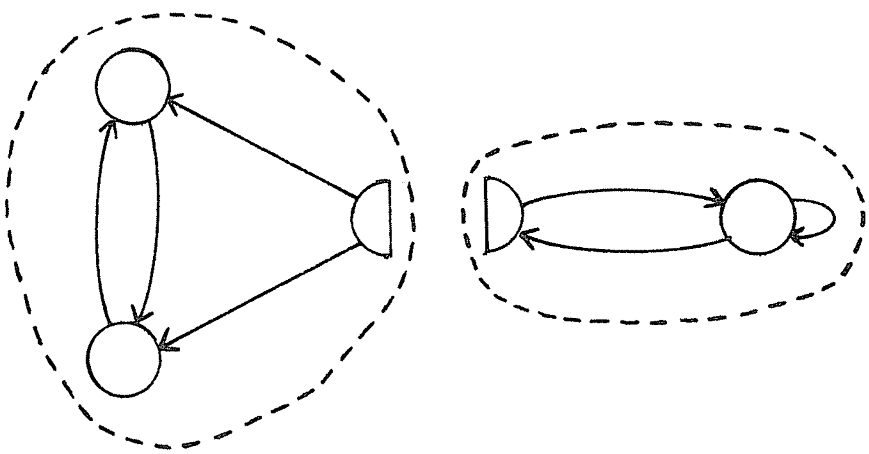


FIGURE 10b

(1) Any l-arc whose destination node is the node to be split must have the successor language (under the transformation by the associated production) of the origin node language contained within one of the split halves.

(2) Any l-arc whose origin node is the node to be split must have the predecessor language of the destination node language contained within one of the split halves.

(3) Any l-arc whose origin node and destination node are both the node to be split must not, when re-computed for the two split halves, appear as a l-arc from one split half to the other.

Figures 11a, b, and c show how violations of conditions (1), (2), and (3) respectively cause the new nodes to be connected by a path of l-arcs, and so be in the same module. Either this test or the test on the whole module can err by accepting a split which will not eventually cause a basic process split, but this will be discovered by the full re-computation.

The process of analysis can terminate when the designer is satisfied with the information he has received, or the procedure for recommending node splits cannot find any more distinctions on which to base a split proposal. Top-down analysis as we have defined it is an algorithm because once all the nodes of the current fps are as small as the nodes of the canonical fps, no further splits are possible.

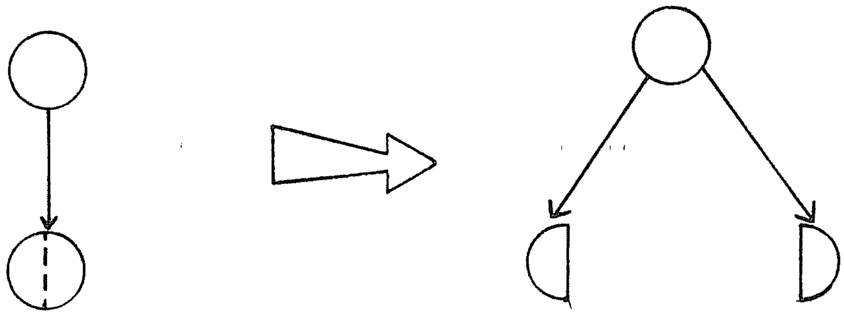


FIGURE 11a

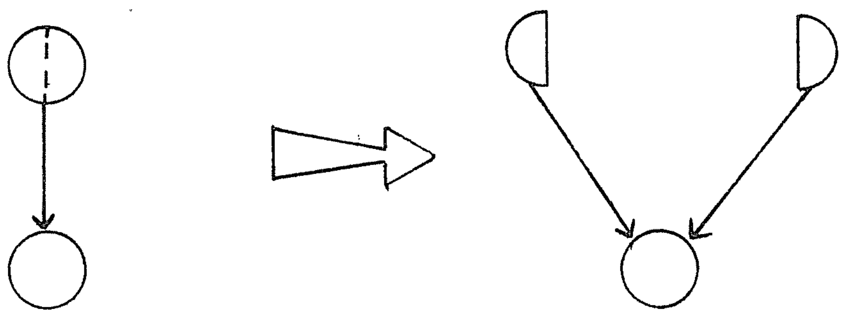


FIGURE 11b

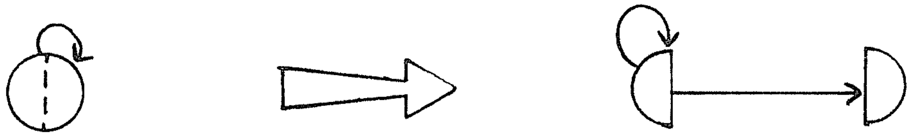


FIGURE 11c

We will now give a very simple example of top-down analysis, to be followed by a more elaborate one in Section V. The system complex consists of the single system whose SR is given in Figure 12. Observers are not included in this example. The system reads records off a tape, deleting them if they are labeled "0" and recording them in a database if they are labeled "1". Figure 13 shows the canonical fps and its basic process structure, which is the target of our top-down analysis. It is obvious that calculating this whole fps would be a waste: the nodes [[tape/Σ_0^*] - [tape/0;Σ*,Σ*/Σ_0^*] - [tape/1; Σ*,Σ*/Σ_0^*]] and [[database (Σ*)] - [database (Σ*;Σ*,Σ*)]] are irrelevant to the actual computations of this system, as can be seen from the initial σ .

The total language (the symbols [,], +, and * are metasymbols used in regular expressions) is [tape/ [0+1+2+3+4+5+6+7+8+9+;+,+/]* + database ([0+1+2+3+4+5+6+ 7+8+9+;+,]*)]. The fps at this stage is shown in Figure 14a. Suppose our procedure makes the obvious proposal of splitting the total language into [tape/[0+1+2+3+4+5+6+7+ 8+9+;+,+/]*] and [database ([0+1+2+3+4+5+6+7+8+9+;+,]*)].

It passes the quick test easily, for the only single-antecedent production to be dealt with has both predecessor and successor languages contained in [tape/[0+1+2+3+4+5+6+ 7+8+9+;+,+/]*]. So we split the node, re-compute arcs and modules (there is split) and get the fps in Figure 14b. Note that we have arrived at the basic process structure of the canonical fps (if we ignore the irrelevant node).

Our procedure might very well recommend a split between [tape/Σ_0^*] - [tape/1[0+1+2+3+4+5+6+7+8+9+;+,+/]*] and

{<u>σ:</u> tape/0;1,3/1;16,256/1;9,81/0;3,10/1;13,169/
<u>and</u>
database (7,49;4,16;12,144;20,400)

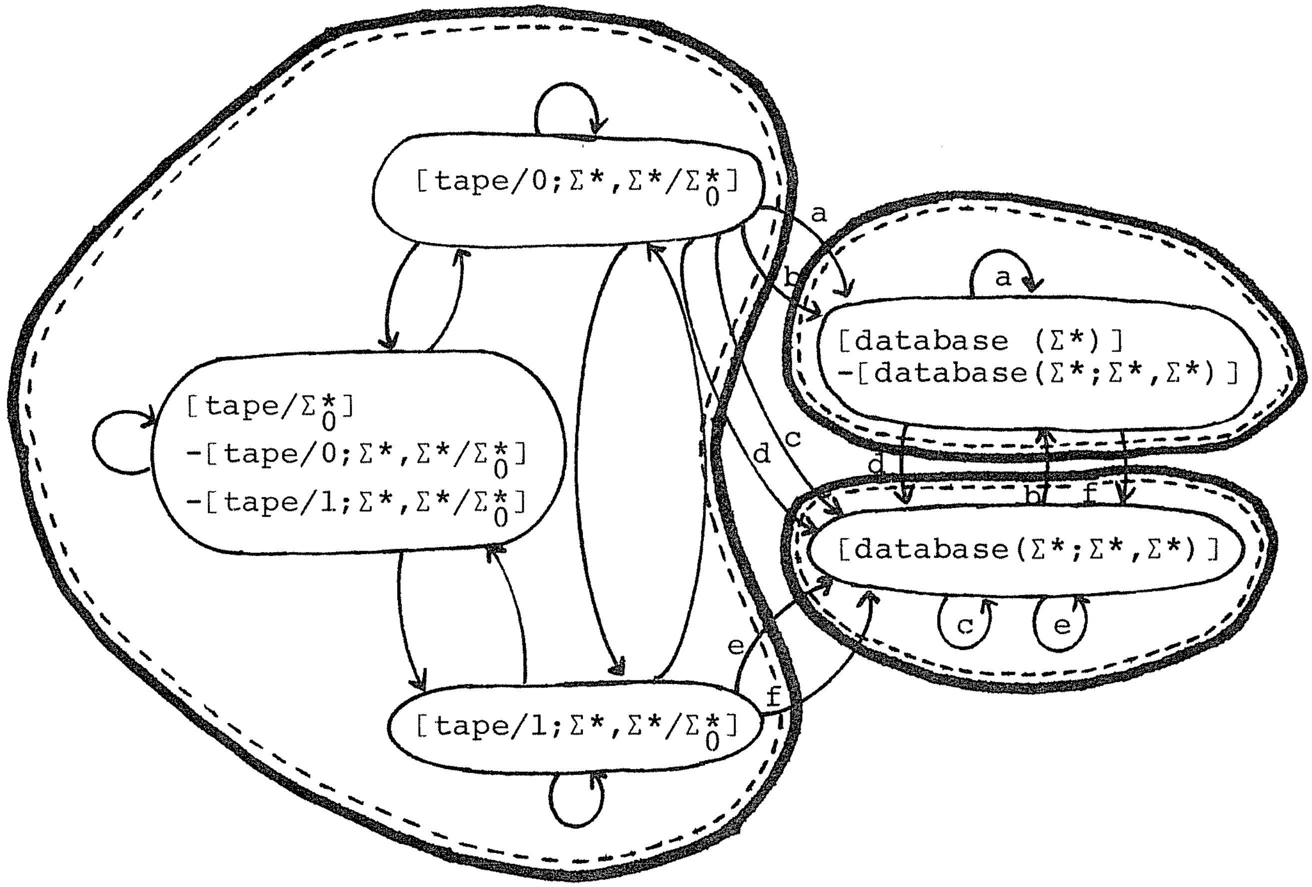
<u>χ:</u> 0123456789;,

<u>π:</u> tape {$\bar{\Delta}$ <u>χ:</u>01 <u>π:</u>};\$,\$/{\$<u>χ:</u>0123456789;,/<u>π:</u>} → tape/$\$_3$ <u>or</u>

tape/0;\$,\$/{\$ <u>χ:</u>0123456789;,/<u>π:</u>} <u>and</u>
database (\$) → database ($\$_4$) <u>or</u>

tape/1;\$,\$/{\$ <u>χ:</u>0123456789;,/<u>π:</u>} <u>and</u>
database(\$) → database($\$_4$;$\$_1$,$\$_2$)}

FIGURE 12



--- module boundary

—— basic process boundary

[,],+,-,*,Σ, and Σ_0 are metasymbols; Σ = (0+1+2+3+4+5+6+7+8+9+;+,), Σ_0 = (0+1+2+3+4+5+6+7+8+9+;+,+/)

FIGURE 13

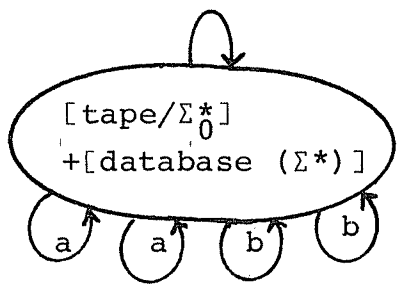


FIGURE 14a

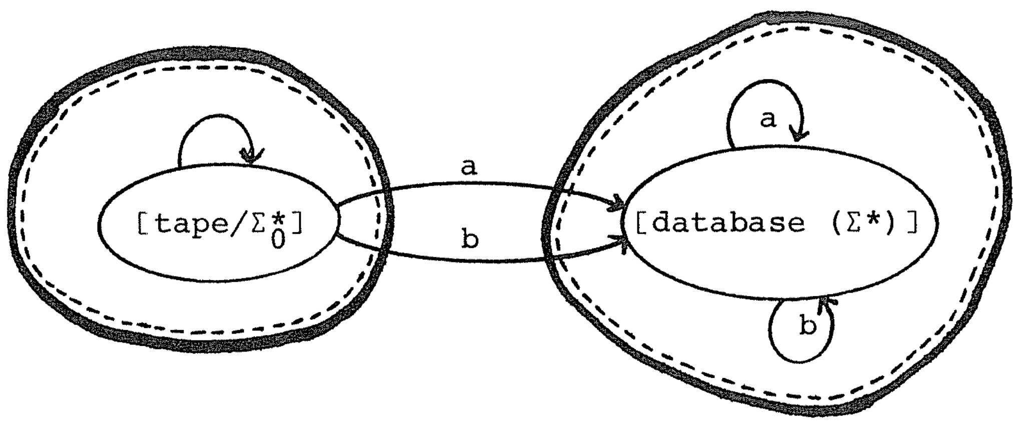


FIGURE 14b

[tape/1[0+1+2+3+4+5+6+7+8+9+;+,+/]*] because the antecedents of the second and third productions suggest that this might be a useful distinction. This split, however, fails to satisfy the third condition of the quick test. Thus we know that the split will produce no new modules. Assuming there are no more proposals for node splits, our top-down analysis procedure has terminated, having brough us to the basic process structure of the canonical fps.

IV. DELIMITED PATTERN LANGUAGES

In [3] it was shown that the class of languages matched by antecedents and generated by consequents is a subclass of the regular languages called pattern languages. Because pattern languages are not closed under complementation, algorithms based on fps's may propose optimizing transformations which cannot be applied to the original systems. Furthermore, under many circumstances pattern languages are no more efficient computationally than regular languages.

Ideally, the antecedent/consequent language class of our formal definition universe would be closed under the Boolean operations, and operations on these languages would be very efficient. One possibility would be to extend the universe so that non-counting languages or even regular languages could be used, as both these classes are closed under the Boolean operations. Such an extension is known to be feasible [4], but it would make the computation problem much worse. We prefer to place restrictions on the forms of antecedents and consequents so that they match delimited pattern languages, a subclass of the regular languages whose properties will be discussed here. They are closed under the Boolean operations, and seem to have excellent computational characteristics. A simple syntax checker could determine whether or not the productions in a system to be analyzed complied with the restrictions, and compliance need not even be enforced: it could be offered as an option to the designer who was particularly interested in analytic efficiency. Thus the use of delimited pattern languages seems to be a convenient way to add to the speed and flexibility of our analysis algorithms.

Definition:

A delimited pattern language (dpl) over Σ is any language denoted by an expression (a delimited pattern expression or dpe) of the following form:

Let each X_i be a subset of Σ, and let i refer to a indexing of $P(\Sigma)$ (the power set of Σ) such that:

$X_0 = \phi$,

$X_i = x_i,\ 1 \le i \le n \quad (|\Sigma| = n)$,

X_i = some subset of Σ with between 2 and n-1 members, $n + 1 \le i \le 2^n - 2$,

$X_{2^n-1} = \Sigma$.

Then a dpe is:

(1) ϕ, denoting the empty set; or

(2) λ, denoting the null string; or

(3) any string of the elements X_i or $X_j^* X_k$ where $X_j \cap X_k = \phi$ $(1 \le i,j,k \le 2^n-1)$; or

(4) X_i^*, or any string of the form defined in (3) with an X_i^* concatenated on the right $(1 \le i \le 2^n-1)$; or

(5) any finite union of distinct terms of the forms defined in (2), (3), and (4).

Note that the dpl's are not closed under concatenation--it is known that the class of non-counting languages is the smallest class containing λ and all single-character strings, and closed under both the Boolean operations and concatenation [5]. We are much more interested in having closure under complementation than closure under concatenation (there is a direct choice between them in our language definition), especially since we can always achieve the effects of concatenation by adding a final delimiting character to the leftmost language and then concatenating.

Theorem:

The class of delimited pattern languages is properly contained in the class of pattern languages.

Proof:

We refer to the presentation of pattern languages in [3]. Any dpe is a pattern expression, so any delimited pattern language is also a pattern language.

Proper containment holds because $X_i^* X_j^*$ is a pattern expression, but not a dpe.

Q. E. D.

First we will show the restrictions on antecedents and consequents necessary to guarantee that our computations will only be carried out on dpl's.

Theorem:

(a) Any non-empty dpl which can be expressed by a dpe with a single term is the language matched by some antecedent;

(b) any non-empty dpl which can be expressed by a dpe with a single term is the language generated by some consequent;

(c) the language matched by any antecedent in which each unsubscripted \$ is either followed immediately by a character not in the vocabulary associated with the variable, is followed immediately by a $\bar{\Lambda}$ variable or subscripted \$ whose vocabulary is disjoint with that associated with the unsubscripted \$, or is the last symbol in the antecedent, is a dpl;

(d) the language generated by any consequent in which each \$ associated with an unsubscripted \$ in the antecedent is either followed immediately by a character not in the vocabulary associated with the variable, is followed immediately by a $\bar{\Lambda}$ variable or a \$ associated with a subscripted \$ in the antecedent whose vocabulary is disjoint with that associated with the unsubscripted \$, or is the last symbol in the consequent, is a dpl.

<u>Proof</u>:

Parts (a) and (b) must be true for dpl's because they are true for pattern languages (there is a proof in [3]).

(c) In [3] it is proved that the language matched by any antecedent is a pattern language. The language is shown to be describable by a string of X_i's and X_j^*'s, in which an X_j^* only appears because of an unsubscripted \$ in the antecedent. If any such occurrence in the antecedent is followed by a character not in the vocabulary associated with the variable, X_j^* will be followed by X_i, $1 \le i \le n$, $X_i \cap X_j = \phi$. If the occurrence is followed by a $\bar{\Lambda}$ or $\$_i$ with a vocabulary X_k, $X_j \cap X_k = \phi$, then X_j^* will be immediately followed by X_k. If the occurrence of X_j^* is at the end of the antecedent, of course, it needs no delimiter. Thus the restrictions are sufficient so that the describing pattern expression is also a delimited pattern expression.

(d) Essentially the same as for (c).

Q. E. D.

Theorem:

The class of dpl's is closed under the Boolean operations.

Proof:

Part I: The class is closed under union.

Let p and q be dpe's for dpl's P and Q, respectively. If p or q is ϕ, then $P \cup Q$ is Q or P, respectively, which is a dpl in either case. Otherwise we can remove redundant terms from $p + q$, and the result is then a dpe for $P \cup Q$.

Part II: The class is closed under intersection.

Let $P = \bigcup_i P_i$ and $Q = \bigcup_k Q_k$ be any two delimited pattern languages where P_i and Q_k can be expressed by the dpe's $\prod_j P_{ij}$ and $\prod_\ell Q_{k\ell}$, respectively. If either P or Q is ϕ, then $P \cap Q$ is ϕ, a dpl.

Otherwise, $$P \cap Q = (\bigcup_i P_i) \cap (\bigcup_k Q_k) = \bigcup_{i,k} (P_i \cap Q_k) .$$

Since we know that dpl's are closed under union, it is only necessary to show that $P_i \cap Q_k$ is a dpl by constructing a dpe for $\prod_j P_{ij} \cap \prod_\ell Q_{k\ell}$.

The following table is a recursive definition of the desired expression. It assumes that if P_i and Q_k consist of a different number of factors, trailing λ's are added to the shorter one so that their lengths become equal. $X_{r\cap s}$ denotes $\{x \mid x \in \Sigma \text{ and } x \in X_r \cap X_s\}$. The resulting expression must be "normalized" by carrying out all factored concatenation (thus removing parentheses), removing trailing λ's from terms, and removing redundant terms.

P_{iu}	Q_{kv}	$\prod_{j=u} P_{ij} \cap \prod_{\ell=v} Q_{k\ell}$
λ	λ	λ
λ	X_s	ϕ
λ	X_s^*	λ
X_r	λ	ϕ
X_r^*	λ	λ
X_r	X_s	$X_{r\cap s}\ (\prod_{j=u+1} P_{ij} \cap \prod_{\ell=v+1} Q_{k\ell})$
X_r^*	X_s	$X_{r\cap s}\ (\prod_{j=u} P_{ij} \cap \prod_{\ell=v+1} Q_{k\ell}) \cup$ $(\prod_{j=u+1} P_{ij} \cap \prod_{\ell=v} Q_{k\ell})$
X_r	X_s^*	$X_{r\cap s}\ (\prod_{j=u+1} P_{ij} \cap \prod_{\ell=v} Q_{k\ell}) \cup$ $(\prod_{j=u} P_{ij} \cap \prod_{\ell=v+1} Q_{k\ell})$
X_r^*	X_s^*	$X_{r\cap s}^*\ [(\prod_{j=u+1} P_{ij} \cap \prod_{\ell=v} Q_{k\ell}) \cup$ $(\prod_{j=u} P_{ij} \cap \prod_{\ell=v+1} Q_{k\ell})]$

The proof that this expression does denote the intersection will not be given here, because it proceeds by case analysis and is lengthy. What we must show is that it is a genuine dpl, i.e. each X_i^* is either at the end of a term or is followed by an X_j, where $X_i \cap X_j = \phi$.

In the result column of the table, the factor $X^*_{r \cap s}$ is followed by $(\prod_{j=u+1} P_{ij} \cap \prod_{\ell=v} Q_{k\ell})$ or $(\prod_{j=u} P_{ij} \cap \prod_{\ell=v+1} Q_{k\ell})$--since these two cases are symmetrical, we will discuss only the first one. $P_{i(u+1)}$ must be λ (the end of the term) or some X_t where $X_r \cap X_t = \phi$; Q_{kv} is X_s^*. If P is λ, $X^*_{r \cap s}$ is immediately followed by the end of the term (all λ's are trailing in P_i, Q_k, and $P_i \cap Q_k$). If P is X_t, $X^*_{r \cap s}$ is immediately followed by $X_{t \cap s}$, and $X_{r \cap s} \cap X_{t \cap s} = \phi$ because $X_r \cap X_t = \phi$.

Part III: The class is closed under complement with respect to Σ^*.

The complement of ϕ is denoted by the dpe $X^*_{2^n-1}$. Let $\bigcup_i P_i$ be any other dpe. $\overline{\bigcup_i P_i} = \bigcap_i \overline{P_i}$, and since we know that dpl's are closed under intersection, all we must show is that the complement of a dpl denoted by a single-term dpe is a dpl.

The complement of λ is $X_{2^n-1} X^*_{2^n-1}$.

Any other term P_i can be expressed as a concatenation $\prod_{j=1}^{n} P_{ij}$.

$$\overline{\prod_{j=1}^{n}} P_{ij} = \sum_{k=0}^{n-1} S_k + \sum_{k=0}^{n} L_k$$

where: $S_k = \phi$ if $k = 0,\ n = 1,\ P_{i1} = X^*_u$

or $k > 0,\ P_{ik} = X^*_u$

or $k = n-1,\ n > 1,\ P_{ik} = X_u,$

$P_{i(k+1)} = X^*_w$;

$S_k = \lambda$ if $k = 0,\ P_{i1} = X_u$;

$S_k = \prod_{j=1}^{k} P_{ij}$ if $k > 0,\ P_{ik} = X_u,\ P_{i(k+1)} = X_w$;

$S_k = \prod_{j=1}^{k+1} P_{ij}$ if $k = 0,\ n > 1,\ P_{i1} = X^*_u$

or $0 < k < n-1,\ P_{ik} = X_u,\ P_{i(k+1)} = X^*_w$;

$L_k = \phi$ if $0 < k < n,\ P_{ik} = X^*_w$

or $k = n-1,\ P_{ik} = X_w,\ P_{i(k+1)} = X^*_u$;

$L_k = \prod_{j=1}^{k} P_{ij}\ X_{\overline{k}} X^*_{2^n-1}$ if $k = n,\ P_{ik} = X^*_w$;

$L_k = \prod_{j=1}^{k} P_{ij} X_{\overline{k+1}} X^*_{2^n-1}$ if $0 < k < n,\ P_{ik} = X_w,$

$P_{i(k+1)} = X_u$ or $k = 0,\ P_{i(k+1)} = X_w$

or $k = 0,\ n = 1,\ P_{i(k+1)} = X^*_w$;

$$L_k = \prod_{j=1}^{k+1} P_{ij} X_{\overline{(k+1)\cap(k+2)}} X^*_{2^n-1} \quad \text{if} \quad 0 < k < n-1,$$

$$P_{ik} = X_w, P_{i(k+1)} = X^*_u$$

$$\text{or} \quad k = 0,\ n > 1,\ P_{i(k+1)}$$

$$= X^*_u \ ;$$

$$L_k = \prod_{j=1}^{k} P_{ij} X_{2^n-1} X^*_{2^n-1} \quad \text{if} \quad k = n,\ P_{ik} = X_w \ .$$

As before, $X_{\overline{k\cap\ell}} = \{x | x \notin X_k \text{ or } x \notin X_\ell\}$, etc.

$\prod_{j=1}^{0} P_{ij} = \lambda$. We assume that ϕ's, λ's, and redundant terms will be removed from the resulting expression.

First we must demonstrate that this is the correct formula for the complement. Any single-term dpl is accepted by a simple, deterministic finite automaton of the type shown in Figure 15a, which accepts the dpl $X_i^* X_j X_k^* X_\ell X_m X_n^*$. It is understood that on any inputs except those causing labeled transitions, the machine goes to a non-accepting sink state which is not shown.

Since the machine is deterministic, all we have to do to obtain a machine accepting the complement is interchange the accepting and

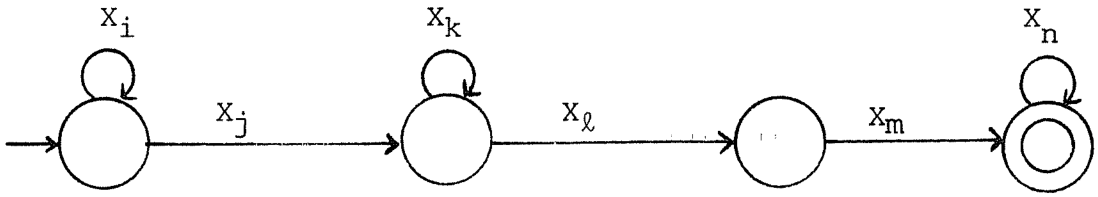


FIGURE 15a.

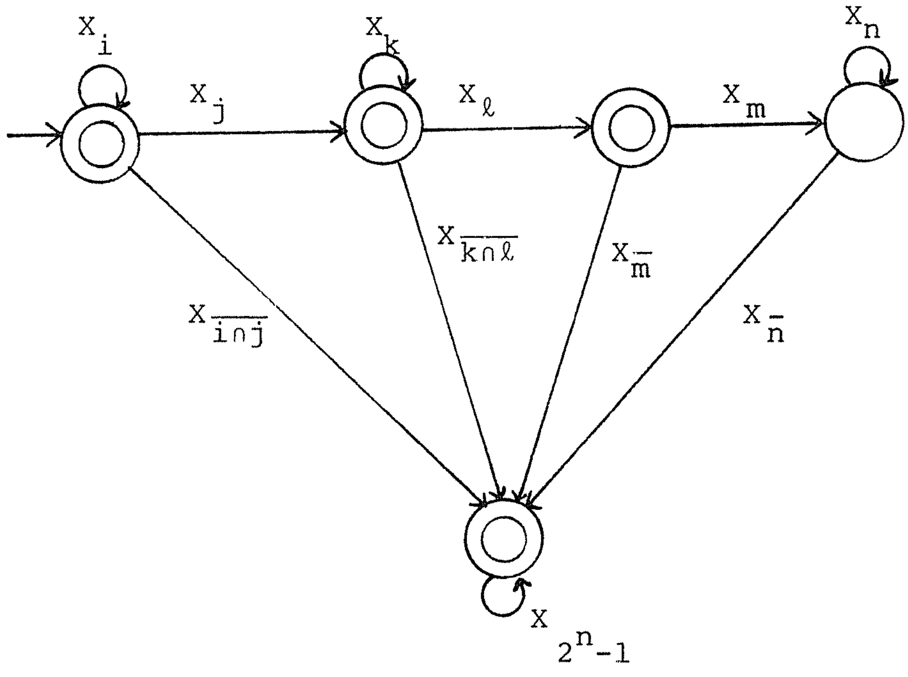


FIGURE 15b.

non-accepting states. This is done in Figure 15b, with the former sink state now explicit. The formula is simply an expression of the language accepted by this complement machine. Each S_k is a language accepted at a state corresponding to an explicit non-accepting state of the original machine; each L_k is a language accepted at the former sink state (and arriving by a different path from the other L_k's).

By examining each expression for S_k or L_k and the conditions under which it is used, it can be determined easily that each is a valid dpe. Thus the sum of these terms is also a valid dpe.

Q. E. D.

Corollary:

The complement of any dpl with respect to any containing dpl is a dpl.

Proof:

$$L_1 - L_2 = L_1 \cap \overline{L_2} .$$

We have now established that delimited pattern languages have all the properties we have claimed for them except computational efficiency. We are interested in the following operations: taking the union or intersection of two sets, taking the complement of one set with respect to another, testing two sets for equality or containment, and performing an R-model step on a set. We will show that on dpl's these operations can be performed almost exclusively by string manipulation algorithms, which are intrinsically faster than the finite-automaton-construction-and-manipulation necessary for regular languages (and even subclasses of them larger than the dpl's).

The existence of string manipulation algorithms for unions, intersections, complements, and differences is shown in the preceding proof. They can also be used to calculate R-model steps, when the antecedents match dpl's (because they obey the restrictions in the earlier theorem) and they are being applied to dpl's. This is because a match of an antecedent to a language is really just the intersection of the antecedent language and the state language, and can be computed like any other. Once the match is found, constructing the consequent with the correct variable values is trivial; string, rather than machine, manipulation will suffice throughout. It is also worth noting that dpl antecedents have the desirable characteristic of matching any process state in one way only, at most.

The most challenging operations are those of testing two languages for equality or containment, but the dpl's have a very important property that will facilitate them: any dpl has a unique string representation which we call the canonical form. For any two dpl's represented by canonical form dpe's, string identity is equivalent to language equality! The containment test is also greatly simplified.

Theorem:

For any dpl there is a unique dpe to which any other dpe denoting the language can be algorithmically transformed.

Proof:

The essence of the proof is that any dpl is accepted by a deterministic reduced finite automaton having no loops along a path from the initial state to an accepting state with path lengths greater than one. Since the finite automaton is deterministic and reduced, it is unique for the dpl it accepts. Since it has only trivial loops, there is a finite number of distinct paths between the initial state and some accepting state. Each distinct path corresponds to a term in the canonical form dpe; the terms can be sorted according to the indexing on X_i's to determine a particular order for them.

Any single-term dpe corresponds straightforwardly to a deterministic finite automaton with no loops of path length greater than one, as shown in Figure 15. The machine accepting a dpl with a multiple-term dpe may have non-deterministic transitions out of the initial state only (for example $X_i X_j^* X_k + X_i X_m^* X_n$, the machine for which is shown in Figure 16). We know that we can make a deterministic machine from any non-deterministic one, but it must be the case that the deterministic one still has no loops with path length greater than one.

This is true because the deterministic machine is formed by a simple subset construction on the states of the non-deterministic machine; the transition from any state of the new machine on any input is to the state which is the union of the states of the old machine to which transitions under that input are made from the states of the old machine included in the subset.

For instance, let the non-deterministic machine have initial state S_0, and i linear sequences of states $S_{i1}, S_{i2}, \ldots, S_{ij_i}$, each one corresponding to a single term in a dpe. Then a state of the deterministic machine is some subset of $\{S_0, \underset{i}{\cup} (\overset{j_i}{\underset{k=1}{\cup}} S_{ik})\}$. From any old state S_{ik}, the only possible transitions are to S_{ik} or $S_{i(k+1)}$. Thus from any new state $\{S_{i_1k_1}, S_{i_2k_2}, \ldots, S_{i_nk_n}\}$, the transition on any input must be some subset of $\{S_{i_1k_1} \oplus S_{i_1(k_1+1)}, S_{i_2k_2} \oplus S_{i_2(k_2+1)}, \ldots, S_{i_nk_n} \oplus S_{i_n(k_n+1)}\}$, where $\oplus$ signifies that one or the other may be present, but not both. Clearly the only possible input sequences which would return the deterministic machine to a previous state $\{S_{i_1k_1}, S_{i_2k_2}, \ldots, S_{i_nk_n}\}$ would be any sequences of inputs consisting entirely of inputs which themselves cause transitions from the state to itself--loops of path length one.

Thus we have shown that any dpl is accepted by a deterministic finite automaton with no non-trivial loops. Next we must show that this machine can be reduced without the introduction of non-trivial loops.

Any reduction algorithm operates by merging equivalent states, where two states S_1 and S_2 of machine M are equivalent if and only if $T(M_{S_1}) = T(M_{S_2})$, where M_{S_i} is the finite automaton obtained by treating state S_i as the initial state of M, and $T(M)$ denotes the set of all input sequences accepted by M.

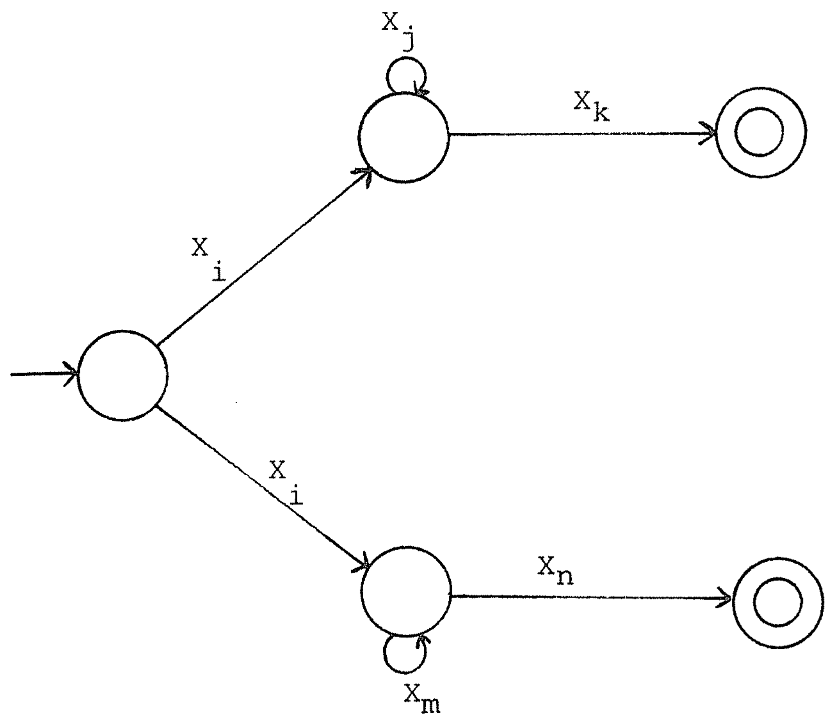


FIGURE 16

In our deterministic machine with no non-trivial loops, there is a finite number of paths from the initial state to an accepting state, and the states along each path can be ordered (numbered) so that no transitions are ever made to a lower-numbered state from a higher-numbered one (as a consequence of previous parts of this proof). The only way that a merging of states in the reduction process could produce a non-trivial loop would be to merge two states S_i and S_j , $j > i + 1$, in the same numbered sequence. But since there is a sub-sequence of an accepting path from S_i to S_j , some string w , $|w| \geq 2$, causes a transition from S_i to S_j ; consequently $S_i \equiv S_j \Rightarrow T(M_{S_i}) = T(M_{S_j}) \Rightarrow wT(M_{S_j}) \subseteq T(M_{S_j})$, a contradiction. Therefore no two states such as S_i and S_j can be merged during reduction, and so reduction of the machine cannot introduce non-trivial loops.

Q.E.D.

We will now give an example of the conversion of dpe $X_iX_j^* + X_iX_k^*X_\ell$, where $X_j \cap X_k \neq \phi$, $X_j \cap X_\ell \neq \phi$, and all other alphabets are pairwise disjoint, to its canonical form. The non-deterministic machine is shown in Figure 17a. The deterministic reduced machine, shown in Figure 17b, has transitions on the disjoint input sets X_i , $X_{j\cap k}$, $X_{j\cap\ell}$, $X_{j-k-\ell}$, X_{k-j}, and $X_{\ell-j}$. We have numbered the states so that we can indicate the four paths to an accepting state: <1,2>, <1,2,3>, <1,2,5>, <1,2,4,5>. Thus the canonical form (with terms in sorted order) is $X_iX_{j\cap k}^*$ $+ X_iX_{j\cap k}^*X_{j-k}X_j^* + X_iX_{j\cap k}^*X_{k-j}X_k^*X_\ell + X_iX_{j\cap k}^*X_{\ell-j}$.

Conversion of a dpl to its canonical form obviously needs to be streamlined by the development of a string algorithm to do it. It would also be helpful if the string operations for complementation and intersection preserved canonical form. This is already true of the formula for complementing a single-term dpe! Since the complement of a multiple-term dpe is the intersection of these single-term complements, the intersection formula becomes crucial. There is much reason to expect that the intersection formula can also be made to preserve the canonical form, however. The essence of canonical form is that all the terms in the dpe represent disjoint languages. If we can take two languages in canonical form, $P_1 + P_2 + \ldots + P_n$ and $Q_1 + Q_2 + \ldots + Q_m$, all the termwise intersections $P_i \cap Q_j$ must also be disjoint, and this already indicates some similarity of the result with the canonical form.

The efficiency of top-down analysis will be greatly enhanced by the use of delimited pattern languages and associated string algorithms. The improvement over regular language computations might very well mean an order-of-magnitude increase in the size of practically analyzable systems.

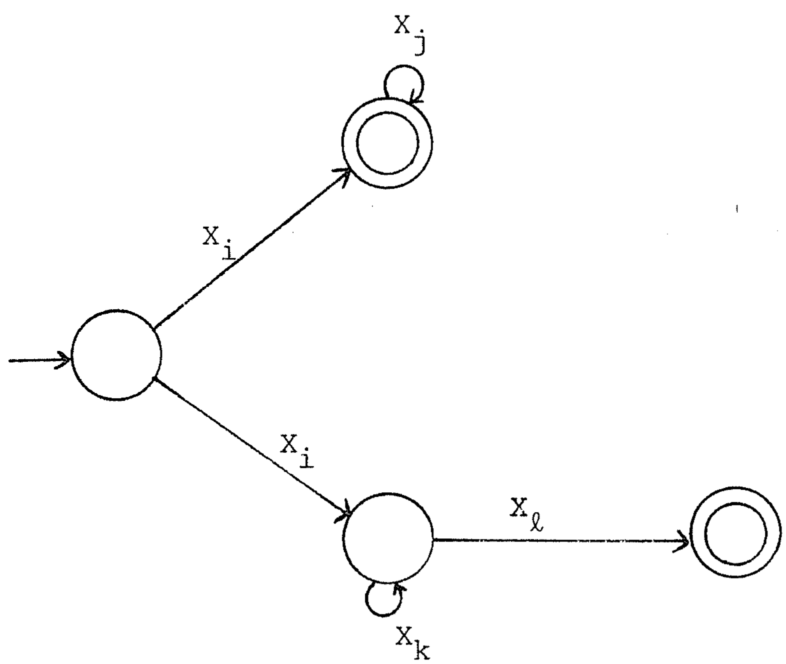


FIGURE 17a.

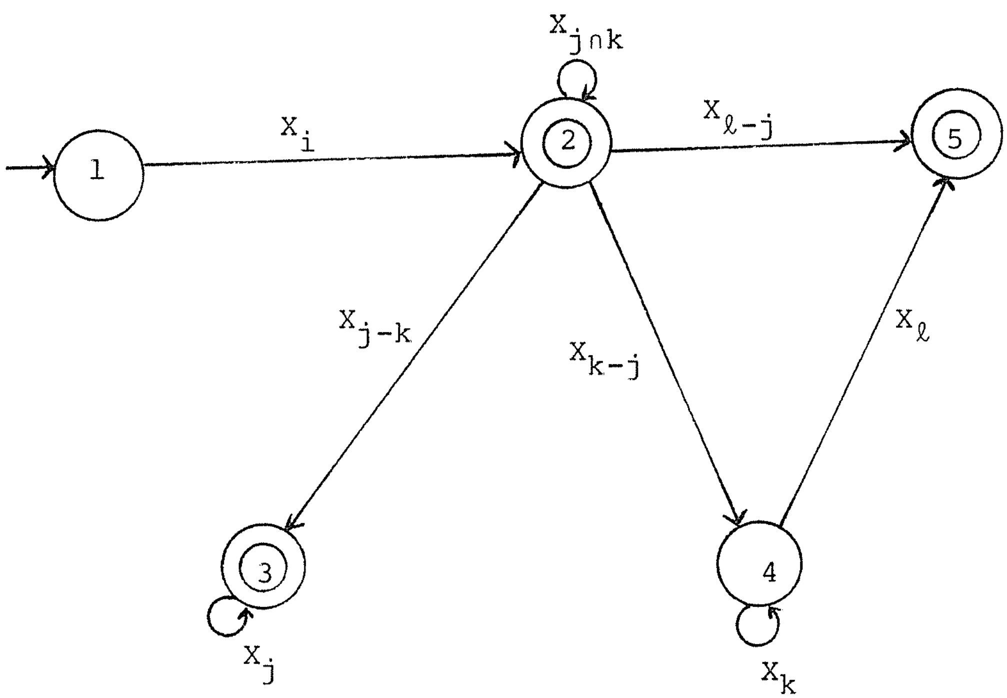


FIGURE 17b.

V. A DETAILED EXAMPLE

We will now examine a system complex composed of two systems, a teletype handler (TH) and a processing system (PS). Here are the SR's for TH and PS, respectively.

{<u>σ:</u> tty-handler <u>and</u> may-transmit <u>and</u>
abuf()in <u>and</u> bbuf()in <u>and</u> cbuf()in

<u>χ:</u> 0123456789#

<u>π:</u> tty-handler → tty-handler <u>or</u>
tty-handler → handler <u>or</u>
tty-handler → a <u>or</u>
tty-handler → b <u>or</u>
tty-handler → c <u>or</u>
a and abuf($)in → abuf($$_1$)in <u>or</u>
b and bbuf($)in → bbuf($$_1$)in <u>or</u>
c and cbuf($)in → cbuf($$_1$)in <u>or</u>
a:$\bar{\#}$ and abuf($)in → abuf($$_1$$\bar{\#}_1$)in <u>or</u>
b:$\bar{\#}$ and bbuf($)in → bbuf($$_1$$\bar{\#}_1$)in <u>or</u>
c:$\bar{\#}$ and cbuf($)in → cbuf($$_1$$\bar{\#}_1$)in <u>or</u>
a:# and abuf($)in → transmit(a:$$_1$#) <u>or</u>
b:# and bbuf($)in → transmit(b:$$_1$#) <u>or</u>
c:# and cbuf($)in → transmit(c:$$_1$#) <u>or</u>
a:# and abuf($)in → abuf()out <u>or</u>
b:# and bbuf($)in → bbuf()out <u>or</u>
c:# and cbuf($)in → cbuf()out <u>or</u>
transmit({$\bar{\Delta}$<u>χ:</u> abc <u>π:</u>}:$) <u>and</u> may-not-transmit →
transmit($\bar{\Delta}_1$:$$_1$) <u>or</u>
may-transmit → may-transmit <u>or</u>
may-not-transmit → may-not-transmit <u>or</u>
transmit({$\bar{\Delta}$<u>χ:</u> abc <u>π:</u>}:$) <u>and</u> may-transmit →
$\bar{\Delta}_1$:$$_1$ → processor <u>or</u>

transmit({$\bar{\Delta}\underline{\chi:}$ abc $\underline{\pi:}$}:\$) and may-transmit →
    acknowledge → processor — or

transmit({$\bar{\Delta}\underline{\chi:}$ abc $\underline{\pi:}$}:\$) and may-transmit →
    may-not-transmit → may-transmit — or

acknowledge → may-send → may-not-send — or

a:\$ and abuf()out → abuf($\$_1$)out → abuf()out — or

b:\$ and bbuf()out → bbuf($\$_1$)out → bbuf()out — or

c:\$ and cbuf()out → cbuf($\$_1$)out → cbuf()out — or

abuf()out → abuf()out — or
bbuf()out → bbuf()out — or
cbuf()out → cbuf()out — or

abuf($\bar{\Delta}$\$)out → $\bar{\Delta}_1$ → tty-a — or

bbuf($\bar{\Delta}$\$)out → $\bar{\Delta}_1$ → tty-b — or

cbuf($\bar{\Delta}$\$)out → $\bar{\Delta}_1$ → tty-c — or

abuf($\bar{\#}$\$)out → abuf($\$_1$)out — or

bbuf($\bar{\#}$\$)out → bbuf($\$_1$)out — or

cbuf($\bar{\#}$\$)out → cbuf($\$_1$)out — or

abuf(#)out → abuf()in — or
bbuf(#)out → bbuf()in — or
cbuf(#)out → cbuf()in — }

{$\underline{\sigma:}$ processor-system and may-send and
a0()0a and b0()0b and c0()0c and
a1()1a and b1()1b and c1()1c

$\underline{\chi:}$ 0123456789#

$\underline{\pi:}$ processor-system → processor-system — or
processor-system → processor — or

acknowledge → may-transmit → may-not-
    transmit — or

{$\bar{\Delta}\underline{\chi:}$ abc $\underline{\pi:}$}{$\bar{\Delta}\underline{\chi:}01\underline{\pi:}$}(){$\bar{\Delta}\underline{\chi:}01\underline{\pi:}$}{$\bar{\Delta}\underline{\chi:}$ abc $\underline{\pi:}$}→
    $\bar{\Delta}_1\bar{\Delta}_2()\bar{\Delta}_3\bar{\Delta}_4$ — or

{$\bar{\Delta}\underline{\chi:}$ abc $\underline{\pi:}$}:{$\bar{\Delta}\underline{\chi:}01\underline{\pi:}$}\$ and
    {$\bar{\Delta}\underline{\chi:}$ abc $\underline{\pi:}$}{$\bar{\Delta}\underline{\chi:}01\underline{\pi:}$}(){$\bar{\Delta}\underline{\chi:}01\underline{\pi:}$}{$\bar{\Delta}\underline{\chi:}$ abc $\underline{\pi:}$}
    → $\bar{\Delta}_3\bar{\Delta}_4(\$_1)\bar{\Delta}_5\bar{\Delta}_6$ → $\bar{\Delta}_3\bar{\Delta}_4()\bar{\Delta}_2\bar{\Delta}_1$ — or

a0({$\bar{\Delta}$\$<processor 0>})0a → send(a:$\bar{\Delta}_1\$_1$) — or

a1({$\bar{\Delta}$\$<processor 1>})1a → send(a:$\bar{\Delta}_1\$_1$) — or

$a0()0a\ \underline{\text{and}}\ b0(\{\bar{\Delta}\$<\text{processor }0>\})0b \rightarrow \text{send}(b{:}\bar{\Delta}_1\$_1)$ $\underline{\text{or}}$

$a1()1a\ \underline{\text{and}}\ b1(\{\bar{\Delta}\$<\text{processor }1>\})1b \rightarrow \text{send}(b{:}\bar{\Delta}_1\$_1)$ $\underline{\text{or}}$

$a0(\bar{\Delta}\$)0a\ \underline{\text{and}}\ b0(\bar{\Delta}\$)0b \rightarrow b0(\bar{\Delta}_2\$_2)0b$ $\underline{\text{or}}$

$a1(\bar{\Delta}\$)1a\ \underline{\text{and}}\ b1(\bar{\Delta}\$)1b \rightarrow b1(\bar{\Delta}_2\$_2)1b$ $\underline{\text{or}}$

$a0()0a\ \underline{\text{and}}\ b0()0b\ \underline{\text{and}}\ c0(\{\bar{\Delta}\$<\text{processor }0>\})0c$
$\quad \rightarrow \text{send}(c{:}\bar{\Delta}_1\$_1)$ $\underline{\text{or}}$

$a1()1a\ \underline{\text{and}}\ b1()1b\ \underline{\text{and}}\ c1(\{\bar{\Delta}\$<\text{processor }1>\})1c$
$\quad \rightarrow \text{send}(c{:}\bar{\Delta}_1\$_1)$ $\underline{\text{or}}$

$a0(\bar{\Delta}\$)0a\ \underline{\text{and}}\ c0(\bar{\Delta}\$)0c \rightarrow c0(\bar{\Delta}_2\$_2)0c$ $\underline{\text{or}}$

$a1(\bar{\Delta}\$)1a\ \underline{\text{and}}\ c1(\bar{\Delta}\$)1c \rightarrow c1(\bar{\Delta}_2\$_2)1c$ $\underline{\text{or}}$

$b0(\bar{\Delta}\$)0b\ \underline{\text{and}}\ c0(\bar{\Delta}\$)0c \rightarrow c0(\bar{\Delta}_2\$_2)0c$ $\underline{\text{or}}$

$b1(\bar{\Delta}\$)1b\ \underline{\text{and}}\ c1(\bar{\Delta}\$)1c \rightarrow c1(\bar{\Delta}_2\$_2)1c$ $\underline{\text{or}}$

$\text{send}(\{\bar{\Delta}\ \underline{\chi{:}}\ abc\ \underline{\pi{:}}\}{:}\bar{\Delta}\$)\ \underline{\text{and}}\ \text{may-not-send} \rightarrow$
$\quad \text{send}(\bar{\Delta}_1{:}\bar{\Delta}_2\$_1)$ $\underline{\text{or}}$

$\text{may-send} \rightarrow \text{may-send}$ $\underline{\text{or}}$
$\text{may-not-send} \rightarrow \text{may-not-send}$ $\underline{\text{or}}$

$\text{send}(\{\bar{\Delta}\ \underline{\chi{:}}\ abc\ \underline{\pi{:}}\}{:}\bar{\Delta}\$)\ \underline{\text{and}}\ \text{may-send} \rightarrow \bar{\Delta}_1{:}\bar{\Delta}_2\$_1$
$\quad \rightarrow \text{handler}$ $\underline{\text{or}}$

$\text{send}(\{\bar{\Delta}\ \underline{\chi{:}}\ abc\ \underline{\pi{:}}\}{:}\bar{\Delta}\$)\ \underline{\text{and}}\ \text{may-send} \rightarrow \text{acknowledge}$
$\quad \rightarrow \text{handler}$ $\underline{\text{or}}$

$\text{send}(\{\bar{\Delta}\ \underline{\chi{:}}\ abc\ \underline{\pi{:}}\}{:}\bar{\Delta}\$)\ \underline{\text{and}}\ \text{may-send} \rightarrow \text{may-not-send}$
$\quad \rightarrow \text{may-send}$ $\}$

TH receives characters from each of the three teletypes A, B, and C , which are the observers for this system complex. Each teletype has its own transmission channel (a, b, or c) to TH, and the messages are labeled according to their origins (a:, b:, or c:). TH has a buffer for each teletype in which it collects the characters from that teletype until the end-of-line character # is transmitted.

TH then attempts to send the completed line to the processing system for processing, but this transmission may be delayed until an acknowledgment of TH's last transmission is received. The two systems have a two-way acknowledgment agreement to prevent message loss by overwriting--thus the system complex is rate-independent.

When TH receives the processor's output from this input line (which is guaranteed to come after a finite delay), it inserts it in the appropriate teletype's buffer and then transmits it, character by character, to the teletype. TH will not work correctly unless the teletypes make no new transmissions until they have received replies from their previous transmissions.

PS receives input lines, applies processors to them, and transmits the output lines back to TH. The processors themselves are buried in RPR's; most of the productions of PS are devoted to allocating them.

The first character of each input line is a 0 or 1, indicating which of the two processors this is intended for. On any given system step PS could receive requests by A, B, and C for the same processor, but each processor can only work on one input at a time. Therefore the requests are buffered, then satisfied according to a scheme which gives A the highest priority and C the lowest.

As an example of systems modeling, the most interesting thing about this complex is the number of realistic trade-offs between generality and efficiency that went into its design. The most obvious is the fact that the systems can handle only a fixed number of active teletypes, using fixed buffer spaces, all bound to constant names.

We begin top-down analysis with the only information immediately available: the antecedent and consequent languages of the productions. These lists only include the languages which belong to the total languages for each system, and therefore exclude messages the generating systems cannot receive themselves. Our regular expressions use the metasymbols Σ, Σ_, λ, *, +, [, and]. Σ denotes the alphabet (0+1+2+3+4+5+6+7+8+9+#), and Σ_ denotes Σ with the # missing.

TH	PS
tty-handler	processor-system
handler	processor
may-transmit	may-send
may-not-transmit	may-not-send
acknowledge	acknowledge
a	a0()0a
b	a1()1a
c	b0()0b
abuf()in	b1()1b
bbuf()in	a0(Σ Σ*)0a
cbuf()in	a1(Σ Σ*)1a
abuf(Σ*)in	b0(Σ Σ*)0b
bbuf(Σ*)in	b1(Σ Σ*)1b
cbuf(Σ*)in	c0(Σ Σ*)0c
abuf(Σ*Σ_)in	c1(Σ Σ*)1c
bbuf(Σ*Σ_)in	[a+b+c][0+1]()[0+1][a+b+c]
cbuf(Σ*Σ_)in	[a+b+c][0+1](Σ*)[0+1][a+b+c]
a:Σ*	[a+b+c]:[0+1]Σ*
b:Σ*	send(a:ΣΣ*)
c:Σ*	send(b:ΣΣ*)
a:#	send(c:ΣΣ*)
b:#	send([a+b+c]:ΣΣ*)

TH continued

c:#

a:Σ_

b:Σ_

c:Σ_

transmit(a:Σ*#)

transmit(b:Σ*#)

transmit(c:Σ*#)

transmit([a+b+c]:Σ*)

abuf()out

bbuf()out

cbuf()out

abuf(Σ*)out

bbuf(Σ*)out

cbuf(Σ*)out

abuf(ΣΣ*)out

bbuf(ΣΣ*)out

cbuf(ΣΣ*)out

abuf(Σ_Σ*)out

bbuf(Σ_Σ*)out

cbuf(Σ_Σ*)out

abuf(#)out

bbuf(#)out

cbuf(#)out

For reference we will give the nodes of the canonical fps's for these systems, although they are not necessary at this stage.

TH	PS
tty-handler	processor-system
handler	processor
a	acknowledge
b	may-send
c	may-not-send
abuf()in	[a+b+c]:[0+1]Σ*
bbuf()in	send(a:ΣΣ*)
cbuf()in	send(b:ΣΣ*)
abuf(Σ*Σ_)in	send(c:ΣΣ*)
bbuf(Σ*Σ_)in	a0()0a
cbuf(Σ*Σ_)in	a1()1a
abuf(Σ*#)in	b0()0b
bbuf(Σ*#)in	b1()1b
cbuf(Σ*#)in	a0(ΣΣ*)0a
a:#	a1(ΣΣ*)1a
b:#	b0(ΣΣ*)0b
c:#	b1(ΣΣ*)1b
a:Σ_	c0(ΣΣ*)0c
b:Σ_	c1(ΣΣ*)1c
c:Σ_	c[0+1]()[0+1]c
[a:] + [a:ΣΣΣ*]	
[b:] + [b:ΣΣΣ*]	
[c:] + [c:ΣΣΣ*]	
may-not-transmit	
may-transmit	
transmit(a:Σ*#)	
transmit(b:Σ*#)	
transmit(c:Σ*#)	
transmit([a+b+c]:[λ+Σ*Σ_])	
abuf()out	
bbuf()out	
cbuf()out	
abuf(#)	
bbuf(#)	
cbuf(#)	

TH continued

acknowledge
abuf(Σ_Σ*)out
bbuf(Σ_Σ*)out
cbuf(Σ_Σ*)out
abuf(#Σ*)out
bbuf(#Σ*)out
cbuf(#Σ*)out

Now we are ready to begin top-down analysis. We decide that our first step may entail splitting the total language into several sublanguages, instead of just two; the syntactic criterion on which the splits will be made is that an antecedent or consequent language containing other antecedent or consequent languages (but not contained in any others) will become a separated node. This is based on the notion that the existence of a covering antecedent or consequent indicates a meaningful first-order grouping of antecedent/consequent languages.

According to this criterion, the nodes of the first fps would be:

TH

(1) abuf(Σ*)in
(2) bbuf(Σ*)in
(3) cbuf(Σ*)in
(4) abuf(Σ*)out
(5) bbuf(Σ*)out
(6) cbuf(Σ*)out
(7) a:Σ*
(8) b:Σ*
(9) c:Σ*
(10) transmit([a+b+c]:Σ*)
(11) tty-handler + handler + a + b + c + may-transmit + may-not-transmit + acknowledge

PS

(12) [a+b+c][0+1](Σ*)[0+1][a+b+c]
(13) [a+b+c]:[0+1]Σ*
(14) send([a+b+c]:ΣΣ*)
(15) processor-system + processor + acknowledge + may-send + may-not-send

Since this makes too many nodes in TH for an initial fps, however, languages (1), (2), and (3) will be united on the basis of their obvious syntactic similarities, as will (4), (5), and (6), and (7), (8), and (9). The fps is shown in Figure 18. Note that there is only one arc component drawn from any particular node to any other particular node--each letter label indicates the existence of a separate arc component with that label; () stands for a component with no label, i.e. a 1-arc.

If we had maintained the indicated separation between languages (1), (2), and (3), etc., the process [[a+b+c] buf (Σ*)in] + [[a+b+c] buf(Σ*)out] would have separated into three.

Now suppose that the designer is quite confident about his design for TH, but concerned about PS. At this point he can begin to concentrate on the characterization of PS. Based on obvious dichotomies in the languages as matched by antecedents of PS, he splits (12) into [a+b+c][0+1]()[0+1] [a+b+c] and [a+b+c][0+1](ΣΣ*)[0+1][a+b+c], and (15) into [processor-system + processor + acknowledge] and [may-send + may-not-send]. Intra-system arcs must be re-computed, because a node in each basic process is being split, and so each basic process could split. Outgoing inter-system arcs cannot affect the basic process structure, and so we omit them in the fps shown in Figure 19. Since the previous step of the analysis, one of the basic processes has split into two. The other experienced an internal split from one module into two, but remained a single basic process. Just to show how it looks, we give the canonical fps for this system in Figure 20.

Admittedly, the process structure of PS is not very illuminating. This is at least partly because PS is already excellent from a structured programming standpoint, and so

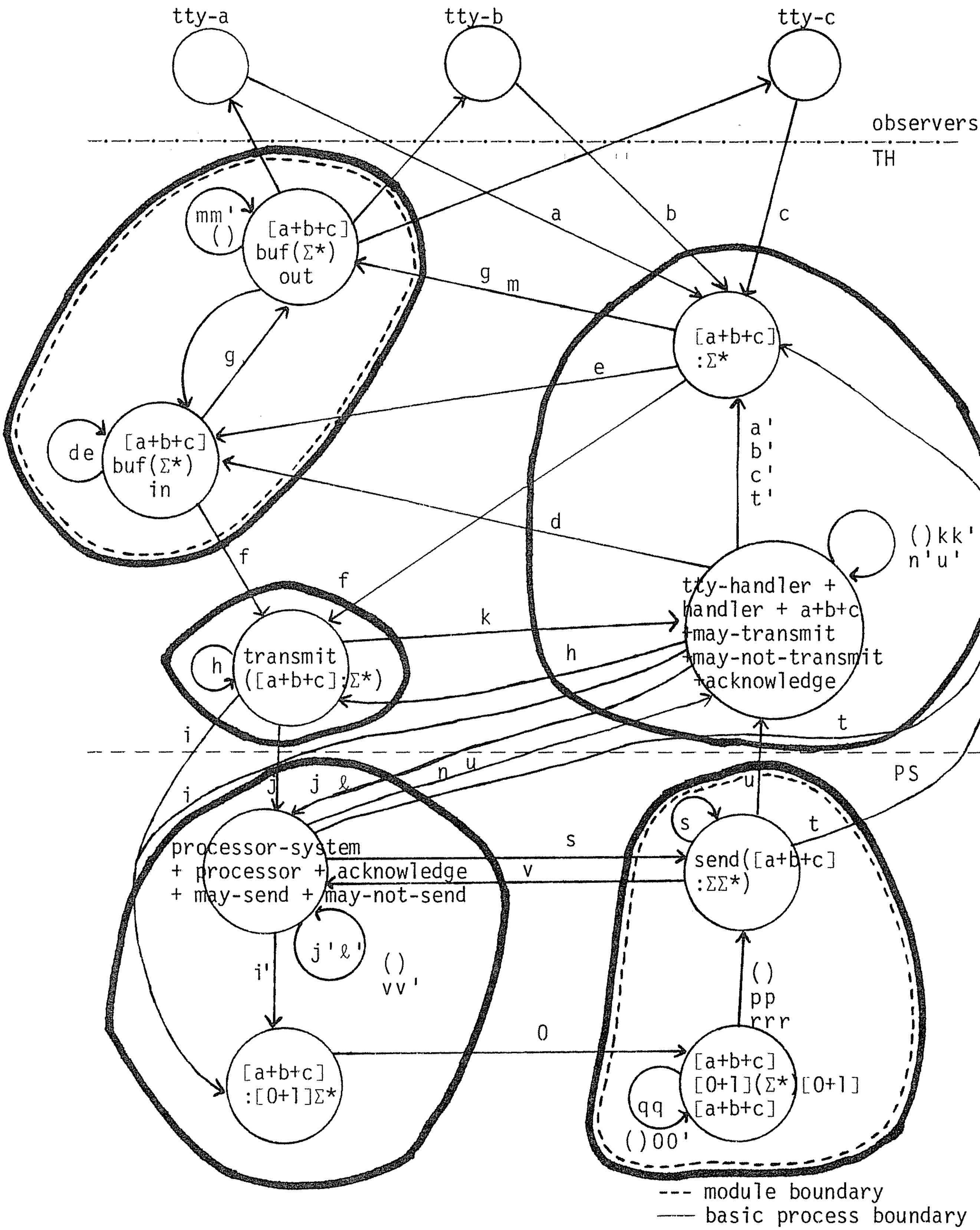

tty-a
tty-b
tty-c
observers
TH
mm'
()
[a+b+c]
buf(Σ*)
out
a
b
c
g
m
g
[a+b+c]
:Σ*
e
de
[a+b+c]
buf(Σ*)
in
a'
b'
c'
t'
d
()kk'
n'u'
f
f
tty-handler +
handler + a+b+c
+may-transmit
+may-not-transmit
+acknowledge
k
h
h
transmit
([a+b+c]:Σ*)
i
t
PS
i
j
j ℓ
n u
u
processor-system
+ processor + acknowledge
+ may-send + may-not-send
s
v
s
t
send([a+b+c]
:ΣΣ*)
j'ℓ'
()
vv'
i'
()
pp
rrr
0
[a+b+c]
:[0+1]Σ*
[a+b+c]
[0+1](Σ*)[0+1]
[a+b+c]
qq
()00'
--- module boundary
—— basic process boundary

FIGURE 18

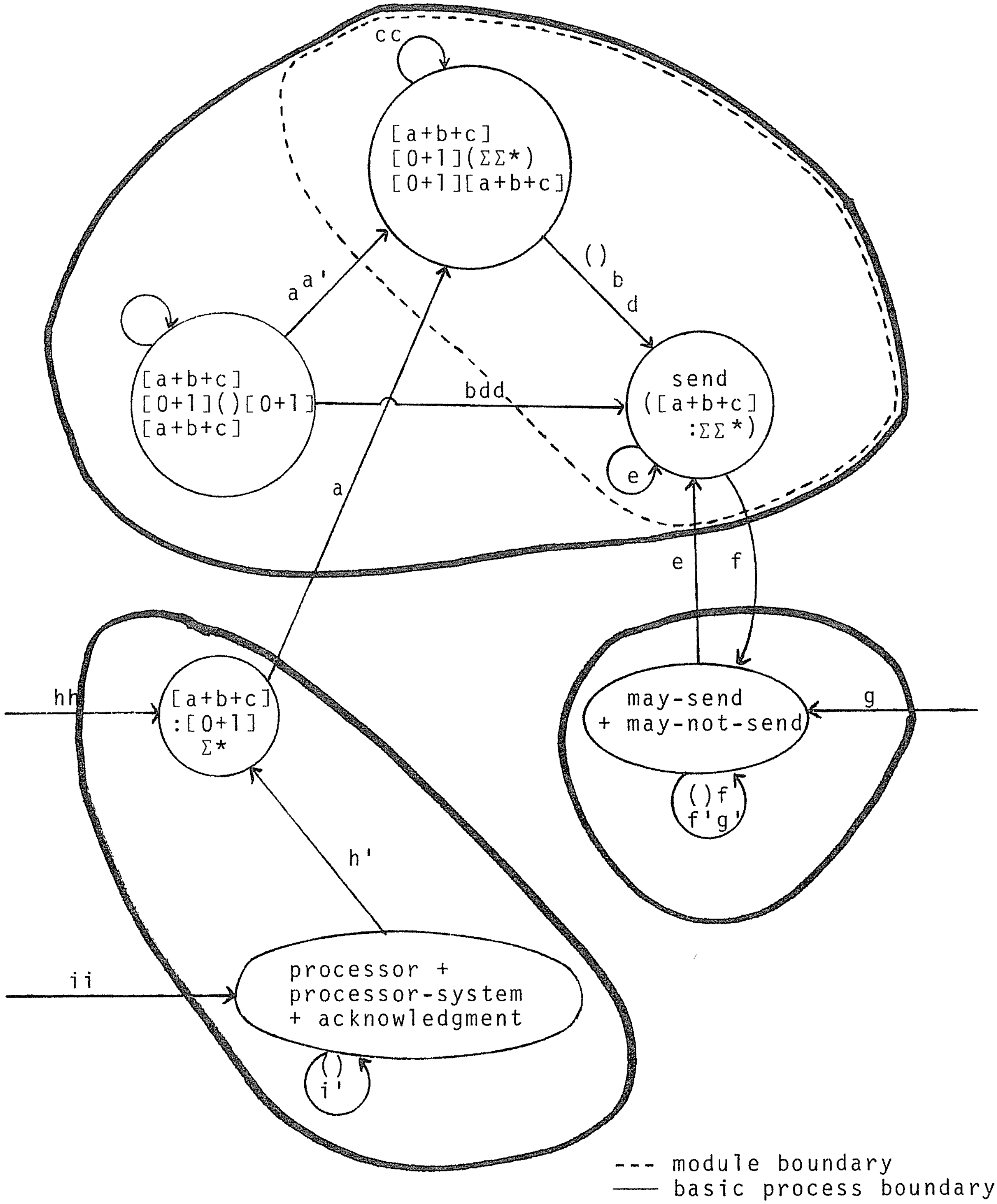

cc
[a+b+c]
[0+1](ΣΣ*)
[0+1][a+b+c]
()
b
d
aa'
[a+b+c]
[0+1]()[0+1]
[a+b+c]
bdd
send
([a+b+c]
:ΣΣ*)
e
a
e
f
hh
[a+b+c]
:[0+1]
Σ*
may-send
+ may-not-send
g
()f
f'g'
h'
ii
processor +
processor-system
+ acknowledgment
()
i'
--- module boundary
—— basic process boundary

FIGURE 19

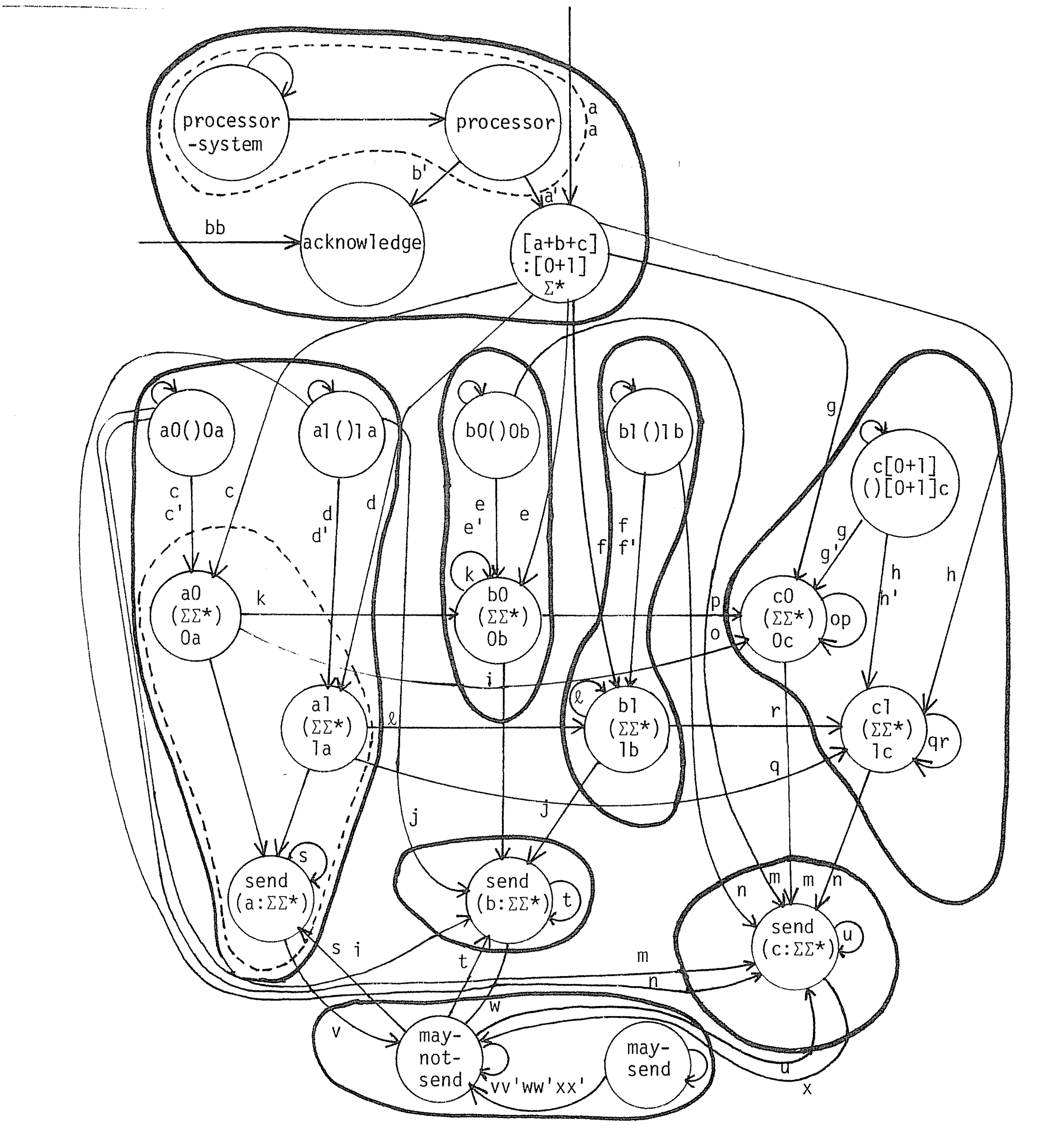


FIGURE 20

- - - module boundary
basic process boundary

our techniques can offer little to improve it. Imagine that the two processors were explicit and not hidden in RPR's. Then each one would engender a number of nodes linked by 1-arcs. Our techniques could be used to show that the nodes belonged to two distinct processes separate from the rest of the system; in a more sophisticated form they might also be used to prove that these processes had no synchronous relation to the rest of the system, and thus could be hidden in RPR's!

Even as far as it is developed, our theory of process structuring has three uses: (1) it can be used by the designer as a diagnostic check on his notions of which states interact, which form isolated localities, and which can have side effects on others, (2) it can serve as a precise basis for factoring the design itself and the problem of verifying it, and (3) it can indicate potential parallelism and other opportunities for implementation-independent optimizations. As mentioned in the preceding paragraph, extension of these techniques may make it possible to carry out the optimizations automatically, and prove that the transformed systems are equivalent to the originals (this is discussed at length in [2]).

Since the use of these techniques does require that systems designs be expressed in our formal definition universe, eventually designers may work with syntactic process structure already in mind. Then the definitions and their implications may evolve into principles of good design analogous to, but more precise than, the ideals of structured programming.

VI. CONCLUSION

Although we have presented some aspects of a procedure for design analysis in great detail, it is the technique involved that we wish to emphasize. We feel that the four concepts characterizing our procedure are essential to any successful attempt to analyze large-scale system designs. To summarize:

(1) The system design must be well-defined (requiring the rigor of a formal system in which to express it), and it must be possible to derive an abstraction of it, algorithmically, from the system representation. It is commonly understood that simplifying abstractions are necessary to the analysis of large designs; it is not so commonly understood that it is impractical to expect people to derive the abstractions--for this is an enormous, error-prone job. It is also very creative, in its general form, which is why we must impose sufficient structure to make automation possible.

(2) Assertions must be provable from the abstraction of the design. This requires (a) proof that an assertion about the abstraction is a valid statement about the object it came from, and (b) that the structures of interest are recognizable in the syntax of the abstraction. If these conditions are not met, analysis of the abstraction cannot be automated.

(3) Analysis must be an iterative, top-down procedure. No matter how much time and money are available for computing, there is a system design whose analysis will exhaust these resources, because exponential growth is intrinsic to this kind of computation. The only way to outwit the situation is to arrange it so that when you buy as much information as you can afford, you can get an overview instead of a detailed fragment.

(4) It should be possible to increase computational efficiency by placing effective restrictions on the objects to be analyzed. There is always a trade-off between complexity and verifiability, and this enables the designer to use it to his advantage.

In [3] we compared our formal definition universe, and the fps abstraction of it, to other models of computation. We found the other models unsuited to our purposes because of one or more of these shortcomings: only capable of representing a single synchronous process, the abstraction is not algorithmically derived from a well-defined object, the abstraction is special-purpose, or the abstraction is not susceptible to syntactic analysis. We could now add the criticism that many of these abstractions are strictly one-level descriptions, so that hierarchical analysis would be difficult.

REFERENCES

[1] Fitzwater, D. R., and Smith, Pamela Z. "A Formal Definition Universe for Complexes of Interacting Digital Systems," Computer Sciences Technical Report #184, University of Wisconsin, Madison, Wisconsin, 1973.

[2] Smith, Pamela Z., and Fitzwater, D. R. "A Concept of Equivalence Between Formally Defined Complexes of Interacting Digital Systems," Computer Sciences Technical Report #213, University of Wisconsin, Madison, Wisconsin, 1974.

[3] Smith, Pamela Z., and Fitzwater, D. R. "Finite Process Structures," Computer Sciences Technical Report #216, University of Wisconsin, Madison, Wisconsin, 1974.

[4] Fitzwater, D. R. "Effective Definition and Representation of Digital Processes and Systems," Computer Sciences Technical Report to be published.

[5] McNaughton, Robert, and Papert, Seymour. <u>Counter-Free Automata</u>. Cambridge, Massachusetts: The M.I.T. Press, Research Monograph No. 65, 1971.