WIS-CS-74-216
COMPUTER SCIENCES DEPARTMENT
The University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

Received June 21, 1974

FINITE PROCESS STRUCTURES

by

Pamela Z. Smith
and
D. R. Fitzwater

Computer Sciences Technical Report #216

August 1974

FINITE PROCESS STRUCTURES

by

Pamela Z. Smith
and
D. R. Fitzwater

ABSTRACT

A finite graph model is defined to describe all the computations, finite or infinite, generated by a formally defined complex of interacting digital systems. The graph, called a finite process structure, is an abstraction which can be formulated and computed directly from system representations. Finite process structures have properties making them excellent tools for design analysis; in particular, the infinite variety of graphs describing the processes of a single system at all levels of detail arranges itself into a lattice--so it is possible to find desired characterizations algorithmically. Thus top-down hierarchical analysis of systems becomes a plausible goal. Within this context, a semantic theory of process structuring is initiated: a process definition is given, and compared to state-of-the-art definitions of processes.

FINITE PROCESS STRUCTURES

I. INTRODUCTION

This report is a continuation of the research presented in [8], [17], and [25] on the properties of formally defined complexes of interacting digital systems.

Our purpose is to analyze system complexes--algorithmically, and without recourse to simulation or human interpretation of their computations--in the hopes that an understanding of their structures will lead us to diagnostic information, optimizations, and tools for design automation. Most of our results are based on finite process structures, which are functional models of the computations generated by system complexes. In this report we will define them and discuss the properties that make them so useful.

A finite process structure is a finite, directed graph which describes all the computations, finite or infinite, that can be generated by a system or a complex of interacting systems. It is an abstraction which can be formulated and computed directly from system representations. Furthermore, the level of abstraction is completely flexible, so that the analysis can be tailored to the problem it is to solve.

We naturally turn our attention to processes, because they are the basic structural units of computation. Having a finite process structure for a system complex allows us to recognize the processes in it, using a purely syntactic definition of process that has been developed specifically within this context. The definition is substantially the same, in spirit and content, as the state-of-the-art definition of process (although we will make small claims to superiority). We suggest that ours is more useful because it is representation-dependent (this should not be a restriction because the representation medium itself is abstract and very general) and can be applied algorithmically; this means that it can be used to find processes, besides simply verifying that a recognized structure is a process.

Future research will extend this syntactic theory of process structuring to definitions of specific kinds of process and sub-process structures, in the hopes of exploiting their properties. It will also include techniques for the top-down hierarchical analysis of systems--because we feel that this approach is the only one which offers any significant hope of taming the sheer computational complexity of system analysis. The ultimate justification of this work, of course, must come from examples demonstrating that the process structure we deduce from a system complex is intuitively reasonable, and contributes to the solution of practical design problems.

It might be interesting to look at examples of other research with some of the same objectives as ours. The work of Gilbert and Chandler [9] supplied important ideas for [17]; their graph models of systems are much more like Johnson's system state graphs than our finite process structures, but the idea of capturing complete information about particular aspects of computations in finite graphs is the same. Gilbert and Chandler assume a known and well-defined process structure (with special properties) from which they take the abstraction of the system that their analysis is based on. Here is where our work differs: we also compute the abstraction of the system representation the analysis will be using. That is important in itself, but it also enables us to discover and work with a much wider variety of process and system structures than Gilbert and Chandler's technique.

We share almost all of the goals of the LOGOS project ([10], [12], [5], [24], [11]), an attempt to develop a complete on-line system for integrated hardware and software design. In particular, we agree that there must be a representation system which is convenient for designers, capable of expressing structures at all levels of detail, and susceptible to analysis and checking without the difficulty and expense of simulation. The representation must facilitate proofs. The representation chosen for LOGOS is rather complicated and non-uniform, however, and so it is difficult to see how it will be possible to verify assertions about designs--the designers seem to be hoping that progress with Petri nets, formal program proving, etc. will solve the problems of global design analysis.

We feel that these problems, especially computational complexity, will not be overcome so easily. Our representation is simple and uniform, and the finite process structure abstraction of it can be used directly in proofs. Both in the representation system which is part of the formal definition universe, <u>and</u> in the finite process structure modeling of systems, it is possible to approach designs iteratively, looking at increasingly detailed specifications. Even so, we believe that computational complexity is the ultimate problem in the field of automated systems design: it will not be alleviated except by revolutionary methods, and certainly not by waiting for new techniques to apply to a representation developed without regard to it.

One relevant difference between our goals and the LOGOS goals is that LOGOS has been conceived within contemporary ideas of how systems should be structured and how they are eventually implemented, and these ideas have greatly influenced its shape. We wish to provide tools that will still be useful if the computing environment changes, and so our representation and concepts of structure are less dependent on present computer architecture.

II. REGULAR LANGUAGE CHARACTERIZATIONS

1. State Structure

It has been found that, because of the design of the formal definition universe, regular languages give excellent descriptions of the structure of process states. This is because the set of all strings that a given antecedent can match is a regular language; likewise, the set of all strings generated by a consequent can be closely approximated by a regular language which contains it.

We value regular language characterizations of process state structures because they are natural and convenient, but most of all because they are finite representations of infinite classes of process states. So when we analyze systems, we will use regular languages as equivalence classes for the states occurring during computation: this is the source of our ability to give a finite representation for an infinite computation or set of computations.

2. The R-Model Universe

In subsequent sections we will apply the productions of systems to these regular language equivalence classes to get successor languages, each representing the result of applying a production to all members of the predecessor language (if the production has multiple antecedents, there will be a predecessor language for each antecedent) and taking the union of the strings generated. Since the application of productions to regular languages is certainly not part of the formal definition universe, what is the basis for such calculations?

We will now call the primitive automaton of the formal definition universe, which interprets systems whose states are represented as finite sets of strings, the F (for finite) -model interpreter. This is to distinguish it from the R (for regular language) -model interpreter running in the R-model universe as defined by Johnson [17]. The R-model interpreter runs systems having finite sets of regular languages as their representations of state information. Johnson proved that all F-model computations are contained in their corresponding R-model computations.

There are three cases in which the R-model loses precision, meaning that the result of applying a production to a regular language is a regular language larger than the union of the F-model results of applying the production to each string in the predecessor language.

Case I: An antecedent of the production contains an RPR. The RPR pattern matches a regular language which every RPR result must belong to. Even though the RPR results may all be contained in a much smaller language, the R-model generates consequent languages as if the pattern language were the smallest language containing the RPR results. There are two reasons for this: one is that analysis within RPR's appears to be more trouble than it is worth, and the other is that we view RPR's as belonging to a lower level of the computational hierarchy which can be analyzed separately, if necessary. Then the results of RPR analysis would be plugged into analysis at the higher level.

Case II: The consequent pattern contains a repeated variable, for example "A $\$_1$ B $\$_2$ C $\$_1$ D." In these cases the language consisting of all consequents need not be regular, and so the best we can do is approximate it with a containing regular language. In the case of the consequent shown, the consequent language is context-free. If "$\$_1$" in the antecedent had matched the regular language R_1 and "$\$_2$" had matched R_2, then the containing regular language would be A R_1 B R_2 C R_1 D.

Case III: The antecedent contains two or more variables, all of which are used to construct the consequent, and in the languages associated with the antecedent variables when the antecedent is matched to a regular language, certain sublanguages of one variable's language only appear in the regular process state language with certain sublanguages of the other variables' languages. The loss of precision is not really surprising because the information being lost is context-sensitive, exactly what you would expect to lose in a regular language characterization. Unfortunately, it can occur when the regular language is as simple as the single string "ab." If the production is

$$\$\ \$ \rightarrow \$_1\ \$_2\ ,$$

$\$_1$ matches the language $\{\lambda,a,ab\}$ and $\$_2$ matches the language $\{ab,b,\lambda\}$, so the consequent language generated is $\{\lambda,a,b,ab,aab,abb,abab\}$, even though the F-model result is simply the string "ab." Occurrences of this problem are curtailed, although not eliminated, if antecedents can match single strings in only one way. In Section II.4 we will mention a sufficient condition ensuring this.

The relationship of the R-model interpreter to the F-model is summarized in Figure 1.

3. Computational Considerations

Case II might be interpreted as an indication that we could get better results if we used more powerful languages for our analyses. It is well to remember that the regular languages are the largest known class of languages for which there are algorithms for determining the equality, containment, and emptiness of languages, or for performing such operations as union, intersection, and difference

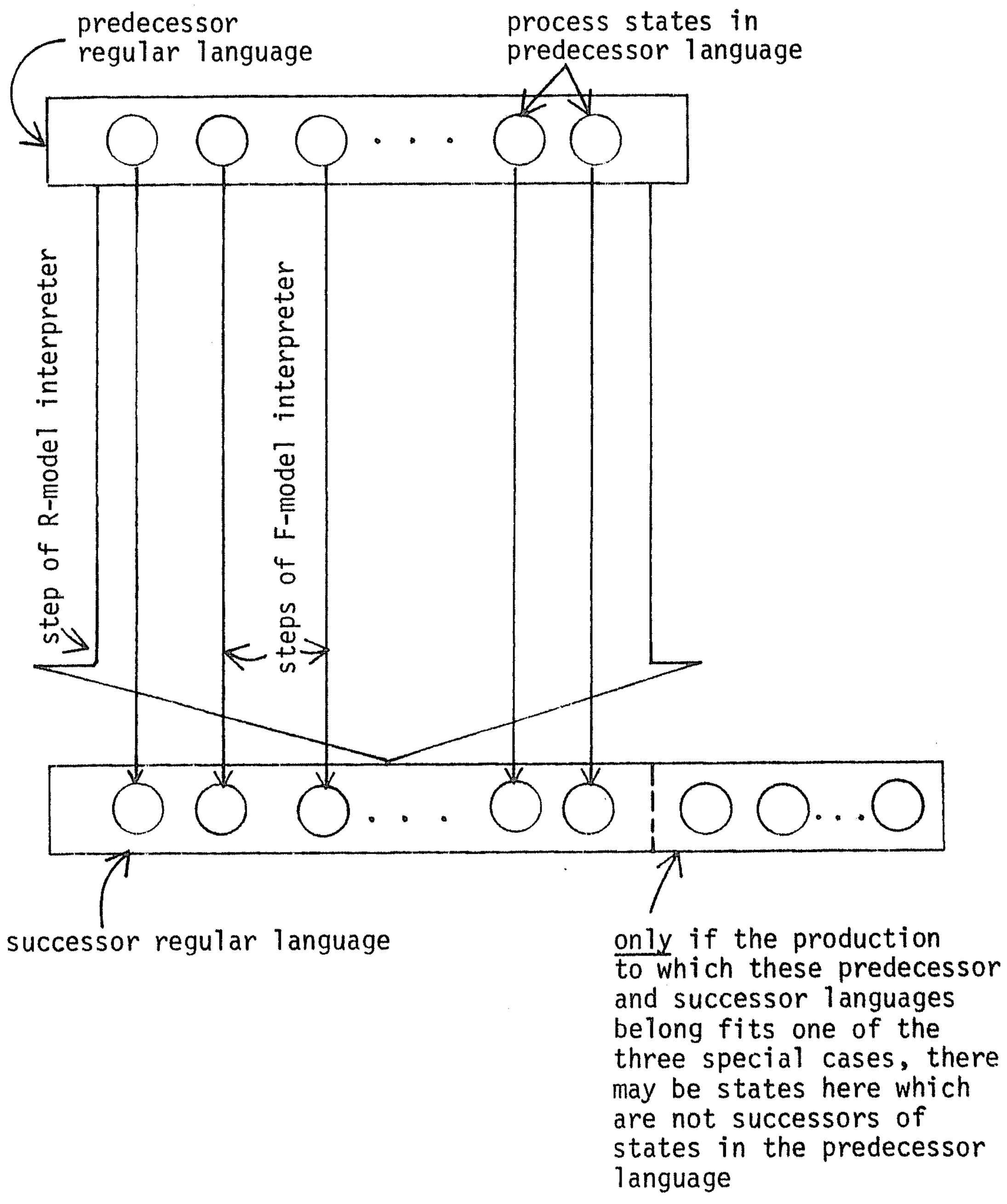

predecessor regular language
process states in predecessor language
step of R-model interpreter
steps of F-model interpreter
steps
successor regular language
only if the production to which these predecessor and successor languages belong fits one of the three special cases, there may be states here which are not successors of states in the predecessor language

Figure 1

on pairs of languages, and which are closed under union, intersection, and complementation. We need these manipulations of languages to manipulate finite process structures, and thereby the system complexes they model, and so it would be very foolish to place ourselves under the handicap of not always being able to perform them.

From a practical standpoint, computations that are extremely lengthy might as well be impossible, and even regular language problems can involve serious complexities. The formal definition universe is particularly adaptable, however, to schemes for reducing complexity.

The languages matched by antecedents and generated by consequents are a subclass of the regular languages we have named pattern languages. Their properties will be discussed in the next section. Pattern languages will occur naturally in our analysis, especially since they are closed under some of the most common operations on languages. Although no work has been done on the specific computational complexity of problems for pattern languages, the results for other subclasses of the regular languages [27] suggest that the improvement over regular languages will be substantial.

It would have been possible to construct the formal definition universe so that the languages matched by antecedents and generated by consequents were regular languages, or some other subset of them besides pattern languages--the domain of systems definable would not be altered, but only the convenience with which they could be defined. We mention this because we will be suggesting that the formal definition universe be altered to achieve certain computational goals. It could be extended so that antecedent and consequent languages were non-counting languages (a subclass of the regular languages about which many interesting theorems can be proved, see [22]), or restricted to the delimited pattern languages (see next section). There is nothing sacred about the way antecedents and consequents were originally defined, and so we think that these alternatives deserve serious consideration.

4. Properties of Pattern Languages

In this section we will define a subclass of the regular languages called the class of pattern languages, and show that the language matched by any antecedent or generated by any consequent is a pattern language. We will show that the class of pattern languages is closed under union and intersection, prove that the closure of the pattern languages under complementation is contained in the non-counting languages, and discuss the implications of pattern language properties on the computational complexity of manipulations of fps's.

Let $\sum = \{x_1, x_2, \ldots, x_n\}$ be an ordered set of n characters, λ be the null string, and ϕ be the empty set.

Regular expressions (re's) over $\sum$ and the regular languages (rl's) they denote are defined recursively as follows:

(1) ϕ is an re denoting the rl ϕ ;

(2) λ is an re denoting the rl $\{\lambda\}$;

(3) x_i is an re denoting the rl $\{x_i\}$, $1 \leq i \leq n$;

(4) if p and q are re's denoting rl's P and Q respectively,

then

(a) $(p + q)$ is an re denoting $P \cup Q$,

(b) (pq) is an re denoting PQ , and

(c) $(p)^*$ is an re denoting P^* .

The definition of pattern languages uses subsets X_i of $\sum$, where the index refers to an ordering of $P(\sum)$ (the power set of $\sum$):

$X_0 = \phi$

$X_i = \{x_i\}$, $1 \leq i \leq n$

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

$X_{2^n-1} = \sum$

Pattern expressions (pe's) over $\sum$ and the pattern languages (pl's) they denote are defined recursively as follows:

(1) ϕ is a pe denoting the pl ϕ ;
(2) λ is a pe denoting the pl $\{\lambda\}$;
(3) X_i is a pe denoting the set X_i, $1 \leq i \leq 2^n-1$;
(4) X_i^* is a pe denoting the set X_i^*, $1 \leq i \leq 2^n-1$;
(5) if p and q are pe's denoting pl's P and Q respectively, then
  (a) $(p + q)$ is a pe denoting $P \cup Q$, and
  (b) (pq) is a pe denoting PQ.

Theorem: The pattern languages over $\sum$ are properly contained in the regular languages over $\sum$.

Proof: Let X_i be the member of $\mathcal{P}(\sum)$ which is $\{y_1, y_2, \ldots, y_m\}$. Containment holds because pe X_i can be expressed by the re $(y_1 + y_2 + \ldots + y_m)$, pe X_i^* can be expressed by the re $(y_1 + y_2 + \ldots + y_m)^*$, and all other pe's or parts of them are also re's. Proper containment holds because rl $(x_1x_2)^*$ is not a pl (a pe for it would have to be infinite: $\lambda + X_1X_2 + X_1X_2X_1X_2 + X_1X_2X_1X_2X_1X_2 + \ldots$).

Null Reduction Procedure: If p and q are fully-parenthesized pe's, make these substitutions wherever applicable. u and v are any symbol strings either or both of which may be absent in any application.

(1) if $p = u(\phi q)v$ or $u(q\phi)v$, then $p := \phi$;
(2) if $p = u(\phi + q)v$ or $u(q + \phi)v$, then $p := uqv$;
(3) if $p = u(\phi + \phi)v$, then $p := \phi$;
(4) if $p = u(\lambda + \lambda)v$, then $p := uv$;
(5) if $p = u(\lambda q)v$ or $u(q\lambda)v$, then $p := uqv$.

Theorem: The null reduction procedure is an algorithm that forms an equivalent fully-parenthesized pe for the same pl in which ϕ never appears in a product or sum and λ never appears in a product.

Proof: By inspection.

We defined pattern expressions in such a way as to show their similarity to regular expressions, but this leaves them unnecessarily complex. They are fully parenthesized and may have redundant ϕ and λ symbols, all of which can be removed. To arrive at this reduced form, we apply first the null reduction procedure, and then the parenthesis elimination procedure.

The following algorithm is based on the assumption that in the evaluation of pe's concatenations (products) are performed before unions (sums). The result of applying the null reduction procedure and the following parenthesis elimination procedure to a pattern expression is to produce an equivalent reduced pattern expression (rpe). Reduced pattern expressions differ from the originally defined pe's in three ways:

(1) ϕ appears only when it is the total rpe;
(2) λ appears only as a whole term in a sum;
(3) no parentheses appear, and the convention is that products are evaluated before sums.

From now on, the term "pattern expression" may mean an original pe, an rpe, or a hybrid expression using elements of both forms.

Parenthesis Elimination Procedure: If p, q, and r are pe's, make these substitutions wherever applicable. u and v are any symbol strings, either or both of which may be absent in any application.

(1) if $p = u(qr)v$, then $p := uqrv$;
(2) if $p = u(q + r)v$, then $p := uqv + urv$;
(3) eliminate redundant terms.

Theorem: The null reduction procedure followed by the parenthesis elimination procedure produces an equivalent rpe for the same pl which has the form $P = \sum_i \prod_j P_{ij}$ where

(1) $P_{ij} \in \{\phi, \lambda, X_k, X_k^*\}$,
(2) P_{ij} can be λ only when the maximum value of j is 1,
(3) P_{ij} can be ϕ only when the maximum values of both i and j are 1, and
(4) $P_i = \prod_j P_{ij} \neq P_k = \prod_\ell P_{k\ell}$ unless $i = k$.

Proof: By inspection.

The next thing we wish to show is that antecedent and consequent languages (the languages matched or generated by them, respectively) are pattern languages, and pattern languages can be made into antecedent and consequent languages.

Theorem: (a) The language matched by any antecedent is a pattern language;
(b) the language generated by any consequent is a pattern language;
(c) any non-empty pattern language which can be expressed by an rpe with a single term is the language matched by some antecedent;
(d) any non-empty pattern language which can be expressed by an rpe with a single term is the language generated by some consequent.

Proof: The proofs of all four parts will be by construction over Σ, where Σ is the set of all possible characters appearing in process states.

(a) When the syntax of an SR reveals a blank space where an antecedent belongs, the pl associated with that antecedent is λ. All other antecedents are symbol strings, and an rpe for the associated pl is produced by substituting for the antecedent symbols according to the following table. RPR's are first replaced by their axiom patterns (but variables in the pattern are still associated with the appropriate χ-sets). Because each variable has its own associated vocabulary, in the rest of the proof we will use X_j to denote that subset of Σ which is the appropriate vocabulary for the variable being discussed.

antecedent symbol	pe symbol
character A	X_k where $X_k = \{A\}$
variable $\overline{A}$	X_k where $X_k = \{x \in X_j \mid x \neq A\}$
variable $\$$	X_j^*
variable $\$_i$	$\underbrace{X_j\ X_j\ \cdots\cdots\ X_j}_{i}$

Also, if $\$_i$ appears in the antecedent, there must be separate pe terms created substituting strings of i-1, i-2, ..., 1, and 0 X_j's for the $\$_i$ in the antecedent. Since terms must be separated like this for each $\$_i$, if there are n subscripted $\$$'s in the antecedent, and the wth one has subscript t_w, then the number of terms in the resultant pe is

$$\prod_{w=1}^{n} (t_w+1).$$

(b)

consequent symbol	pe symbol
character A	X_k where $X_k = \{A\}$
variable $\overline{A}_i$	X_k where $X_k = \{x \in X_j \mid x \neq A\}$
variable $\$_i$	if the corresponding antecedent $\$$ was unsubscripted, X_j^*; if the corresponding antecedent $\$$ was subscripted with t, $t+1$ different terms having strings of from 0 to t X_j's

(c) An antecedent can also be formed from a single-term pe by symbol substitution.

pe symbol	antecedent symbol
λ	blank space
X_i	character x_i if $1 \leq i \leq n$, otherwise $\{\overline{\Delta}\underline{\chi}\underline{:}\langle X_i\rangle\ \underline{\pi}\underline{:}\}$ where $\langle X_i\rangle$ is a metanotation for $y_1 y_2 y_3 \ldots y_n$, $y_j \in X_i$, $1 \leq j \leq n$
X_i^*	$\{\$\underline{\chi}\underline{:}\langle X_i\rangle\ \underline{\pi}\underline{:}\}$

(d)

pe symbol	consequent symbol
λ	blank space
X_i	character x_i if $1 \leq i \leq n$, otherwise $\overline{\Delta}_k$ where k is the index of an antecedent variable having the vocabulary X_i
X_i^*	$\$_k$ where k is the index of an antecedent variable having the vocabulary X_i .

Q.E.D.

Theorem: For any set $\sum$, the class of pattern languages over $\sum$ is closed under union and intersection.

Proof:

Part I: The class is closed under union.

If p and q are pe's for pl's P and Q, then a pe for $P \cup Q$ is $(p + q)$, so $P \cup Q$ must be a pl.

Part II: The class is closed under intersection.

Let $P = \underset{i}{\cup} P_i$ and $Q = \underset{k}{\cup} Q_k$ be any two pattern languages where P_i and Q_k can be expressed by the rpe's $\underset{j}{\Pi} P_{ij}$ and $\underset{\ell}{\Pi} Q_{k\ell}$, respectively. If either P or Q is ϕ, then $P \cap Q$ is ϕ, a pl.

$$\text{Otherwise,}\quad P \cap Q = (\underset{i}{\cup} P_i) \cap (\underset{k}{\cup} Q_k)$$

$$= \underset{i,k}{\cup} (P_i \cap Q_k).$$

Since we know that pl's are closed under union, it is only necessary to show that $P_i \cap Q_k$ is a pl by constructing a pe for $\underset{j}{\Pi} P_{ij} \cap \underset{\ell}{\Pi} Q_{k\ell}$.

The construction of $\underset{j}{\Pi} P_{ij} \cap \underset{\ell}{\Pi} Q_{k\ell}$ is recursive, according to the following table. Assume $\underset{j}{\Pi} P_{ij}$ or $\underset{\ell}{\Pi} Q_{k\ell}$ to be extended with λ's, if necessary, so that $j = \ell$. Since we are dealing with non-empty rpe's, we know that $\underset{j}{\Pi} P_{ij}$ and $\underset{\ell}{\Pi} Q_{k\ell}$ have no parentheses or ϕ's, and only trailing λ's (unless the whole term is λ).

We use the notation $X_{r\cap s}$ for $X_t = \{x \in \Sigma \mid x \in X_r \ \& \ x \in X_s\}$. Recall that if $X_r \cap X_s = X_0 = \phi$, ϕ concatenated with anything reduces to ϕ, and ϕ^* reduces to λ.

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

It is easy to show that the result is a pe and that it is contained in $P_i \cap Q_k$. The proof that $P_i \cap Q_k$ is contained in the result proceeds by case analysis, and is lengthy.

Q.E.D.

We know that the pattern languages are not closed under complementation. A conterexample is $\overline{(01)^*}$. This is a pl, as can be seen from this pe for it:

$$X_{2^n-1}^*(X_0X_0+X_1X_1)X_{2^n-1}^* + X_1X_{2^n-1}^* + X_{2^n-1}X_0 ,$$

but its complement, $(01)^*$, is definitely not a pl.

Theorem: The pattern languages and their closure under complementation are both contained in the non-counting languages.

Proof: One way of characterizing the class of non-counting languages is that it is the class described by star-free extended regular expressions, i.e., regular expressions using the union, intersection, concatenation, and complementation operators, but not the star. We will show that any pl can be expressed in a star-free extended regular expression. It will then be obvious that the closure of the pl's under complementation is contained in the non-counting languages, because if q is a star-free extended regular expression, so is $\overline{q}$.

In the definition of a pattern expression, the only components which are not also valid in star-free extended regular expressions are X_i and X_i^*. But

$X_i = (x_{i1}+x_{i2}+\ldots+x_{im})$, where $\{x_{i1},x_{i2},\ldots,x_{im}\} = \{x_j | x_j \in X_i\}$,

and $X_i^* = \overline{\overline{\phi} X_{\overline{i}} \overline{\phi}}$, where $X_{\overline{i}}$ is a shorthand for $\{x_j | x_j \notin X_i\}$. Thus any pe can be converted to a star-free extended regular expression, and so any pl is also a non-counting language.

Q.E.D.

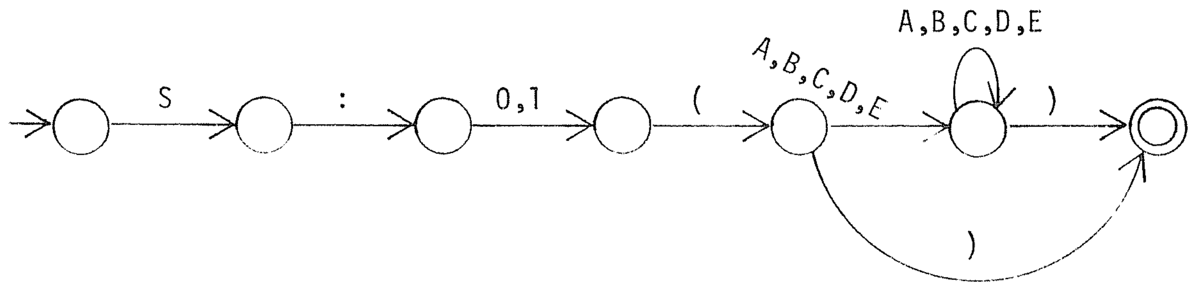
An interesting consequence of the proof that the pl's are closed under intersection is that a pattern expression for the intersection of two pattern languages can be computed by a simple manipulation of the pe's representing them. Such a string operation is vastly more efficient than the algorithm to do the same for regular languages, which requires intermediate construction of finite state machines. We might expect the same to be true of other operations or algorithms on pattern languages.

For those operations requiring the use of finite state machines, the machines accepting pattern languages have a rather simple and distinctive structure, which is shown in the examples of Figures 2a and 2b. These machines are non-deterministic. Each state has an implicit transition to a dead-end non-accepting state on any input not otherwise causing a transition. Note that there are no loops with path lengths greater than one. This is suggestive of the crucial difference between regular languages and non-counting languages, that non-counting languages are expressable without stars.

Union and intersection are undoubtedly the most common operations we will be performing on fps languages. For instance, suppose we want to characterize the set of all strings generated by either a production whose consequent language is L_b or a production whose consequent language is L_c, that match the antecedent language L_a. It is the pattern language $(L_b \cup L_c) \cap L_a$. If we were interested in this language because we were optimizing the SR, and needed to add a production whose antecedent matched just that language, it could be done. In this case there might be a slightly different production for each term in the pe for $(L_b \cup L_c) \cap L_a$, because we only know for sure that we get an antecedent to correspond to a single pe term. Section VII.2 presents another very important application of intersecting pattern languages.

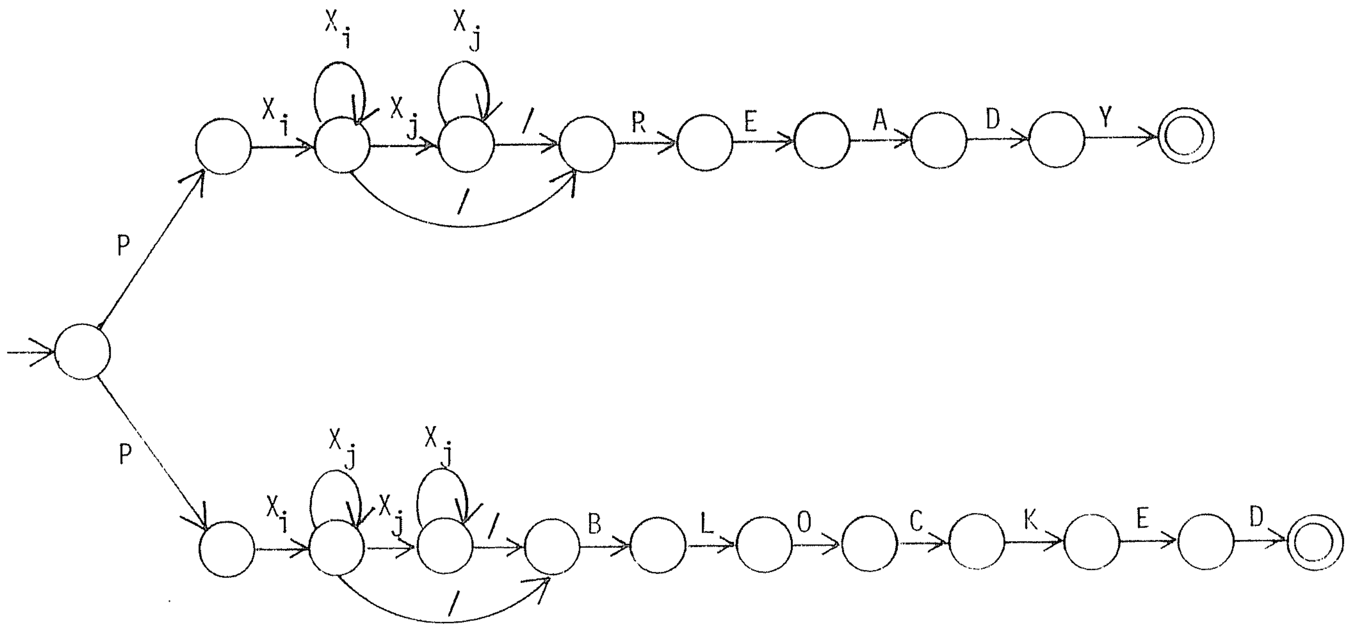
A problem that arises is that an algorithm may return a language $\overline{L_a}$ or $L_c - L_a$ which is not a pattern language. If the objective of the algorithm is to translate the resultant language into a transformation on the SR, it will not be possible to achieve its objective, because the language does not correspond to any antecedent or consequent (or finite union of them).

It is well to remember that we still have a choice in the matter: all forms of diagnostic analysis, and even many optimizing transformations



The automaton accepts the language S: X_i (X_j^*) where X_i = {0,1} and X_j = {A,B,C,D,E}.

Figure 2a



The automaton accepts the language [P X_i X_i^* X_j^*/READY] ∪ [P X_i X_i^* X_j^*/BLOCKED] where X_i = {A,B,E,0,1} and X_j = {0,1,2,3,4,5,6,7,8,9} .

Figure 2b

of system complexes, can be carried out without making fps nodes into antecedent and consequent languages. However, there are two prospective solutions. One of them is to extend the formal definition universe so that antecedent and consequent languages are non-counting languages. This would introduce additional computational complexity, though, and it is not yet certain that such a well-defined extension exists.

A more attractive solution is to restrict antecedents and consequents in SR's which are being analyzed for the purpose of transformation. An easily satisfied and easily checked rule will ensure that the antecedent and consequent languages are delimited pattern languages, a subclass of the pattern languages which is closed under union, intersection, and complementation. The properties of delimited pattern languages will be developed in [26]. This solution has the advantage that the delimited pattern languages are computationally quite superior to the pattern languages! Each language has a canonical form, and there are efficient string manipulations to perform all the useful language operations--they may even preserve the canonical form. Delimited antecedents have the highly desirable property that they can match strings in only one way. We expect that it will be highly advantageous for the designer to restrict himself to delimited antecedents and consequents, no matter what kind of algorithmic analysis he anticipates using.

III. FINITE PROCESS STRUCTURE LANGUAGES

1. Closure and Total Closure

We have already indicated that a finite process structure (fps) is a graph whose nodes are disjoint regular languages. The finite process structure language (fpsl) of an fps describing an isolated system is the union of its node languages. Even when we extend the fps definition to a complex of systems, the resulting fps is still, in many ways, a loosely connected set of single-system graphs--in particular, the nodes for one system can be chosen independently of the nodes for other systems, and so an fpsl is always associated with a single system.

There are four ways in which a process state can come to belong to the process state set of a system: the two "external" ones are (1) belonging to the initial σ, and (2) being accepted as a message from another system; the two "internal" means are (3) being generated as a consequent, and (4) being accepted as a message to this system that was also generated by it. These categories are relevant to two definitions we need.

Definition: A language is closed under a system if and only if as long as no process state not in the language is ever introduced into σ by external means, then no process state not in the language will ever be introduced into σ by internal means.

Closure is important because an fpsl for a system must be a language which is closed under that system. We make this requirement so that the fps will be well-defined, i.e., the result of applying a production of the system to a node language in the fps will be a language contained in some node or nodes of the fps. In [17] there is an algorithm for testing a system and a language for this property.

Definition: A language is totally closed under a system if and only if it is closed under the system and, in addition, if a process state not in the language were introduced into σ by external means, the fact of its presence would not be observable, nor could it have any effect on any subsequent computations of the system complex.

We say "the fact of its presence" because message acceptance always has the side effect of deleting a channel name from σ at the time of acceptance. Such negative side effects are usually ignored, for reasons that will be discussed in Section IV.3. The word "computation" is used in its standard sense, that of a sequence of states corresponding to an interpretation of a digital computing device.

2. The Total Language

The significance of total closure will become clear later; suffice it to say that it is a very useful property of fpsl's. In fact, we will almost always want to work with fpsl's that are totally closed under their systems. We will define (by specifying an algorithm for calculating it) a canonical language for a system called the total language. The total language is totally closed under its system, and any language it is contained in is also totally closed under the system; these properties guarantee its importance in our analysis techniques.

Definition: The total language of a system is the union of all the antecedent languages of all the productions in the system, all the consequent languages of productions having single consequents, and all the message languages generated by the system which can also be received by it.

To find these message languages we must look at, for each production in the system that sends a message, the (channel language, message language) pair defined by the pattern languages matched by its second and first consequents, respectively. Assume we have a list of these pairs (L_c, L_m), and that the union of the antecedent and consequent languages is L_t. Then if L_u is the union of all languages L_m such that (L_c, L_m) is a pair on the list and $L_c \cap L_t \neq \phi$, the total language is $L_t \cup L_u$.

Theorem: The total language of a system is totally closed under that system, as is any language in which the total language is contained.

Proof, Part I: The total language is totally closed under its system.

The total language includes, by construction, every process state that the system can generate as a consequent. It also includes every state the system can generate as a message and also accept: every state which can be generated by the system is considered a potential channel name except those which can only be introduced into σ as accepted messages (and therefore cannot act as channel names for the acceptance of other messages). Thus the total language is closed under the system.

Suppose a state not in the language were introduced into σ externally. It would not be observable because it would not match any antecedent in any production, and therefore could not cause a message to be sent. For the same reason, it could not cause the internal generation of any process state, either (i.e., act as a channel name on which a message is accepted) because (a) if it is an initial state, it is only in σ at the beginning of the first step, when σ'' is always empty, and (b) if it is an accepted message itself, the definition of the primitive automaton ensures that it cannot be used as a channel name. Thus, since this externally introduced state cannot participate in any of the four ways process states are introduced to a σ, it cannot have any effect on any subsequent computations of the system complex.

Proof, Part II: Any language in which the total language is contained is totally closed under its system.

Consider any language L_{t+} containing the total language. It is closed under the system because no state not in L_{t+} can be generated internally. Since $\overline{L_{t+}} \subset \overline{L_t}$, all the arguments that states in $\overline{L_t}$ can have no effect on computations are true for states in $\overline{L_{t+}}$.

Q.E.D.

The total language is not necessarily the smallest language which is totally closed under a system. This is because we added a L_m when its corresponding L_c had only a non-empty intersection with L_t --if L_c were divided into $L_c \cap L_t$ and $L_c - L_t$, and L_m were divided correspondingly, it could be that only a sublanguage of L_m needed to be added to L_t . This procedure, however, requires much more computation, and we judge that the additional precision is not worth the effort, unless it is specifically required.

IV. GRAPHS OF ISOLATED SYSTEMS

1. Single-Component Arcs

We will now define a finite process structure for a single system; in the next part these ideas will be extended to fuse single-system fps's into an fps for a system complex by adding representations of communications between systems and observers (see [25] for a thorough discussion of observing systems). As we have explained, the nodes of the fps are disjoint regular languages whose union is a finite process structure language--the graph is completely determined by the system representation and this choice of nodes.

Each directed arc is implicitly (or explicitly, if one desires to label the productions and then label the arcs accordingly) associated with a production (or two, but that comes later) of the system. If a production has only one antecedent and generates a consequent rather than sending a message, then all its associated arcs have a single component (they are called 1-arcs).

The 1-arc in Figure 3 has the meaning that the antecedent of the production associated with it can match a string in L_1, and when it does, the consequent generated can be in language L_2.

2. Multiple-Component Arcs

Productions which have n antecedents and generate single consequents are associated with arcs having n components, called n-arcs. Productions which have n antecedents and send messages are associated with multiple-component arcs called (n+m)-arcs.

N-arcs are basically the same as single-component arcs (which they include) except that they are associated with productions that

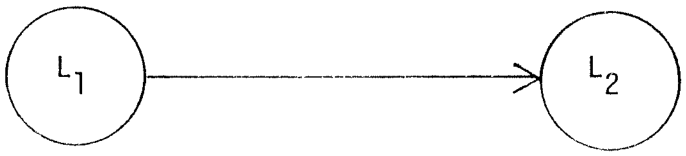


Figure 3

may have more than one antecedent, and there is a component for each antecedent. All the arc components $(n>1)$ are directed to the same destination node, and all bear a common unique label to show that they belong together. The production

$$\text{go } \underline{\text{and}} \text{ process \$ [\$] } \underline{\text{and}} \text{ data [\$]0} \rightarrow \text{permit } \$_1$$

could be associated with either of the 3-arcs in Figure 4.

When a system sends a message which it also accepts, here is what actually happens: in one σ there are process states matching all the antecedents of the production that sends the message, and process states in the same σ also match the antecedents of a production generating a consequent, which is to be the channel name on which the message is accepted. At the end of the step the message is sent and received instantaneously, the new σ is updated from σ'', and the message is then a process state in the new σ (while the channel name has disappeared). The effect is that all the process states in the preceding σ which matched either the antecedents of the message-sending production or the antecedents of the channel-generating production interacted to produce the message as a process state in the succeeding σ. It is this interaction that we need to model in the arcs associated with message-generating productions.

So if a message-generating production has n antecedents, and a consequent-generating production whose consequent language intersects the channel language has m antecedents, an fps may have an $(n+m)$-arc associated with these productions. The arc will have $n+m$ components, one for each antecedent involved, all pointing to the same destination node; the components will have a common unique label, but the labels on the components associated with the channel-generating production will be primed to distinguish them.

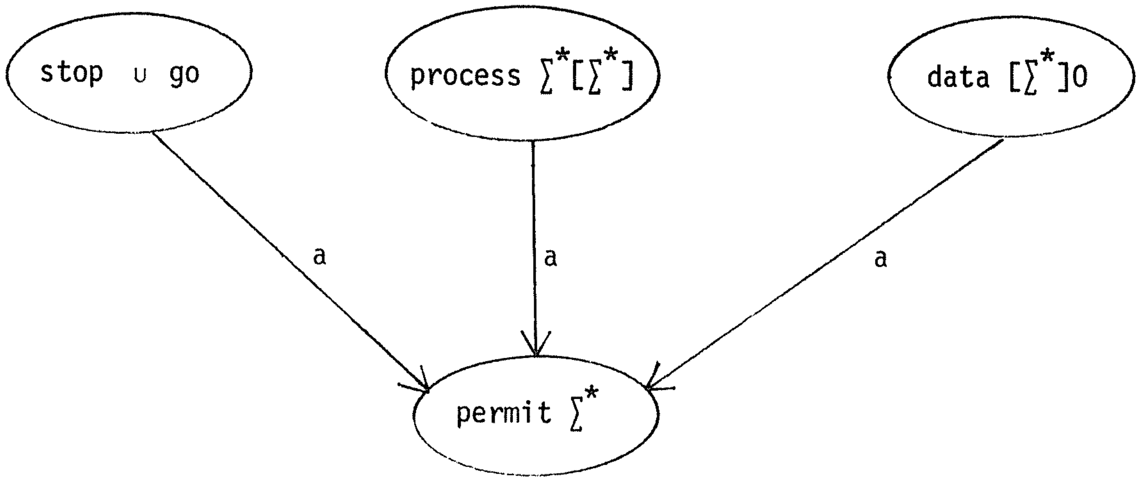


Figure 4a

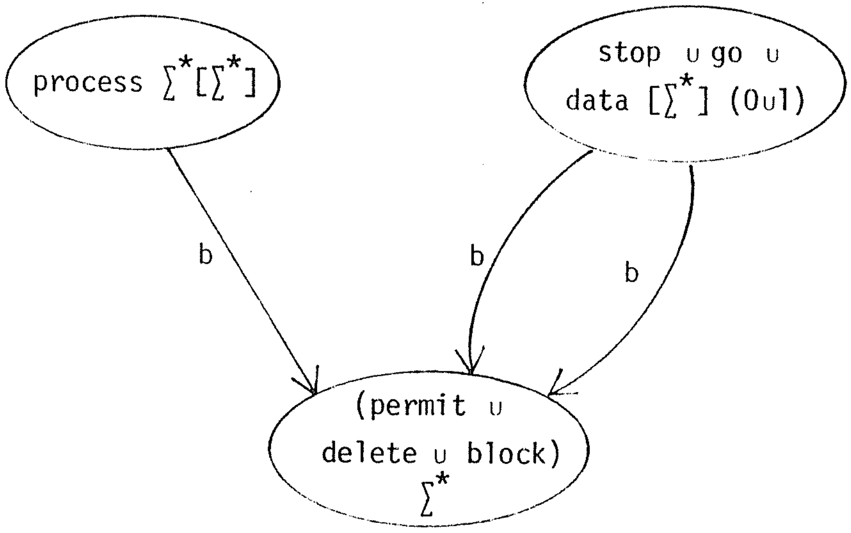

process $\sum^*[\sum^*]$
stop ∪ go ∪ data $[\sum^*]$ (0∪1)
b
b
b
(permit ∪ delete ∪ block) $\sum^*$

Figure 4b

For example, supoose a system sends a message to itself via the production:

$*\$* \rightarrow$ input phase $\rightarrow$ waiting phase

and the process state "waiting phase" can be generated by one of these two productions:

acknowledgement <u>and</u> output phase $\rightarrow$ waiting phase <u>or</u>
waiting phase $\rightarrow$ waiting phase

Then these productions may be represented by the fps in Figure 5.

3. General Interpretations

All arcs which can possibly be drawn between the given fps nodes, associated with the productions of the given system, must be drawn. An arc corresponds to a step in the R-model universe, and can therefore be computed by the R-model interpreter.

All the arcs in an fps have the same basic meaning: if, in any σ of any computation of the system, there is at least one process state contained in each origin node of the arc, then the succeeding σ can include process states contained in the destination node of the arc.

One interaction these arcs do not model is the erasing of a process state that takes place when it functions as a channel name on which a message is accepted. Although it is not clear that we would ever want to model this effect (because, in general, it is not known for sure whether the channel name was generated in the first place), we can deduce how it would have to be incorporated. The arc representing it would have to be consistent with the preceding paragraph, and would have to have the same origin nodes as the (n+m)-arc

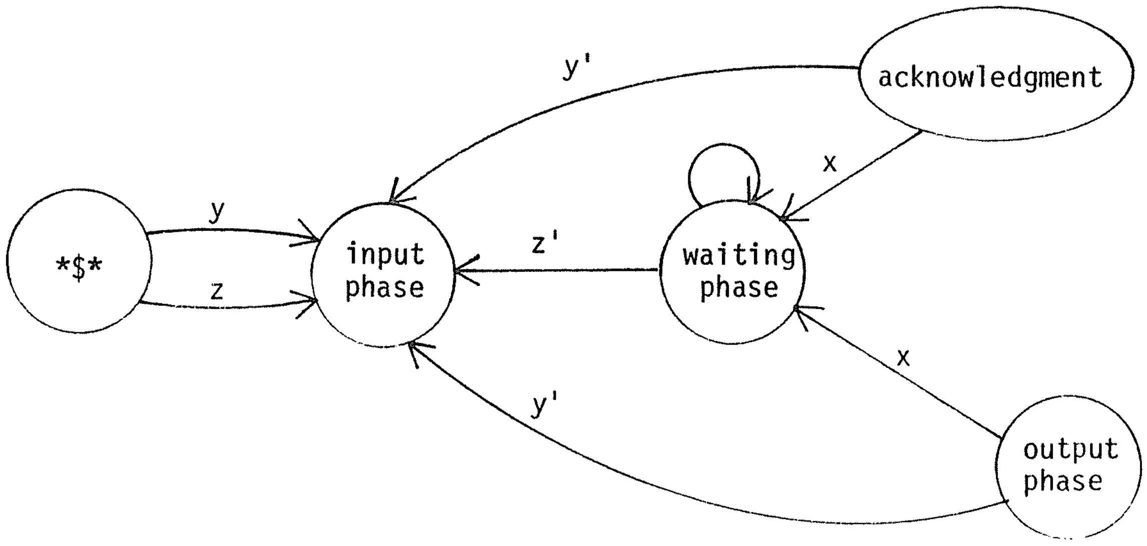

acknowledgment
y'
x
$
y
z
input phase
z'
waiting phase
x
y'
output phase

Figure 5

describing the message acceptance--since it really is an effect caused by the same interaction of process states. The destination node would have to contain the erased process state. Finally, markings on the components would be necessary to show that the arc must be interpreted as an eraser rather than a generator. If we did this by prefixing the component label with a minus sign, the preceding example would turn out as in Figure 6. It is easy to see why we do not want to add these arcs unless they become absolutely necessary!

4. Comparison with Other Techniques for Proving Assertions about Programs and Systems

There are two basic purposes for using finite process structures as abstractions of systems or system complexes. One of them is to attain an understanding of the structure of the system so that it can be factored or otherwise manipulated. This is discussed further in Part VIII. The other purpose is to prove assertions about these systems, based on the confidence that the relationship between systems and their finite process structures is sufficiently rigorous that proofs using finite process structures (in the R-model universe) are valid for systems (in the F-model universe, whose computations are contained in R-model computations, as proved by Johnson).

There is a sufficient number of schemes for proving assertions about programs and systems to create confusion about what objectives each is trying to accomplish. In the hopes of clarifying the matter, thus making it possible to compare our objectives to those of other researchers, we have developed a crude classification scheme into which we will stuff nine research areas as well as our own. They are:

(1) computation graphs [1], [2], [23];
(2) program schemata [20], [7];
(3) Petri nets [14];
(4) specialized graph models [9], [15];

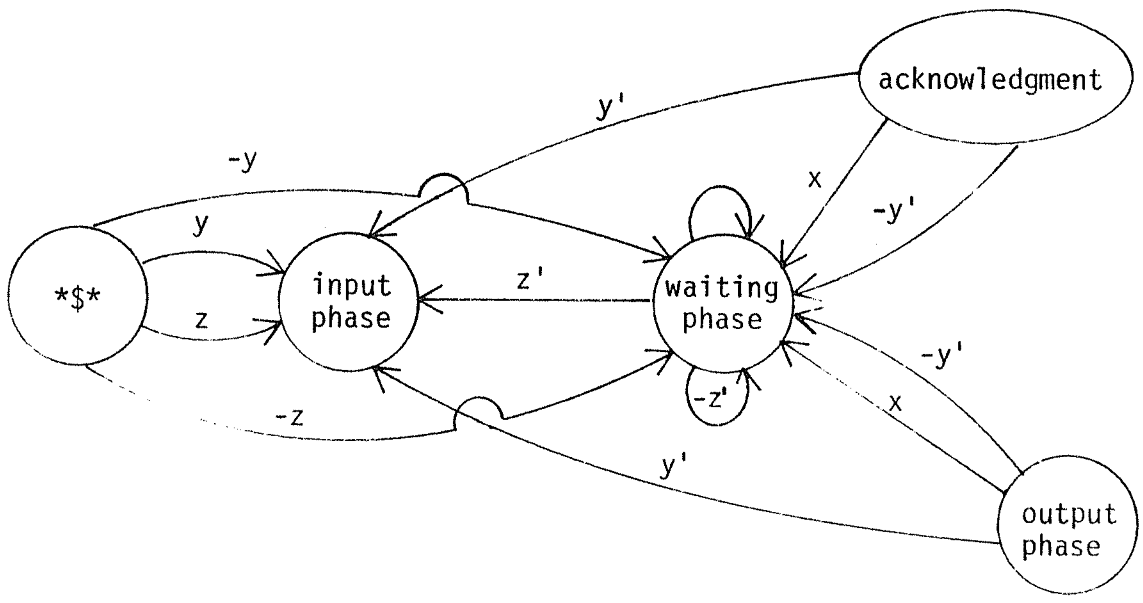

acknowledgment
y'
-y
x
-y'
y
$
z
input phase
z'
waiting phase
-z'
-y'
x
-z
y'
output phase

Figure 6

(5) parallel program schemata [18], [19];
(6) formal and informal program proving [13], [21], [7];
(7) the Vienna Definition Language [29];
(8) operational models (by this we mean models designed to prove particular properties of the operations of particular kinds of systems) [28], [4], [6], [3];
(9) LOGOS [10], [12], [5], [24], [11].

We trust this is a representative sample!

The first category for classifications has to do with the fundamental object that the assertions are about. This is always a program in a programming language, in some form or another--because our interest is in computations, and a programming language is just a notation for directing the generation of computations. The two questions to be asked about a programming language are (1) is it defined by compiler or interpreter (see [29] for a good explanation of the difference)?, and (2) can it only describe a single process, or can it describe interacting processes?

The next category has to do with the abstraction from the fundamental object used by the proof technique. In the cases where the fundamental object is specified, the abstraction is usually computed from the object algorithmically. A good example of this is the formula in first-order predicate calculus which is constructed directly from a program, in Manna's approach to formal program proving. There are many abstractions, however, which are considered sufficiently general to be applicable to many types of fundamental objects, and so the underlying objects are not defined: the theory begins with the abstraction and goes from there. Obvious examples of this are Petri nets, program schemata, and parallel program schemata. The other question about abstractions we can ask is: is the abstraction highly specialized, or can it be used to prove a wide range of properties?

The final question is concerned with the type of assertions to be proved. The two major classes here are assertions about correctness and equivalence, and operational assertions (those about states and conditions arising during computation). These two classes are not mutually exclusive; in fact, many schemes derive assertions of both kinds.

With all humility, we present the classification in Figure 7, hoping that its crudeness will not do too much injustice to the projects entered. The entries are arranged in an order that emphasizes the progression from one concept to another. The classification certainly supports our earlier statement that we share many of the goals of the LOGOS system.

The reason for our objectives should now be clear. We believe that an abstraction should be taken algorithmically from a well-defined fundamental object, because it is the act of finding an abstraction of an extremely complex object that is most challenging, and most beyond the reasonable reach of human capacity. This is enough for verifying assertions, but if we also want to discover optimizations from the abstraction and apply them to the fundamental object, a two-way algorithmic relationship between fundamental objects and abstractions is essential.

We believe that the fundamental object should be interpreter-defined for the reasons given by Wegner in [29] and also in [8]; it should be capable of describing asynchronously interacting processes (without undefined states!), because there is where many of the interesting problems in systems design occur. Finally, we want our techniques, and therefore our abstractions, general enough to prove assertions about a wide range of useful properties of systems designs.

So far we have only mentioned objectives, not possibilities, and it is on the practical side that some of these schemes suffer

SCHEMES FOR PROVING ASSERTIONS	TYPE OF FUNDAMENTAL OBJECT		TYPE OF ABSTRACTION		TYPE OF ASSERTION
	defined by compiler or interpreter?	single- or multiple-process?	unbased or algorithmic?	specialized or non-specialized?	
computation graphs	—	single	unbased	specialized	operations
program schemata	—	single	unbased	specialized	equivalence <u>and</u> operations
Petri nets	—	multiple	unbased	specialized	operations
specialized graph models	—	multiple	unbased	specialized	operations
parallel program schemata	—	multiple	unbased	specialized	operations
formal and informal program proving	compiler	single	algorithmic	non	equivalence
the Vienna Definition Language	interpreter	single	algorithmic	non	equivalence <u>and</u> operations
operational models	interpreter	multiple	algorithmic	specialized	equivalence <u>and</u> operations
LOGOS	interpreter	multiple	algorithmic	non	equivalence <u>and</u> operations
the Formal Definition Universe	interpreter	multiple	algorithmic	non	equivalence <u>and</u> operations

Figure 7

their worst difficulties. For instance, it seems that we are many years away from having mechanical theorem provers powerful enough to resolve the formulas extracted by Manna's program proving method. The representation system of LOGOS may be too complicated to yield many interesting results. We have always been mindful of the need to focus our attention on what can be done in a straightforward, practical way. This has led us to emphasize constraints on designs (which can be viewed as principles of good design) making analysis practical. Thus our approach can be summarized as an attempt to prove all kinds of assertions about all kinds of systems, with a well-structured (effectively decidable!) capability for restricting the kinds of systems or the kinds of assertions when the general cases become impractically difficult.

V. GRAPHS OF SYSTEM COMPLEXES

1. Description of a System Complex FPS

When fps's for individual systems are brought together to make an fps for a system complex, the resultant graph should retain (implicitly or explicitly) the system membership of each node. The reason is that nodes belonging to different systems have a different relationship than nodes belonging to the same system; in particular, nodes belonging to different systems need not be disjoint.

In addition to all the nodes of all the single-system fps's, the system complex fps will have a "dummy" node (because no language is formally associated) for each observer.

Systems and observers can only communicate via the message transmission facility, and we have seen how (n+m)-arcs model this at the intra-system level. The same arcs are used in the same way to describe inter-system communications. Of course, the inter-system (n+m)-arcs do not have quite the same interpretation as the intra-system ones because all the antecedents are not matched in the same σ (the n antecedents of the message-generating production match process states in the sending system; the m antecedents of the channel generator match in the receiving system). There is also an additional element of uncertainty about the eventual creation of a process state contained in the destination node, because the relative rates and transmission times of the systems influence whether the message arrives at the σ'' of the intended receiving system just before the completion of a step in which a proper channel name is generated.

2. An Algorithm for Fusing Single-System FPS's

As before, any arcs which can be drawn must be drawn. To find all the arcs which can be drawn, we begin by making a list of all the (channel language, message language) pairs which can be broadcast throughout the system complex. Each message-sending production in each system can contribute to the list of pairs. If the message-sending production has n antecedents, then it contributes a language pair for each distinct n-tuple of nodes in the fps for that system such that the ith antecedent of the production can match a string in the ith language of the n-tuple--this represents a match of all the antecedents in the R-model universe, and the channel language and message language are the second and first consequents, respectively, generated by the R-model interpreter. The pairs contributed by each observer are easier to come by: they are simply the output channel languages, each with its associated message language.

We will be using as an example the system complex in Figure 8, which has two systems and one observer. The first system (once started by the observer) is just a clock, generating successive binary integers on each step and broadcasting them on the channel "time". The other system monitors them, sending a message to the observer whenever it receives a time divisible by eight. The workings of this system complex are not particularly rate-independent: the monitor may miss any eight-multiple time, if it does not finish a step and seize σ'' before that time is over-written by the next. Anyhow, we will supply all the arcs describing both inter- and intra-system message acceptance at the same time, because the mechanisms involved are identical. Figure 9 is a finite process structure for the complex with no (n+m)-arcs filled in yet.

Now we can write down the list of (channel language, message language) pairs broadcast in this system complex. We have

```
System_1 =

{σ: signal χ: 01
 π: signal → signal                                    or
    start → 0.                                         or
    $. → time                                          or
    {$.$    χ: 01
     π: $0.$ → .$_1 1$_2         or
        $Δ̄1.$ → $_1 Δ̄_1 .0$_2     or
        1.$ → .10$_1             } → $_2. → time}


System_2 =

{σ: time χ: 01
 π: time → time                   or
    $. → time                     or
    $000. → monitor call → observer}
```

Observer:

(output channel language, message language) pairs:
  (signal, start)
  (time, stop)
input channel language:
  observer

Figure 8

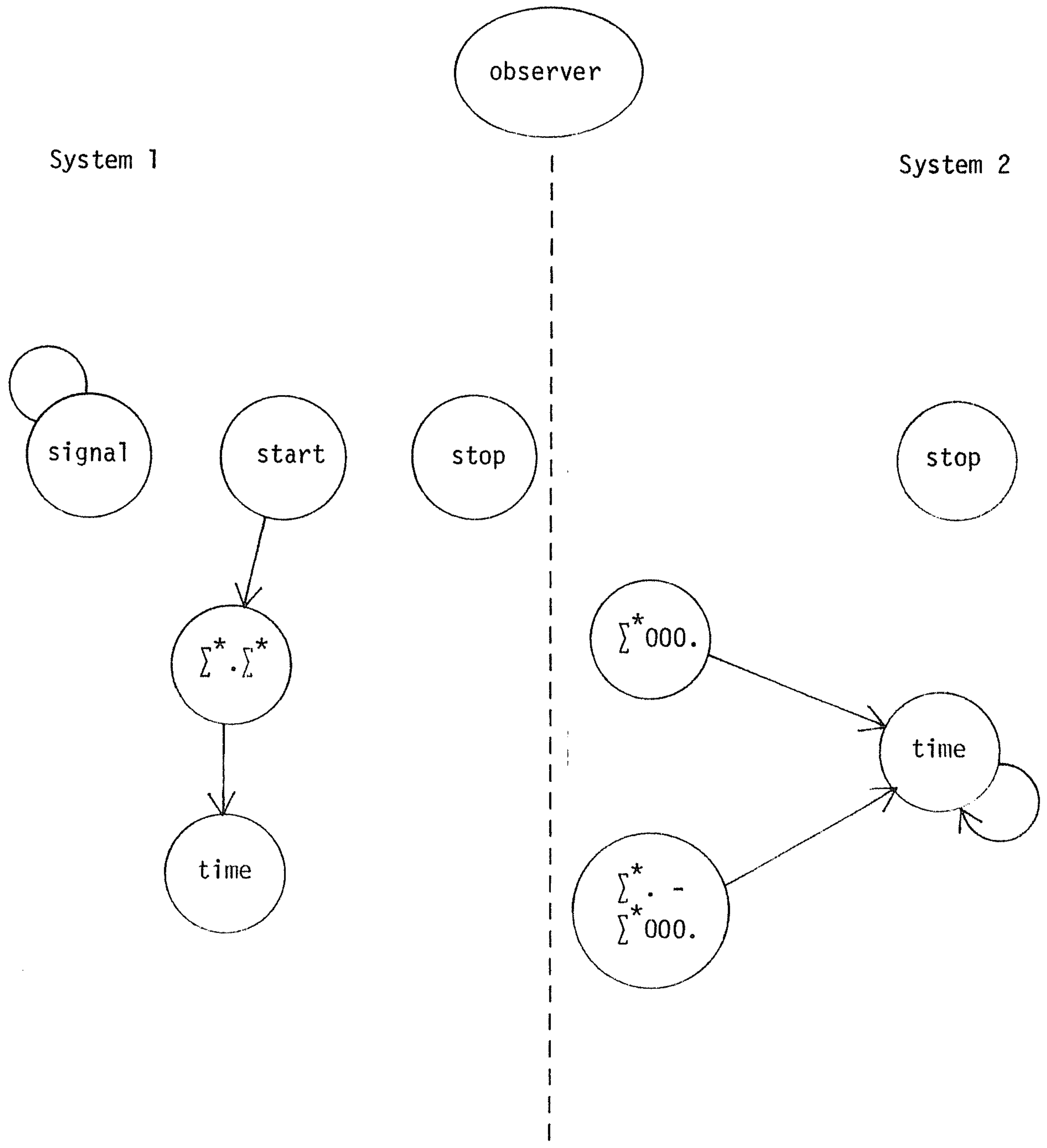

observer
System 1
System 2
signal
start
stop
stop
$\sum^*.\sum^*$
time
$\sum^*000.$
time
$\sum^*. - \sum^*000.$

Figure 9

(1) (signal, start) and (2) (time, stop) from the observer. The antecedent of the fourth production in System$_1$ matches only one node, Σ^*. Σ^* , and so the pair generated is (3) (time, Σ^* .). The third production of System$_2$ also has only one match to the nodes of its fps, generating (4) (observer, monitor call).

The next step is to find, for each pair: (a) all the systems at which it can be accepted, (b) for each system, all the nodes containing strings which can act as channel names, and (c) for each channel name node, all the arcs which can cause the generation of a string in the channel name node. Each entry at level (c) of this informal tree will correspond to an (n+m)-arc in the final fps.

At level (a), a message language can be accepted by a system only if the associated channel language has a non-empty intersection with the union of all the node languages in the system which are destination nodes of at least one n-arc (or the union of all its input channel languages, if it is an observing system). The reason for the restriction on the fpsl is that process states which can only appear in σ as initial states or accepted messages cannot function as channel names. The only exception here is that messages from observers are never accepted by other observers. In our example, we get:

(1)	(signal, start)	accepted at System$_1$;
(2)	(time, stop)	accepted at System$_1$, System$_2$;
(3)	(time, Σ^*.)	accepted at System$_1$, System$_2$;
(4)	(observer, monitor call)	accepted at Observer.

Finding the nodes at level (b) is simply a matter of finding, for each pair accepted at a system, all those suitable nodes such that $[(\text{node language}) \cap (\text{channel language})] \neq \phi$. In our example this is trivial, because each channel language is identical to one node language in each system at which it is received.

At level (c), we remember that the origin nodes of the primed components of an (n+m)-arc are the same as the origin nodes of an n-arc whose destination node contains a possible accepting channel name. Thus for each node of level (b), there is an (n+m)-arc drawn for each n-arc whose destination is that node. Figure 10 shows the completed fps.

One more detail of the procedure must be mentioned. The arcs representing communication between a system and an observer may be simpler than other inter-system arcs, because they need to carry less information. Messages from the observer are always represented by a single unprimed component. Messages to the observer are represented by n-arcs rather than (n+m)-arcs, because there is no channel mechanism to be described.

A certain complication arises when a system can receive as messages strings which are not already contained in its fpsl. A node or nodes must be added to the fps so that these strings will be contained somewhere, but then the union of the fps nodes may not be closed under the system any more. Adjustments will be necessary. This is the primary reason for preferring the total language (or any other language totally closed under the system) as an fpsl: any message language can be added to the fpsl without disturbing its closure property, as the theorem of Section III.2 verifies.

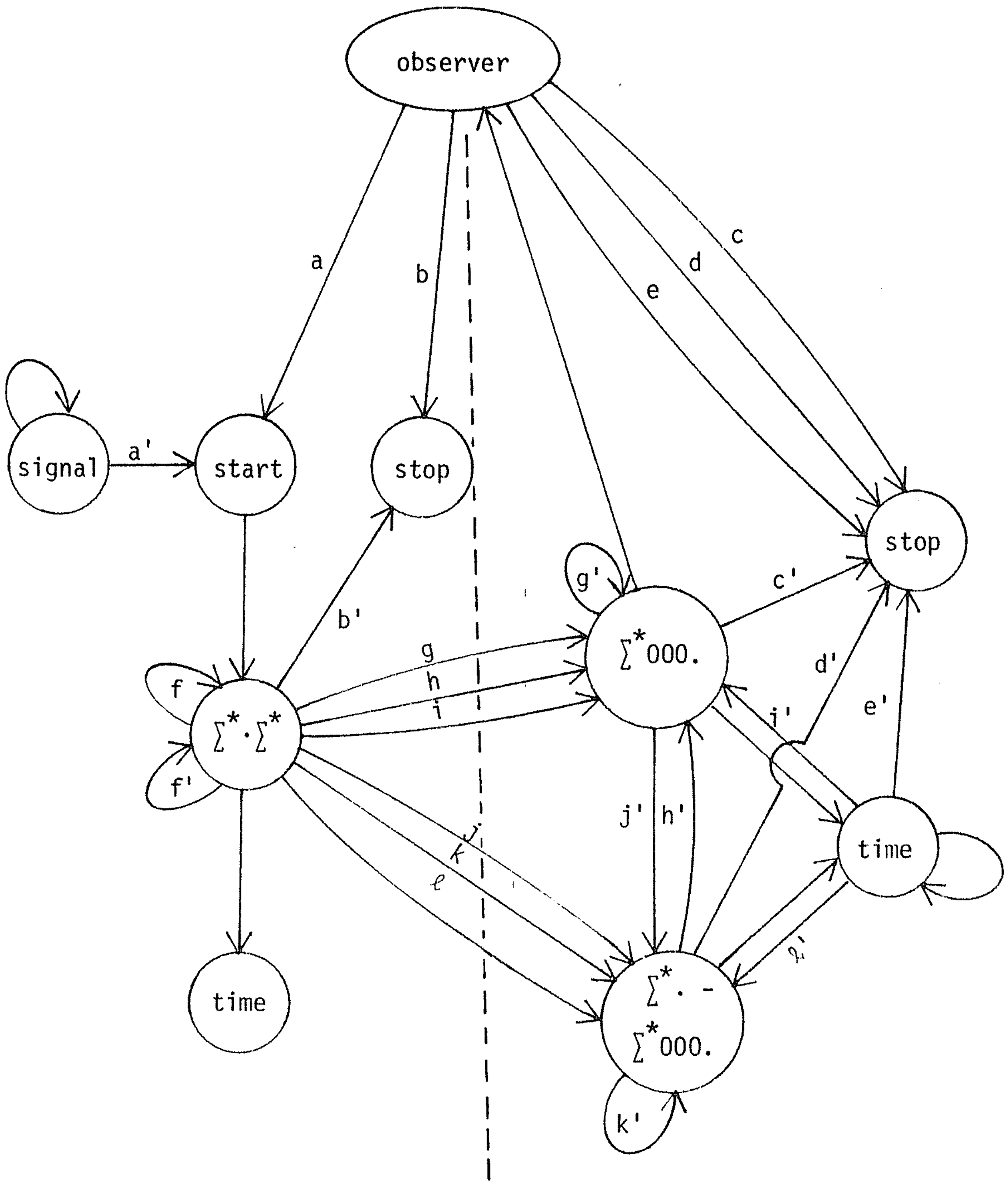

observer
a
b
c
d
e
signal
a'
start
stop
stop
b'
g'
c'
$\sum^*000.$
g
h
i
f
$\sum^*\cdot\sum^*$
f'
d'
e'
i'
j' h'
time
j
k
ℓ
ℓ'
time
$\sum^*. -$
$\sum^*000.$
k'

Figure 10

VI. LATTICES OF FINITE PROCESS STRUCTURES

1. FPSL Choices

Even at the level of isolated systems, there may be an infinite number of finite process structures describing any particular one. Each valid fpsl for the system is associated with a large class of graphs--in this part we will be discussing the properties of such fpsl-classes.

As mentioned in Section III.2, we will usually use the total language as an fpsl, not only because it eliminates problems when drawing an fps for a system complex, but also because we are interested in how our systems process any string that might appear in their σ's. The set of all fps's describing a system whose fpsl is the total language is basic in the sense that it contains all the information about this system which any fps is capable of expressing: any graph whose fpsl is a subset of the total language can be found as a subgraph of some graph in this set; any graph whose fpsl is a superset of the total language contains no additional information. The latter is true because nodes not contained in the total language will not be the origin nodes of any arcs.

There are certainly reasons for using fpsl's smaller than the total language. One of them is to factor the analysis of a system (combinatorics being what they are, two smaller graphs are almost always better than one bigger one). Another is that a small fpsl can lead to a complete, but simple characterization of a system's computations, if it is known that initial states will be restricted to strings in that language.

2. Lattices of FPS's

Unless an fpsl for a system is finite, its fpsl-class of fps's will be infinite, because the fpsl can be partitioned into nodes so many ways. This infinite set of fps's can be arranged in a lattice. Formally, a lattice consists of a pair $<S,R>$ where S is a set of objects and R is a partial order defined on them. A partial order is a relation with the properties:

(1) $\forall x$: $x \, R \, x$ (reflexivity);

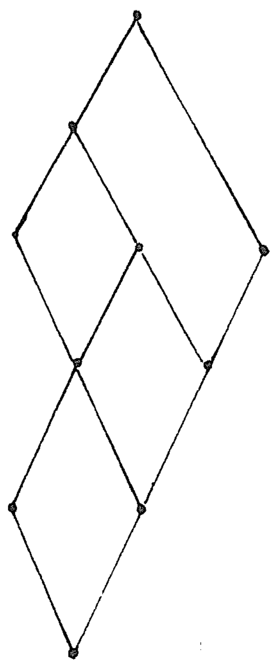
(2) $\forall x,y$: $x \, R \, y \, \& \, y \, R \, x \Rightarrow x = y$ (anti-symmetry);

(3) $\forall x,y,z$: $x \, R \, y \, \& \, y \, R \, z \Rightarrow x \, R \, z$ (transitivity).

For $<S,R>$ to be a lattice there is the additional requirement that any two set members must have a unique least upper bound (join) and a unique greatest lower bound (meet). w is a least upper bound of $\{x,y\}$ if (1) $x \, R \, w$ and $y \, R \, w$ (w is an upper bound of $\{x,y\}$), and (2) z is an upper bound of $\{x,y\}$ $\Rightarrow w \, R \, z$. w is a greatest lower bound of $\{x,y\}$ if (1) $w \, R \, x$ and $w \, R \, y$ (w is a lower bound of $\{x,y\}$), and (2) z is a lower bound of $\{x,y\} \Rightarrow z \, R \, w$. Figure 11 is an example of a lattice.

In our lattice S is the set of all fps's for a particular system with a particular fpsl, and the relation R (symbolized by $\leq$) means "is a lower- (or equal-) resolution form of". The concept of resolution has to do with how detailed the fps is. We say that an fps with many nodes which are small languages has high resolution; an fps with a few large-language nodes has low resolution. It could be quantified roughly by the number of nodes in the fps. In other words, we could draw all our lattices so that fps's with the same number of nodes were on the same horizontal line, indicating that they had similar levels of resolution.

$A \leq B$ if and only if A is an fps which can be derived from fps B by taking as nodes either nodes of B or unions of nodes of B so that each node of B is represented in A exactly once. This rule accomplishes two simple things. First, it ensures that A will have the same fpsl as B, and that its nodes



Dots represent objects in S, and lines represent the relation R. Since R is transitive, if x is connected to y by any upward line or continuous sequence of upward lines, then x R y.

Figure 11

are disjoint. Second, it ensures that all the information in A is also in B , i.e. the whole fps A can be inferred from B , without reference back to the original system. Since all the nodes of A are simple unions of nodes of B , the arcs of B can be transferred directly to A , with each new arc component passing from the node of A containing the origin language of its counterpart in B , to the node containing the destination language of its counterpart in B . Figure 12a is a part of a lattice in which the fps's are represented by their nodes (R, S, and T are regular languages). All three fps's on the middle level have the relation $\leq$ to the fps on the top level; the fps on the bottom has the relation $\leq$ to every other fps in the sublattice. In Figure 12b we see the top fps with its arcs, and a middle-level fps derived from it.

Theorem: $\langle S = \{$all fps's for a given system with a given fps1$\}$, $\leq \rangle$ is a lattice.

Proof: First we must show that $\leq$ is a partial order, then that any two fps's in S have a unique meet and join.

Part I: $\leq$ is a partial order.
Let x, y, z be elements of S .

(1) $\leq$ is reflexive because an fps can be derived from itself by taking as nodes the nodes of itself.

(2) If $x \leq y$ then all the nodes of x are nodes of y or unions of nodes of y . Obviously the only way that y could be constructed from nodes of x or unions of them would be if $x = y$, so $\leq$ is anti-symmetric.

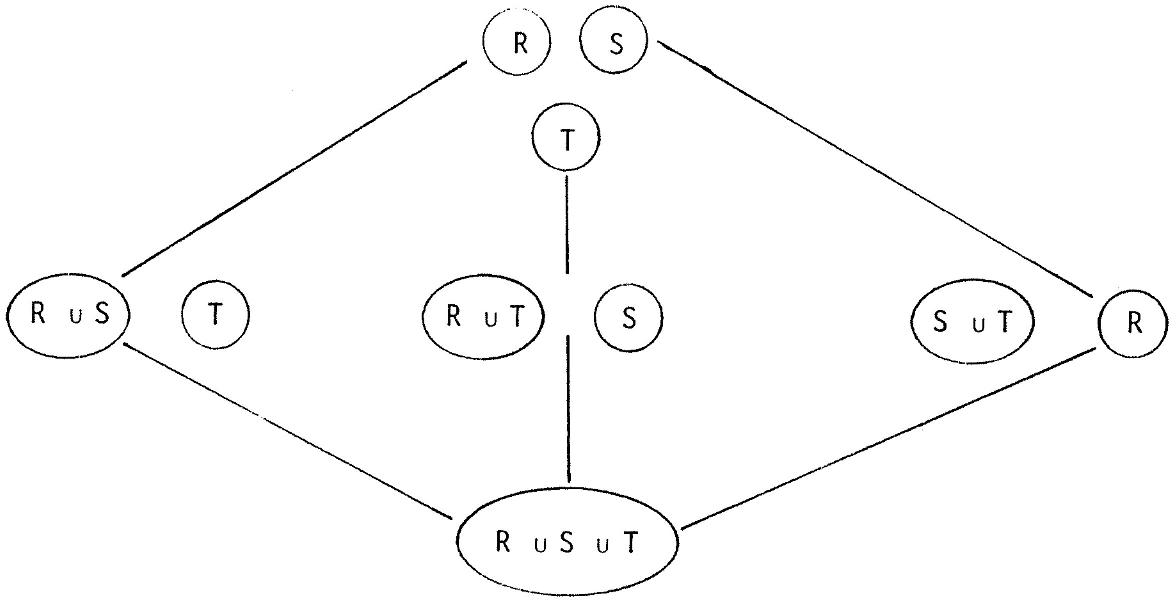


Figure 12a

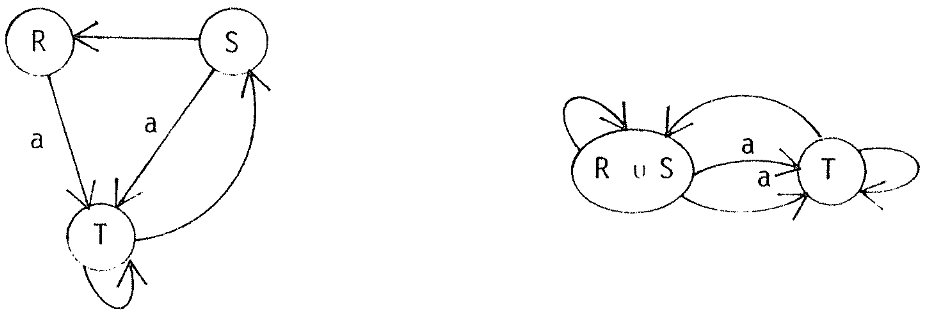


Figure 12b

(3) If $x \leq y$ and $y \leq z$, then x could be derived from z by constructing each node of x as the union of all the nodes of z that went into all the nodes of y that went into that node of x. Therefore $x \leq z$, and $\leq$ is transitive.

Part II: Any two fps's in S have a unique meet and join.

Let A and B be two fps's in S with nodes $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$, respectively. Suppose we draw a grid whose rows represent the languages a_i and whose columns represent the languages b_j, as in Figure 13. Thus both the rows and the columns represent partitions of the fpsl, and the fpsl could be expressed as $\bigcup_{i,j} (a_i \cap b_j)$. We put an "x" in the intersection of a_i and b_j if $(a_i \cap b_j) \neq \phi$.

The join (least upper bound) of A and B is the fps C whose nodes are $\{(a_i \cap b_j) | (a_i \cap b_j) \neq \phi\}$. This fps is certainly unique, and is certainly an upper bound of A and B : $A \leq C$ because each node a_i of A is $\bigcup_j (a_i \cap b_j)$; $B \leq C$ because $b_j = \bigcup_i (a_i \cap b_j)$. It only remains to be shown that if any other fps C' is an upper bound of A and B, then $C \leq C'$. This is true because $\{(a_i \cap b_j) | i = 1,2,\ldots,n ; j = 1,2,\ldots,m\}$ is the largest partition of the fpsl contained in both $\{a_i | i = 1,2,\ldots,n\}$ and $\{b_j | j = 1,2,\ldots,m\}$. Another way to express the relation $\leq$ is to say that $I \leq J$ if and only if the nodes of J represent a partition of the fpsl which is contained in the partition induced by the nodes of I.

The meet D (greatest lower bound) of A and B is the fps with the largest number of nodes partitioning

	b_1	b_2	. . .	. . .	. . .	. . .	b_{m-1}	b_m
a_1	x						x	
a_2	x		x					
.		x			x	x		
.		x				x		
.			x		x	x		
.				x		x		x
.	x							
a_{n-1}			x		x			
a_n				x				

Figure 13

$\bigcup_{i,j} (a_i \cap b_j)$ such that each node of A or B is completely contained within a node of D . A node d_k of D can be found by selecting some node of A a_i: if any node b_j of B intersects a_i, then d_k must be at least as large as $a_i \cup b_j$. If any node of A intersects the new d_k then it must be added to d_k , as must any node of B , etc. When no more node languages can be added in this way, d_k is a node language of D . The others can be found by a similar construction, using nodes of A and B not already included in nodes of B (until all have been used).

D is unique by construction. It is $\leq$ to both A and B because each node is, by construction, both a union of nodes of A and a union of nodes of B . Thus both A and B could be derived from D by splitting nodes. Finally we must show that any other lower bound of A and B is $\leq$ the meet. The nodes of such an fps, D' , would also have to be unions of nodes of A and of nodes of B . Inspection of the algorithm for finding a node of D will convince the reader that it forces only unions of a_i's which are absolutely necessary so that the result will also be a union of b_j's , or vice versa. Therefore the nodes of D' would have to contain the nodes of D , which would imply $D' \leq D$.

Q.E.D.

The reason for disallowing changes to the fpsl during a change in resolution is that no lattice ordering is possible under such circumstances. A lattice ordering can only exist for a set of fps's all with the same fpsl, because steps through a lattice must be uniquely invertible.

The reason for restricting the nodes to being disjoint regular languages is to eliminate redundancy.

It is easy to imagine graph algorithms, such as those whose intention is to partition the fpsl (an important case! see Part VIII), being complicated or destroyed by overlapping node languages.

There are reasons why either of these two degrees of freedom might be desirable for using fps's to model systems, but both of them are really available with the definition of an fps as it stands. Any graph with overlapping or contained nodes can be reached from an fps by a straightforward mapping. Figure 14b shows a descriptive graph with node languages P and Q overlapping, and S completely contained in R . The arcs for this graph are easily found by transferring the arcs of the fps in Figure 14a.

Often a jump to a higher level of resolution entails splitting special cases off a generalized node language, and it becomes clear that the strings left over in the generalized node never appear in computations. They cannot be thrown away from the fpsl, but they can probably be removed from the active part of the fps quite easily: if these leftover strings are put in a node by themselves, they will not be touched by any arcs. Much the same is true of the reverse process, in which padding is added to specialized nodes to make them into a generalized node, while lowering the resolution of a graph. In either case it might be best to let the fpsl be the set of all strings over the global alphabet, and equip any graph algorithms to handle inactive nodes efficiently.

The shape of this lattice is that of a tree which branches upward infinitely. At the root is the meet of the entire lattice--an fps whose single node is the fpsl. The lattice does not have a join, but we can imagine one by pretending we could take the limit of the upward branching of the tree at infinity. This graph (not really an fps any more, since its number of nodes would be infinite) would include all the information that could ever be obtained from any F-model run of this system, except for the types of information intrinsically lost in R-model simulation.

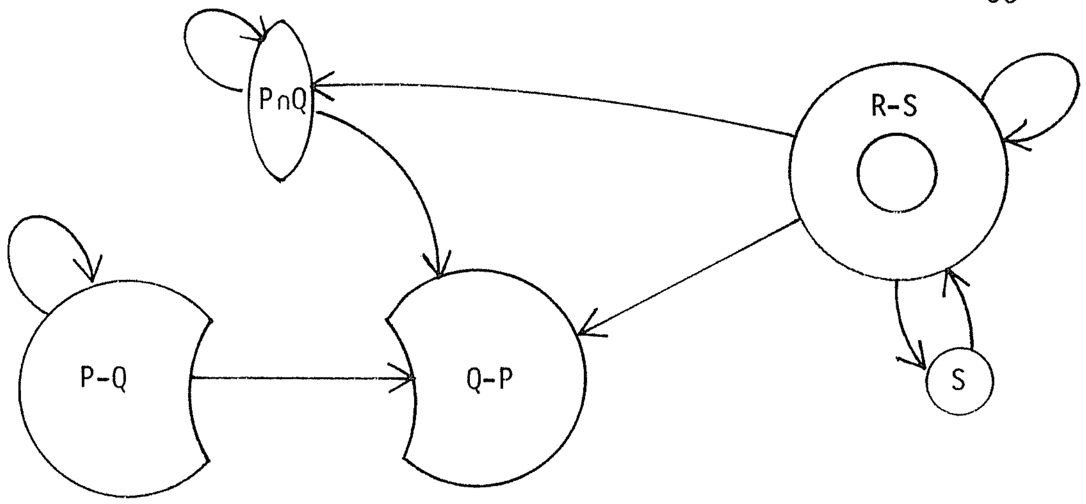


Figure 14a

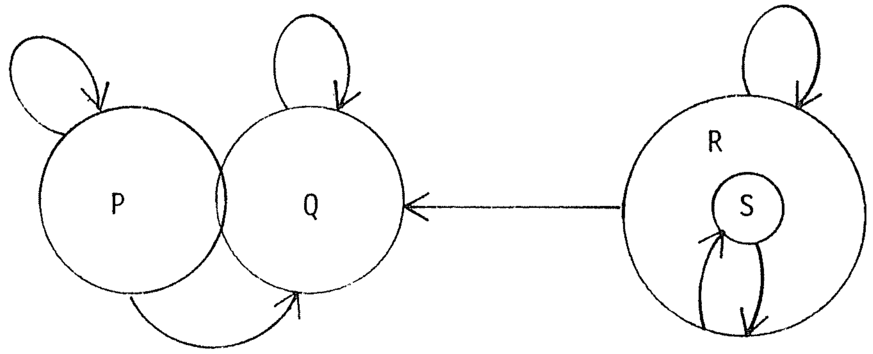


Figure 14b

What is the significance of this lattice? It means that there is an fps describing any part of the system in any level of detail, and the organization imposed by the lattice order provides an algorithm for focusing on it, relating it to the information in other fps's, computing the cost of obtaining such information, etc. For example, the join of any two fps's has been proved to be the lowest resolution, i.e. cheapest, fps containing all the information in both of them. We feel that flexibility in fps characterizations makes our tools practical, and the lattice ordering makes the flexibility manageable.

3. Finite Complete Sublattices

A complete lattice is one in which every set of elements, instead of just every pair, has a unique meet and join. When we wish to study any specific property of systems, we will probably be interested only in a finite, complete sublattice of the infinite lattice which will contain all the information relevant to this property expressable in an fps. Figure 15 shows the relationship between a finite complete sublattice and the infinite lattice it is embedded in. Note that a complete lattice always has a unique "top" and "bottom", which are the join and meet of all the elements in the lattice, respectively.

One way of arriving at a finite, complete sublattice would be to select as its join an fps containing all the information that could ever be relevant to the problem. Then the lattice is formed by lowering the resolution of the join (merging its nodes) repeatedly and in different ways, until the single-node meet is reached. The result is guaranteed to be finite, because there is only a finite number of ways to simplify an fps by merging its nodes.

Another way of forming such a lattice might be to start from the bottom and raise resolution only in ways that provide quantum jumps in the amount of information displayed about the property under study. With a little bit of luck, this repeated raising of

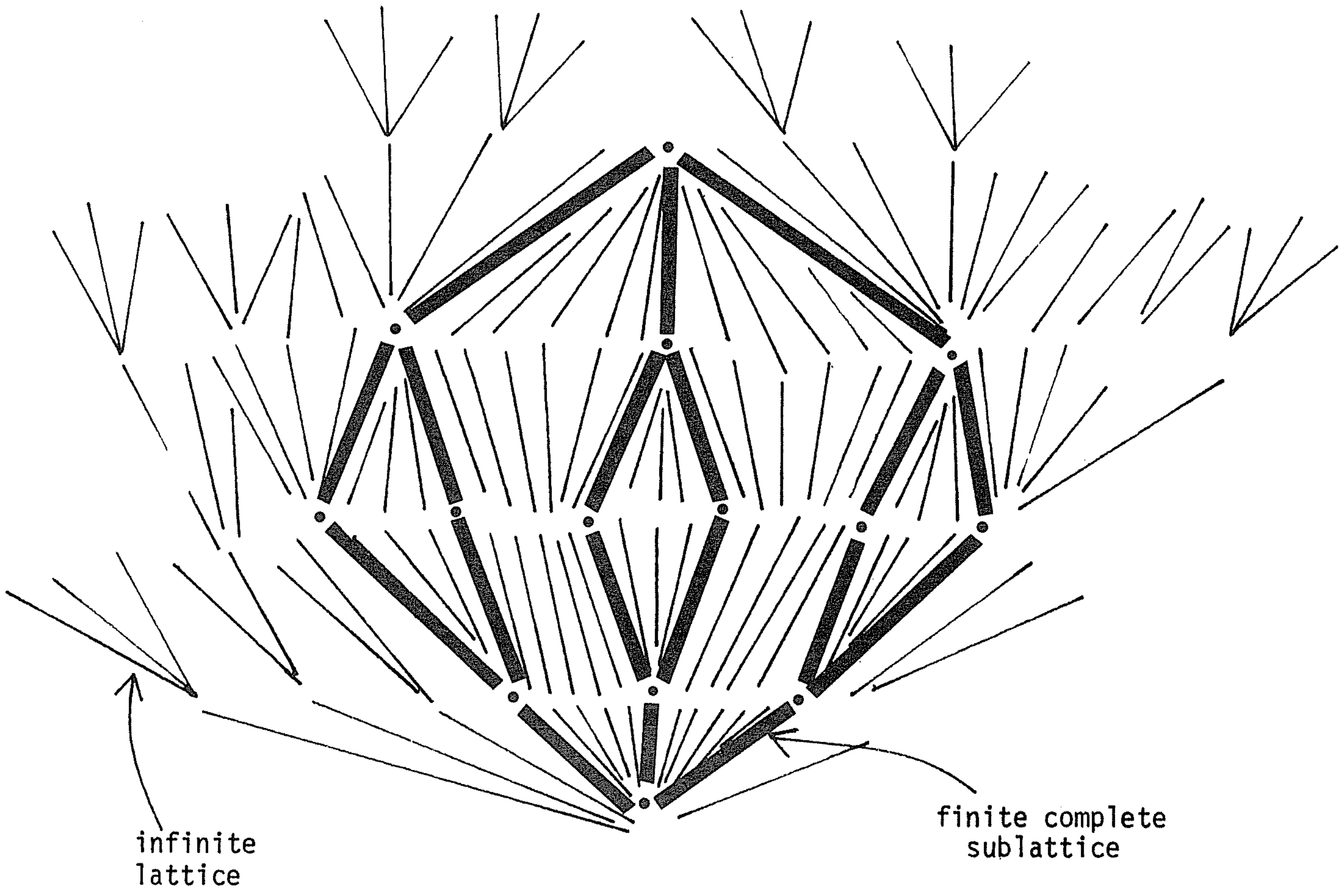


Figure 15

resolution will converge on a maximum-resolution member, the join of the complete lattice.

It is clear that no one is going to compute an entire sub-lattice for any purpose. Our point in describing these structures is that when a designer is searching for the fps which will give him all the information he needs in the simplest way, he should know that only a finite number of graphs are potentially useful and significantly distinct.

VII. THE CANONICAL FINITE PROCESS STRUCTURE FOR A SYSTEM

1. Determinism in FPS's

The foremost property of systems and system complexes that we intend to study using finite process structure graphs is, of course, their process structure. Therefore we would like to find an fps for each system to serve as the join of a finite, complete sublattice containing all the information we can use about the process structure of the system. This fps will be so important to us that we will call it the canonical fps for the system.

We will associate canonical fps's with single systems, for much the same reason that we associate fpsl's with single systems. Later we will define a canonical extension of them to fps's for system complexes. Meanwhile we must recognize that the selection of a canonical fps is basically a design choice, with the purpose of getting the most useful information without incurring any impractical costs, and look at which fps properties are most desirable.

Determinism is a property of arcs in an fps which seems to be related to the quality of the fps for revealing process structure.

Definition: An arc of an fps for an isolated system is deterministic if and only if each time σ of the system contains at least one process state in each one of the arc's origin nodes, then the succeeding σ will contain at least one process state in the arc's destination node.

Experience and intuition indicate that if most of the arcs of an fps are deterministic, then the boundaries between nodes reflect genuine distinctions between different categories of process states, and the whole fps gives an interesting picture of what goes on during the system's computations.

The simplest reason why an intra-system arc might not be deterministic is that one of its origin nodes might not be contained in the pattern language of the antecedent it is supposed to match--then a process state belonging to the node language could be in σ, but would not match the antecedent, if it was in the language L = (node language - antecedent language). This situation is illustrated in Figure 16a. It is easily remedied by splitting the origin node of the arc as in Figure 16b.

Another cause of non-determinism is that the successor language to an origin node, as calculated by the R-model interpreter, might overlap two or more fps nodes. In this case it is not determined which node will actually contain the successor, as shown in Figure 17a. Figures 17b and c give two solutions to the immediate problem.

To carry out a transformation like the one resulting in Figure 17b, it is necessary to know where to split the origin node. The division of the node language $(L_1 \cap L_2)$ into L_1 and L_2 was dictated by the division between $suc(L_1)$ and $suc(L_2)$ caused by the successor node boundary. Certain properties of the R-model interpreter ensure that the correct partition of the origin node language can always be found.

Let L be an origin node language and L' its R-model successor under a production. Because all F-model computations are contained in their corresponding R-model calculations, we know that every string in L has as its successor a string in L'; if we partition L' into L_1' and L_2' because of a node boundary, that will induce a corresponding partition of L. Because of the closure of regular languages under intersections, R-model steps, and inverse R-model steps, the languages L_1 and L_2 such that $L = L_1 \cup L_2$ and the R-model successors of L_1 and L_2 are L_1' and L_2', respectively, will be regular languages.

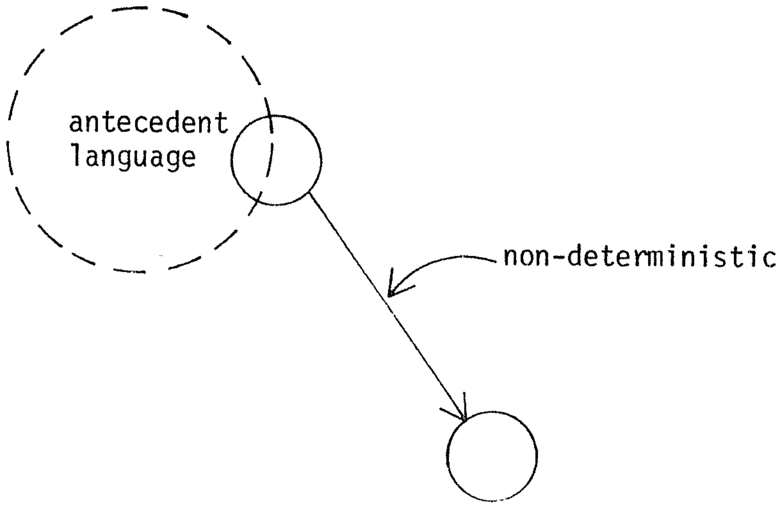


Figure 16a

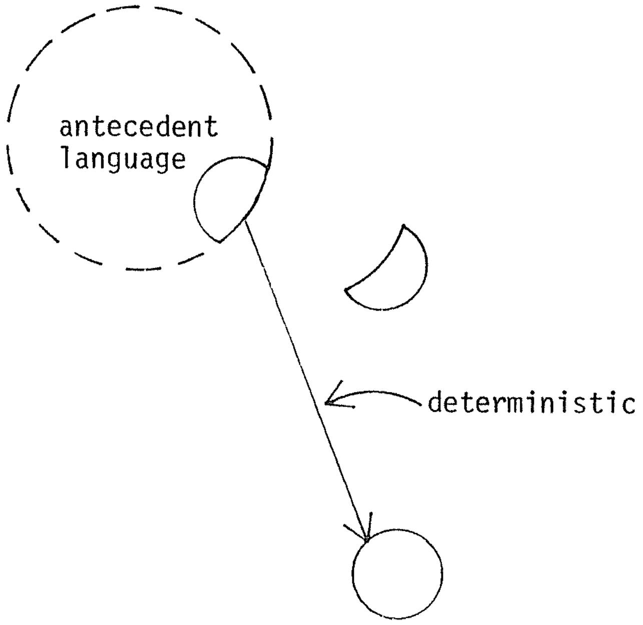


Figure 16b

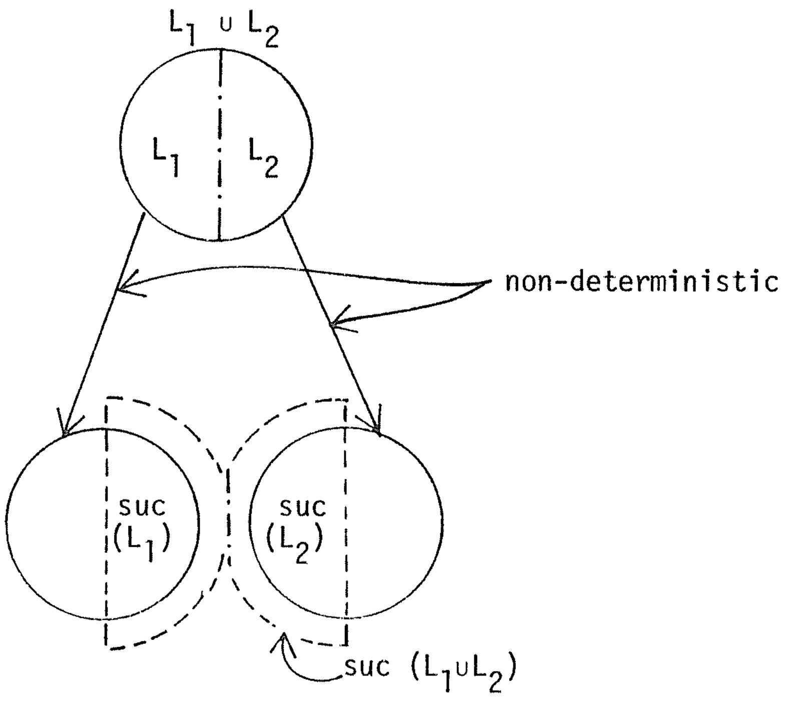
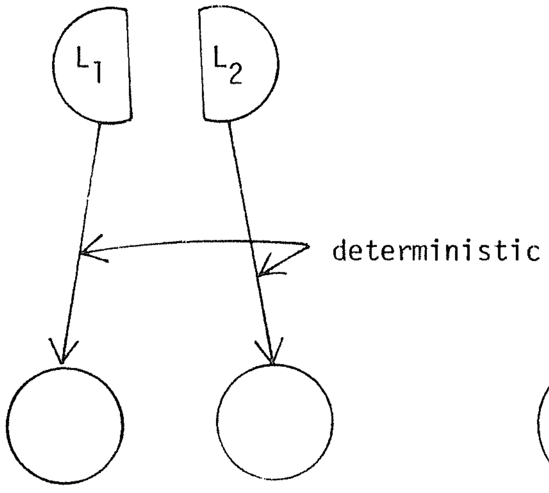


Figure 17a



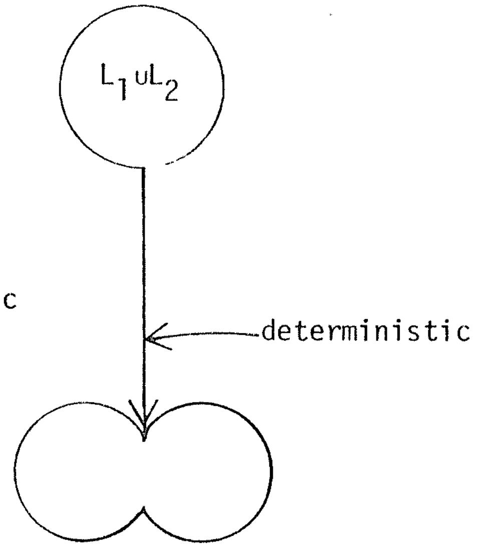


Figure 17b

Figure 17c

Unfortunately, the problem is not so simple. Any splitting of nodes, as in Figure 17b, can cause arcs whose destination node is the split node to become non-deterministic. Any merging of nodes, as in Figure 17c, can cause arcs whose origin nodes are the merged nodes to become non-deterministic. Thus any change which improves the determinacy of the fps in some neighborhood may destroy it in another, and so there is no algorithm for making an fps fully deterministic. One of the most common examples of this is the case in which there is a chain reaction of node-splitting which does not terminate. We see such an example in Figure 18. Figure 18a is a fragment of an fps the arcs of which represent these productions:

$\bar{\Delta}\,\bar{\Delta}\,\$ \rightarrow \bar{\Delta}_2\,\$_1$ <u>or</u>

$\Delta \rightarrow$ stop

Presumably the way to remove the non-determinism is to split $\bar{\Delta}\,\bar{\Delta}\,\Sigma^*$ into two nodes, one whose successor will be $\bar{\Delta}\,\bar{\Delta}\,\Sigma^*$ and one whose successor will be $\bar{\Delta}$, deterministically. We see the result in Figure 18b: the non-determinism is still there, but it has been pushed back one node. There is no way to achieve determinism in this fps short of having an infinite set of nodes $\bar{\Delta}$, $\bar{\Delta}\,\bar{\Delta}$, $\bar{\Delta}\,\bar{\Delta}\,\bar{\Delta}$, $\bar{\Delta}\,\bar{\Delta}\,\bar{\Delta}\,\bar{\Delta}$, ...

Although all the illustrations to the above remarks used 1-arcs, they can be generalized straightforwardly to n-arcs or (n+m)-arcs. With (n+m)-arcs there are additional problems, so that these can be made deterministic only under the most special circumstances. Any inter-system (n+m)-arc is intrinsically non-deterministic because of the vicissitudes of relative rates. Even intra-system (n+m)-arcs, which are as fully synchronous as n-arcs, can be made deterministic only if the accepting channel language (the intersection of the channel language produced by

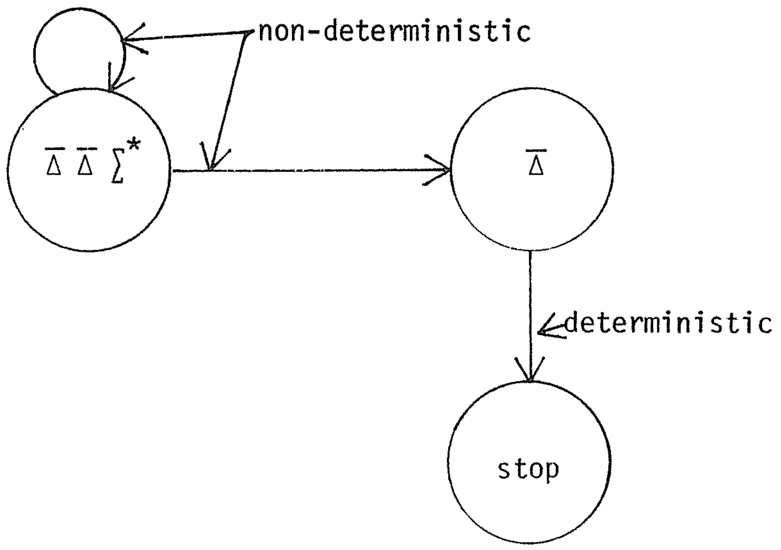


Figure 18a

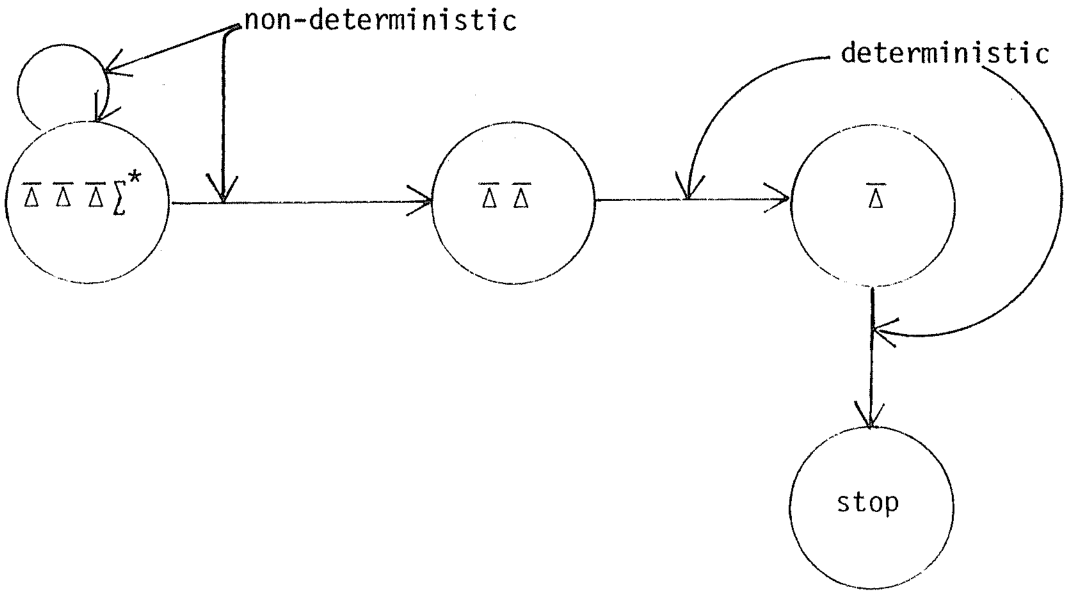

non-deterministic
deterministic
$\overline{\Delta}\,\overline{\Delta}\,\overline{\Delta}\,\Sigma^*$
$\overline{\Delta}\,\overline{\Delta}$
$\overline{\Delta}$
stop

Figure 18b

the message-generating production, and the language from which the accepting channel must come) is finite. Then a separate (n+m)-arc must be drawn for each of the members of this language. The reason is simply that a message will be accepted if and only if there is a process state in σ that matches its channel name exactly--an overlap of containing languages is just not good enough. Of course, even if that step could be carried out, the resultant splitting of nodes to make the new arcs deterministic could be impossible, or disastrous to the determinism of the rest of the graph.

2. Definition of the Canonical FPS

Because of these difficulties we gave up on deterministic graphs and tried to find something more practical. There is a graph rather closely related to deterministic ones which is easily found, and which may even be more meaningful for studying process structure: this will become our canonical fps. The philosophy behind it is very simple. If a node is completely contained within any antecedent language it intersects, then it is known that any arc component whose origin node is that one really applies to the whole node, and the antecedent will always be matched by any process state in that node. If a node is completely contained within any consequent language it intersects, then we have precise knowledge of all the ways in which a process state in that node could have arisen. Thus the flow of process states through operators to other process states is accurately characterized. Our canonical fps is constructed so that all nodes have these two properties.

Definition: The canonical finite process structure for a system is an fps whose fpsl is the total language for that system. The fpsl is partitioned into nodes as follows:

The set of all antecedent languages of productions in the system, plus the set of all consequent languages, plus the set of all message languages such that their associated channel languages have non-empty intersections with the total language (see Part III), is a set of n languages $\{L_1, L_2, \ldots, L_n\}$ such that each L_i is contained in the total language and $\bigcup_{i=1}^{n} L_i$ is the total language. There may be as many as 2^n-1 nodes, each one corresponding to some member of the power set of this set except ϕ. If $\{L_1, L_2, \ldots, L_m\}$ is a member of the power set, then the corresponding potential node language (potential because it may be ϕ) is $L_1 \cap L_2 \cap \ldots \cap L_m \cap \overline{L_{m+1}} \cap \overline{L_{m+2}} \cap \ldots \cap \overline{L_n}$ where $\{L_{m+1}, L_{m+2}, \ldots, L_n\}$ is the set of all members of the n-language set which are not in $\{L_1, L_2, \ldots, L_m\}$.

This has many similarities to the elusive deterministic fps (notice that the first-presented type of non-determinism cannot occur), but also some substantial advantages over it. To begin with, if the antecedent and consequent language are delimited pattern languages, all the nodes of the canonical fps are delimited pattern languages, which certainly need not be the case in a deterministic fps. Secondly, the desirable properties of any fps in the canonical sublattice will be enhanced by raising its resolution. The most frustrating aspect of the search for determinism is that both raising and lowering resolution can result in local successes--so one never knows which direction to try.

Another serious problem with the deterministic fps that the canonical one avoids is the necessity of performing R-model steps just to arrive at the nodes of the fps. Since we have so many other computational problems, introducing a complex iterative process at such an early stage of analysis is just not reasonable unless there is no alternative. The canonical fps, of course, can be defined by a single set of language intersections (which can be performed very efficiently using string operations on delimited pattern expressions, if the antecedent and consequent languages conform to restrictions). Furthermore, the canonical fps is the best characterization of the system obtainable without using R-model steps to calculate nodes, because it uses _all_ the language information present in the SR.

What can we say about the canonical fps in its own right? Only the intuitive conclusion that it is a high-quality description of the process structure of its system (the highest quality obtainable without extraordinary measures) and that it is sufficiently detailed so that analysis restricted to this fps and lower bounds of it will be able to achieve its purposes. This is true for two reasons: (1) obviously, in any neighborhood where special detail is necessary, it will be possible to depart from this finite sub-lattice to get it, and (2) the problem is usually to make things simpler, rather than more complicated--the canonical fps is probably far too detailed itself for most practical large-scale analysis jobs.

Now that we have a canonical fps for an individual system, it is easy to define one for a system complex. There is one node for each observer. Each participating system has all the nodes of the canonical fps for that system. It has, in addition, one node which is [(the union of all the message languages of all the (channel language, message language) pairs in the complex such that the channel language has a non-empty intersection with the total language of this system) - (the total language of this system)]. This extra node absorbs all the process states not in a system's total language that could be accepted as messages by it. It does not matter that all the message languages are lumped together, because so long as they are outside the total language they will not be the origin nodes of any arcs. Of course, the system's fpsl is still closed, because the total language of a system plus anything else is still closed under its system.

VIII. PROCESSES

1. Syntactic Process Structuring

We are interested in processes because they are the basic dynamic units of computation, and must be understood before we can hope to define interesting transformations on system complexes. Any division into processes is subjective (as well as hierarchical, so that processes on one level contain many processes on a lower level), so we cannot expect the answers to questions of process structure to be unique. The ultimate justification of a process structuring technique must come from examples showing that it is intuitively reasonable, and contributes to the solution of practical design problems.

Recently several groups of researchers have given the concept of process precise definitions. We feel that the definition of Horning and Randell [16] is outstanding for its clarity and generality, and so we shall consider it the state-of-the-art process definition, and refer to it often for comparison. Our first observation about Horning and Randell's definition is that it is representation-independent. Thus it is applied with some difficulty to working processes (which can only be observed through their practical representations), usually only as a verification that something perceived as a process does indeed fit the definition.

Our philosophy is that one of the most important tasks of automated design analysis is to discover process structure where the design is too intricate for the structure to be apparent. This is a key step on the way to developing algorithms for identifying and manipulating processes without recourse to supplementary information or human intervention. This philosophy leads us to a syntactic theory of process structuring to be established within the context of the formal definition universe. The idea is that with a representation-dependent definition of a process, we can identify processes strictly on syntactic grounds, and therefore find process structure algorithmically. The same concept is behind "structured programming" as founded by Dijkstra.

Since our definition of a process depends on an fps, the level of resolution of the fps places an upper bound on the level of detail of the process partition. Process structure is altogether dependent on the underlying fps, and any valid application of this definition must take cognizance of the fact that processes are arranged in a relative hierarchy.

Definition: Given a system complex and an fps describing it, a process is 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).

As an example of this definition in action, see Figure 19. The dotted lines show one way of partitioning this system complex into processes.

The computations of a process are simply the sequences of process state sets, consisting solely of process states belonging to the process, generated by the system complex as it runs. If the process has states belonging to more than one system, it can change state whenever one of the participating systems finishes a step.

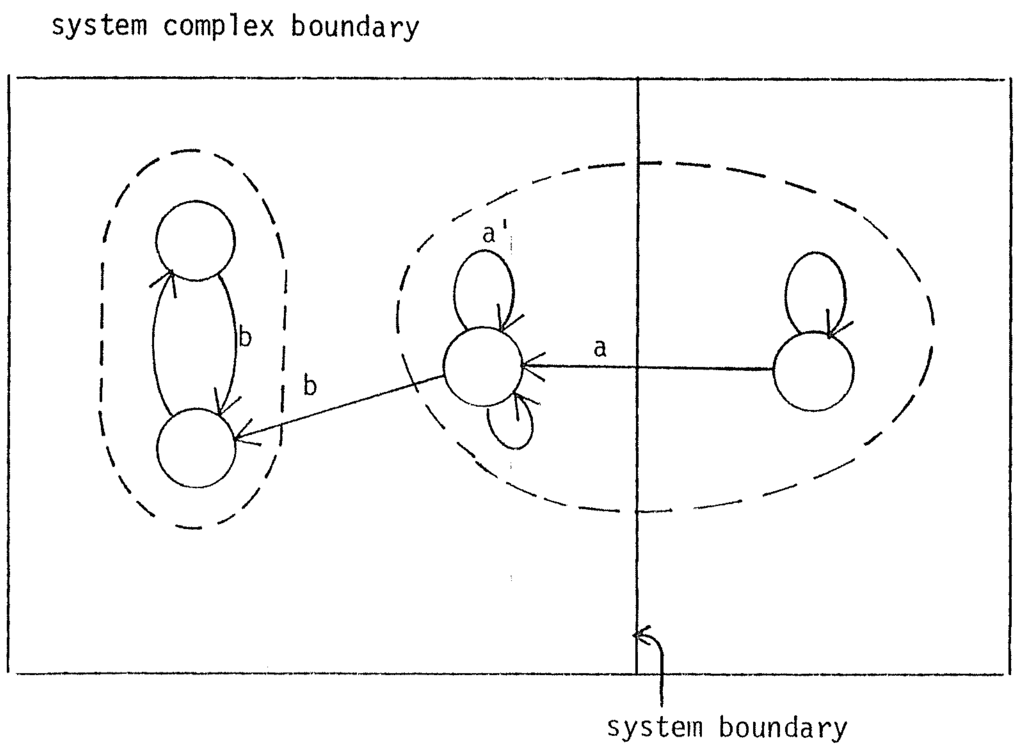


Figure 19

Although this definition is in an unfamiliar guise, it is actually very similar to Horning and Randell's. The two definitions will be compared closely in the next section.

For all aspects of our definition which are paralleled by Horning and Randell's, we will let that stand as sufficient justification. This leaves to be explained only our choice of where process boundaries can be drawn, because Horning and Randell's processes do not interact with each other at all. Figures 20a and b show the kinds of process boundaries precluded by our definition: a process state should certainly be in the same process as a single process state generating it, as should a process state generated by the interaction of two states in the same process. Otherwise we would find ourselves claiming that different times on a cyclic clock were parts of different processes, among other improbabilities. We know intuitively that a process is a cohesive unit; we partially express that cohesion in the rule that any process state generated solely by antecedents in one process must also belong to that process.

Figures 20c and d show process boundaries that may be drawn, and indicate what interactions look like. Note that the two origin nodes represent any number greater than one, and that it does not matter if the multiple components are primed or unprimed. Consequently, any system boundary can be a process boundary, because inter-system arcs are always (n+m)-arcs with n origin nodes on the sending side of the system boundary and m origin nodes on the receiving side. Incidentally, sending a message to an observer is considered to be an interaction, even though all the origin nodes of the arc representing it may be in the same process. This is so because the arc is symmetrical in meaning to (n+m)-arcs, and only fails to have m-components because we do not model message acceptance in observers.

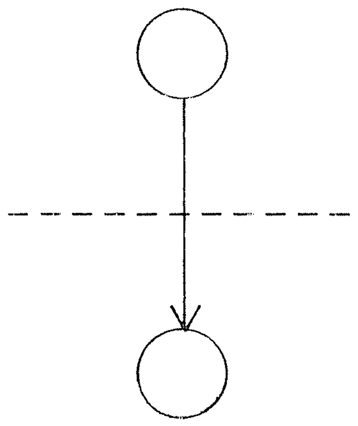


Figure 20a

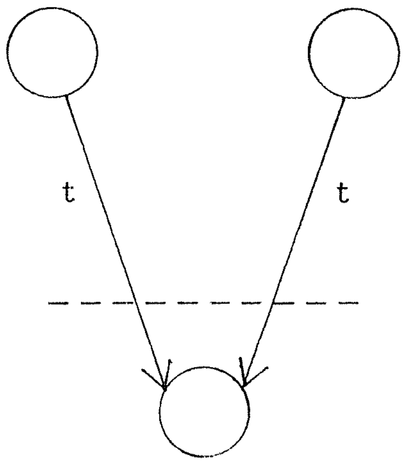


Figure 20b

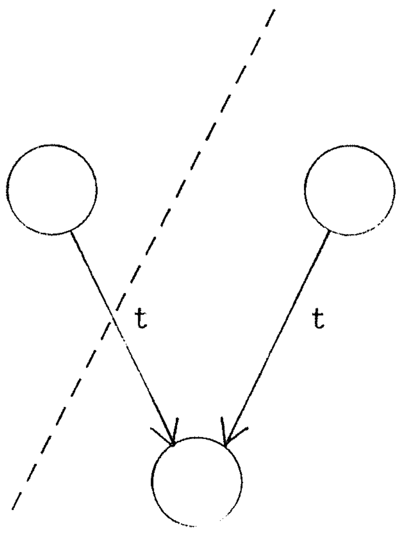


Figure 20c

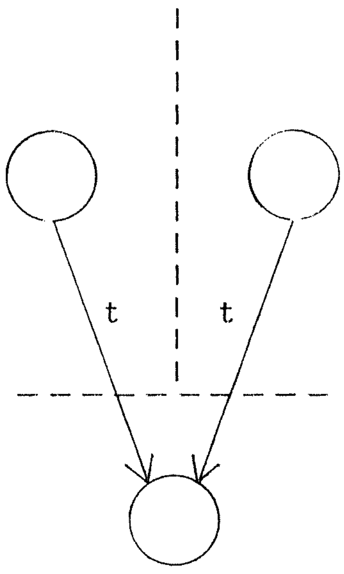


Figure 20d

The next step in building this theory will be to classify specific process structures and learn to exploit their properties. Preliminary work in this direction has been encouraging.

2. Comparison with Horning and Randell's Processes

Horning and Randell's definition of a process fits most of the diverse viewpoints that the authors investigated:

> A process is a triple (S,f,s) where S is a state space, f is an action function in the space, and s is the subset of S which defines the initial states of the process.

The state space consists of all possible values of a set of state variables; the action function f has a domain contained in S, and a range consisting of assignments of new values to state variables.

In the formal definition universe there are no fixed sets of variables: the number and size of process state strings is completely fluid. Thus we need not constrain the state space of a process as rigidly as Horning and Randell do. The natural way to characterize our state spaces is with regular languages describing the delimiter structure of process states, which is how it is done--so our regular languages play exactly the same role as S.

Horning and Randell's action function has its analog in a successor function (recognized by Horning and Randell to be an equivalent notion) defined by the productions of the systems involved. Although we do not have an explicit set of initial states, the same purpose would be served by intersecting the initial state set of each system participating in the process with the process language associated with that system. So we see that all three components of Horning and Randell's processes are present, implicitly, in ours.

We acknowledge that our definition exists within a specific universe. Horning and Randell's definition can only exist within a class of universes conforming to their basic structural assumptions (ours, for instance, does not conform), and so we can say that it exists within a (meta-) universe, too. We claim that our definition has some superiority over theirs only because it exists in a richer universe, one better suited for studying systems problems.

Horning and Randell's representation-independent definition becomes tied to a representation by the definition of a processor, a pair (D,I) in which D is a physical device and

> I is an interpretation of its physical status which indicates at what instants of time, and by what means, the device represents successive states.

This leaves the user of their definition free to use any device, but places on him the overwhelming burden of finding appropriate interpretations, at the right level to be illuminating. In fact, since processes and processors can be in one-to-one correspondence, the problem of identifying processes is contained in the problem of finding interpretations--exactly the job we hope to delegate to syntactic analysis. Our definition requires that the device be a formally defined system complex, but it provides the designer with automatic interpreters at any desired level, and the key to process identification besides.

The state spaces of Horning and Randell's processes are of fixed size and dimensions, which is appropriate to the fact that they are already bound to a fixed physical device. Since our state space is not bound in any way, we allow it to grow and shrink via a much more flexible successor function.

One of the most important differences between our universe and theirs is that we allow the specification and use of any number of asynchronous state spaces, while they have only one state space. The effective difference is that variables in the same state space always change values synchronously, in discrete time steps, while variables in different spaces are mutually asynchronous. In the formal definition universe, each system corresponds to a state space. Horning and Randell do recognize the necessity of modeling asynchronous interactions, but their concessions to it are unconvincing: they combine asynchronous processes to make one process in one "state space," but then the variables of the state space are presumed to be mutually asynchronous--which is self-contradictory, and also discards the relevant information that many of the variables really are synchronous with respect to each other, since they belong to the same synchronous subprocess.

This brings up the problem that Horning and Randell's processes cannot communicate as ours do, and so the only way to describe the interaction of two processes is to define a containing process in which the subprocesses share variables, as above. It seems like a very bad idea to multiply the size of processes just to investigate their connections! Our goal is to factor the analysis of systems as much as possible, and we are convinced that Horning and Randell's approach is not practical. Undoubtedly it was not formulated with such algorithmic analysis in mind.

The final difference between our universe and theirs is that they completely avoid the issues of race conditions and memory interference by calling the result of any such situation undefined--without even giving the user of the processes a way to discover whether his results are undefined or not. In other words, general interaction of processes is not well defined. In our universe nothing is ever undefined. We prefer to face these situations and provide tools for structuring, observing, and controlling them, so that designers have a chance to solve some of the associated problems.

It would certainly be possible to describe a process in the style of Horning and Randell in the formal definition universe, and to recognize it as a process by our definition. It would have a rigid state structure delimiting a fixed number of variables, and no arcs crossing its boundary. We can also talk about processes that grow, disappear, reproduce, interact, etc.

We welcome the current work on process structuring being carried out using other universes, because the correspondence between our process definition and Horning and Randell's shows that it will be possible to apply results from one context to the other. In particular, the interest in processes is stimulating work, such as that on combination and cooperating of processes, which might become part of our future design tools.

REFERENCES

[1] Adams, Duane A. "A Model for Parallel Computations." Parallel Processor Systems, Technologies, and Applications, L. C. Hobbs, et. al., ed., New York: Spartan Books, 1970.

[2] Baer, J. L. "A Survey of Some Theoretical Aspects of Multiprocessing." Computing Surveys 5, no. 1, March 1973.

[3] Belpaire, Gerald, and Wilmotte, Jean Pierre. "Correctness of Realizations of Levels of Abstraction in Operating Systems." Proceedings of the International Symposium on Operating Systems, Theory and Practice, Rocquencourt, 1974.

[4] Berry, Daniel M. "The Equivalence of Models of Tasking." SIGPLAN Notices 7, no. 1, January 1972.

[5] Bradshaw, F. T. "Some Structural Ideas for Computer Systems." COMPCON 72 Digest of Papers, IEEE Computer Society, 1972.

[6] Dijkstra, Edsger W. "The Structure of the "THE"-Multiprogramming System." CACM 11, no. 5, May 1968.

[7] Elspas, Bernard, et. al. "An Assessment of Techniques for Proving Program Correctness." Computing Surveys 4, no. 2, June 1972.

[8] 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.

[9] Gilbert, Philip, and Chandler, W. J. "Interference Between Communicating Parallel Processes." CACM 15, no. 6, June 1972.

[10] Glaser, E. L. "Introduction and Overview of the LOGOS Project." COMPCON 72 Digest of Papers, IEEE Computer Society, 1972.

[11] Glaser, E. L. "LOGOS--Where It Is Now and Where It Is Going." COMPCON 72 Digest of Papers, IEEE Computer Society, 1972.

[12] Heath, F. G., and Rose, C. W. "The Case for Integrated Hardware/Software Design, with CAD Implications." COMPCON 72 Digest of Papers, IEEE Computer Society, 1972.

[13] Hoare, C. A. R. "An Axiomatic Approach to Computer Programming." CACM 12, no. 10, October 1969.

[14] Holt, Anatol W., and Commoner, Frederick. "Events and Conditions." Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, 1970.

[15] Holt, Richard C. "Some Deadlock Properties of Computer Systems." Computing Surveys 4, no. 3, September 1972.

[16] Horning, J. J., and Randell, B. "Process Structuring." Computing Surveys 5, no. 1, March 1973.

[17] Johnson, Robert T. Proving Assertions About the State Structure of Formally-Defined, Interacting Digital Systems. Ph.D. Thesis, University of Wisconsin, Madison, Wisconsin, 1973.

[18] Karp, R. M., and Miller, R. E. "Parallel Program Schemata." Journal of Computer and System Sciences 3, no. 2, May 1969.

[19] Keller, Robert M. "Parallel Program Schemata and Maximal Parallelism I: Fundamental Results." JACM 20, no. 3, July 1973.

[20] Luckham, D. C., Park, D. M. R., and Paterson, M. S. "On Formalised Computer Programs." Journal of Computer and System Sciences 4, no. 3, June 1970.

[21] Manna, Z. "The Correctness of Programs." Journal of Computer and System Sciences 3, no. 2, May 1969.

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

[23] Miller, Raymond E. "A Comparison of Some Theoretical Models of Parallel Computation." IEEE Transactions on Computers c-22, no. 8, August 1973.

[24] Rose, C. W., Bradshaw, F. T., Katzke, S. W. "The LOGOS Representation System." COMPCON 72 Digest of Papers, IEEE Computer Society, 1972.

[25] 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.

[26] Smith, Pamela Z., and Fitzwater, D. R. "Efficient Analysis of the Process Structures of Formally Defined Complexes of Interacting Digital Systems." Computer Sciences Technical Report to be published, University of Wisconsin, Madison, Wisconsin, 1974.

[27] Stockmeyer, L. J., and Meyer, A. R. "Word Problems Requiring Exponential Time: Preliminary Report." Proceedings of the Fifth Annual ACM Symposium on the Theory of Computing, 1973.

[28] Wegner, Peter. "Operational Semantics of Programming Languages." SIGPLAN Notices 7, no. 1, January 1972.

[29] Wegner, Peter. "The Vienna Definition Language." Computing Surveys 4, no. 1, March 1972.