WIS-CS-184-73
Computer Sciences Department
The University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

Received June 13, 1973

A FORMAL DEFINITION UNIVERSE FOR COMPLEXES OF INTERACTING DIGITAL SYSTEMS

by

D. R. Fitzwater
and
Pamela Z. Smith

Computer Sciences Technical Report #184

June 1973

A FORMAL DEFINITION UNIVERSE FOR COMPLEXES OF INTERACTING DIGITAL SYSTEMS

by

D. R. Fitzwater
and
Pamela Z. Smith

ABSTRACT

A language is presented for describing any asynchronous complex of interacting digital systems at any level of abstraction. The resulting representation is a definition of the complex of systems which is both formal, and effective, meaning that it is interpretable by a deterministic automaton. Thus it can be used to study system structures and develop practical design tools.

After the explanation of the formal definition universe, annotated examples of system representations are given. Their usefulness for proving assertions about systems is established in an outline of present and future research, with the ultimate goal of providing assistance for the total design process.

A FORMAL DEFINITION UNIVERSE FOR COMPLEXES OF INTERACTING DIGITAL SYSTEMS

I. INTRODUCTION

A. The Need for a Formal Universe

Each of the physical sciences hypothesizes a particular abstraction of the real universe, which is the theoretical universe of the science. Even though conclusions are validated by measurements on the real universe, progress of the science consists of contributions to the theoretical one.

If computer science has not achieved the status of the physical sciences, it is because there is no general theoretical universe--each research effort is carried out in the arbitrary environment of a particular machine architecture, with the vocabulary of problem and solution being defined by that machine. With no common language in which to define systems, we cannot study their general properties: we cannot achieve inter-system compatibility, and we cannot avoid the continual process of re-inventing problem solutions for each new machine that comes along.

A common universe of systems would provide us with a means to define them formally, study--and prove--their properties, and develop design tools that would be useful in any practical context. We could communicate our ideas in a form as familiar to our listeners as it is to us. Finally, we might free our imaginations from the bounds of existing ways of doing things, which are derived from assumptions about computers past and present, so that we could design for computers of the future.

In short, a formal universe of systems is needed to elevate our field to a true design science. The concepts involved are discussed in a fascinating book by Herbert A. Simon, _The Sciences of the Artificial_ [SI].

It is certainly true that formal schemes for describing computation have been devised before, and that they have proven useful. None of them, however, has all the properties that we feel to be essential to the making of a systems universe.

B. The Essential Properties of a Formal Universe

1. Generality

We must be able to define, in our formal universe, all digital systems of interest; the term "digital" refers to a system that enters a sequence of well-defined, observable states, separated by transition periods of finite duration.

Clearly we must be able to model complexes of interacting asynchronous digital systems. Without this ability we cannot write a valid description of a single third-generation computer, let alone a cross-country network. Basic as this property of a formal definition scheme is, it is the grounds for rejection of nearly every existing candidate.

2. An Interpretable Representation

Much effort has gone into specifying procedures in formal representation-independent notations, so that they could be implemented exactly on different computers. Real system processes, however, consist of manipulations of a representation, and they are always representation-dependent. The system designer has the very real problem of choosing a suitable representation and defining efficient transformations on it.

Representation-independent definitions may lead to some theorems about systems, but since they are not effective (meaning that a mechanical interpreter can "run" the system), they provide no feedback to the designer as to the consequences of his design decisions. The designer may learn that his system has or doesn't have a few special properties, but to learn anything else about it, he has to implement it.

For these reasons, a formal universe must be based on a primitive automaton which interprets representations of complexes of systems specified in a description language. If the representation medium is abstract, it should still be possible to implement it exactly on different computers.

3. The Right Level of Abstraction

The choice of an abstract representation medium is an important one, because inessential detail will obscure concepts, but too much abstraction will deprive us of the ability to study what is interesting. We can take the Turing machine as a bad example. It is too abstract in the sense that it eliminates time and bounded space, yet the transformations that the automaton makes on its representation medium are excruciatingly primitive.

There is also another kind of transformation with which we must be concerned. A given system should have many equivalent representations in the formal universe, each one describing it at a different level of abstraction. Much of the informal design process is concerned with transformation between these equivalent representations, and we can hope to automate--or at least study and aid--this difficult procedure, if and only if our single abstract representation medium is such that these transformations are facilitated.

Some possibilities for the representation medium are integers (via Gödel numbers [MI]), strings, and graphs. Gödel numbers and bit strings are like the tape of a Turing machine in that simple operations require elaborate coding and decoding. This blocks structural insights, and makes design changes difficult. Graph representations certainly allow general objects and transformations, but they are rather far in advance of current linear computer memories. The generality of graphs introduces complications of its own, and it may be that a full graph-structure representation medium would be as unbalanced a choice as Gödel numbers.

Fortunately, the string representation is a happy medium. Its practicality is approved by current technology, and yet it can be used to represent objects as complex as trees. When we base our transformations of the medium on the concept of pattern-matching, we are choosing an operation that is easy for people, but difficult (or "high-level") for machines. This is exactly the property we must have in an effective research and design tool.

4. Neutrality

Since the formal universe is to be used to study possible system structures, it must not impose structure on the systems defined in it, or even bias them toward any structure. A good example can be found in control structures. The only way to avoid biasing our study of them is to insist that, in the formal universe, all possible transformations must occur simultaneously--unless the system designer has defined explicit constraints to prevent this. Thus all degrees of parallelism and sequentiality are available to him, which would not be so if the the transformations in the formal universe were predetermined to be sequential.

C. Desirable Properties of a Formal Universe

One of the most important criteria on which system designs are judged is their efficiency. Because the cost of a design is ultimately dependent on the implementation (and the computer), we can make no direct requirements of the formal universe concerning this, but it seems important that there be some useful relationship between a formally-defined system and the efficiency of its implementation.

For instance, we would hope to find a correlation between the time it takes to interpret a formally-defined system, and the time consumed by an implementation of that structure. Most promising is the idea of finding bounds on formally-defined systems that have some validity for common implementations, and then exploring the ways in which design principles affect these bounds.

Ideally, a formal universe should be such that the simplest and most elegant systems have the most efficient implementations.

D. Presenting ABSTRACT

We have devised a formal universe which consists of a primitive automaton to "run" system representations, and a description language in which to specify system representations. The present version is the result of several years of use, which revealed the need for minor improvements over the original scheme [FH]. Strictly speaking, ABSTRACT is the name of a program (written in Fortran V on a Univac 1108, but now being made, we hope, portable) which implements the primitive automaton, but it may serve as a name for the primitive automaton itself, or the entire formal universe.

The starting point of this effort was with Post production systems [MI], a formalism beautifully suited to our needs in string manipulation. Because we value this bridge between our efforts and previous formal work, we wanted to change the characteristics of Post's systems as little as possible. Two major extensions were inevitable: (1) the formalism had to be extended to include such "real system" concepts as time, space, and inter-system communication, and (2) transformations could no longer be actions that were permitted to happen; they had to be actions that would definitely be carried out, deterministically, by the primitive automaton. Nevertheless, the formal universe contains all Post systems, and a Post system so expressed retains all its observable properties--which is to say that the formal universe is a true extension of Post systems, specifying only additional properties that are irrelevant to the original formalism.

What are the credentials of our own scheme as a formal system? This is a good question, because our subsequent description will be highly informal. It has been shown, however, by Robert T. Johnson [JO], that the transformations made by the primitive automaton on a system representation can be defined rigorously in terms of functional notation. We feel that this would not be a good way to present the concepts, but it does establish our right to call ABSTRACT a formal universe.

We believe that this formal universe has all the essential properties mentioned, and, hopefully, the desirable ones also. It is offered as a hypothesis of our science--but our only claim is that it is the best hypothesis we have yet.

II. THE FORMAL UNIVERSE AND ITS DESCRIPTION LANGUAGE

A. Basic System Representations

A system representation (henceforth abbreviated SR) has three parts: (1) a set of process states, (2) an alphabet, and (3) a set of productions. Each process state is a character string, and the set of them represents the observable state of the system--the set of process states is the only changeable part of an SR. The alphabet specifies the set of characters which can be matched by a variable in a pattern. The productions are rules by which the primitive automaton will transform the process states. Thus the overall appearance of an SR is:

{$\underline{\sigma:}$ <process state set> $\underline{\chi:}$ <alphabet> $\underline{\pi:}$ <production set>}

It may be worth noting that all the special characters we use, e.g., "{" and "$\underline{\sigma:}$", are arbitrary choices. A more formal treatment might use a vocabulary of abstract terminal symbols and then define our characters as print equivalents. As long as we understand that we can do this, there seems to be no reason for actually doing it.

The process states (separated by "<u>and</u>") which are written in the SR form the initial process state set, also called the σ-set or just σ. Once the system goes into execution, σ may never again be the same as the initial set. The null string is a valid process state, and is always recognized by context. For instance,

{$\underline{\sigma:}$ START <u>and</u> $\underline{\chi:}$ ...

shows a σ with two process states, "START" and the null string. If "$\underline{\sigma:}$" is immediately followed by "$\underline{\chi:}$", σ consists of just the null string; we know that an empty initial σ-set was not intended because (as we shall see) an SR with an empty σ-set has halted forever.

The alphabet is just a string of characters, with no delimiters between them. Characters can be used in a process state without appearing in the alphabet.

The productions in the production set, or π, are separated by "<u>or</u>". A simple production might have the form:

$$a_0 \ \$ \ a_1 \ \$ \ a_2 \rightarrow b_0 \ \$_2 \ \$_2 \ b_1 \ \$_1$$

where the a's and b's stand for literal strings. Each "\$" in the antecedent (the part to the left of "$\rightarrow$") is a pattern which matches the null string or any string consisting only of characters in the alphabet. Each literal string in the antecedent is a pattern which matches only itself. Each "$\$_i$" in the consequent (the part to the right of "$\rightarrow$") represents the string which was matched by the ith "\$" in the antecedent. There are no restrictions on the number or sequence of variables or literal strings in antecedents or consequents.

Since σ, π, and the alphabet are sets, the order in which their elements are given is irrelevant. SR's which differ only in the ordering of set elements are formally equivalent.

The finite interval of time between two adjacent time spans that the system is in a discrete, observable state, is called a system step, and it is during this time that the primitive automaton transforms σ. Thus it produces a sequence of σ-sets or system states. The meaning of a production is this: For every different way in which the antecedent can match a member of the σ-set with which the system step begins, a process state goes into the next σ. It is formed according to the consequent, with literal strings copied and "$\$_i$"'s replaced by the substrings matched by the corresponding "\$". Each σ contains only those process states formed from the application of productions on the preceding system step. Otherwise a process state, once in the set, would have to stay there forever. If σ becomes empty, it is not possible that the system could ever change state again (how could a production be applied, with nothing for its antecedent to match?) and so it halts.

It is very important that, during a system step, the results are put in a buffer called σ' ; at the end of the step σ' becomes the new σ. This means that there is no interaction between the results being generated and the σ they are being generated from: all productions are applied simultaneously to all the process states in σ that their antecedents can match in some way, but the system state does not change until the end of the step.

A flowchart (Figure 1.) may clarify all this.

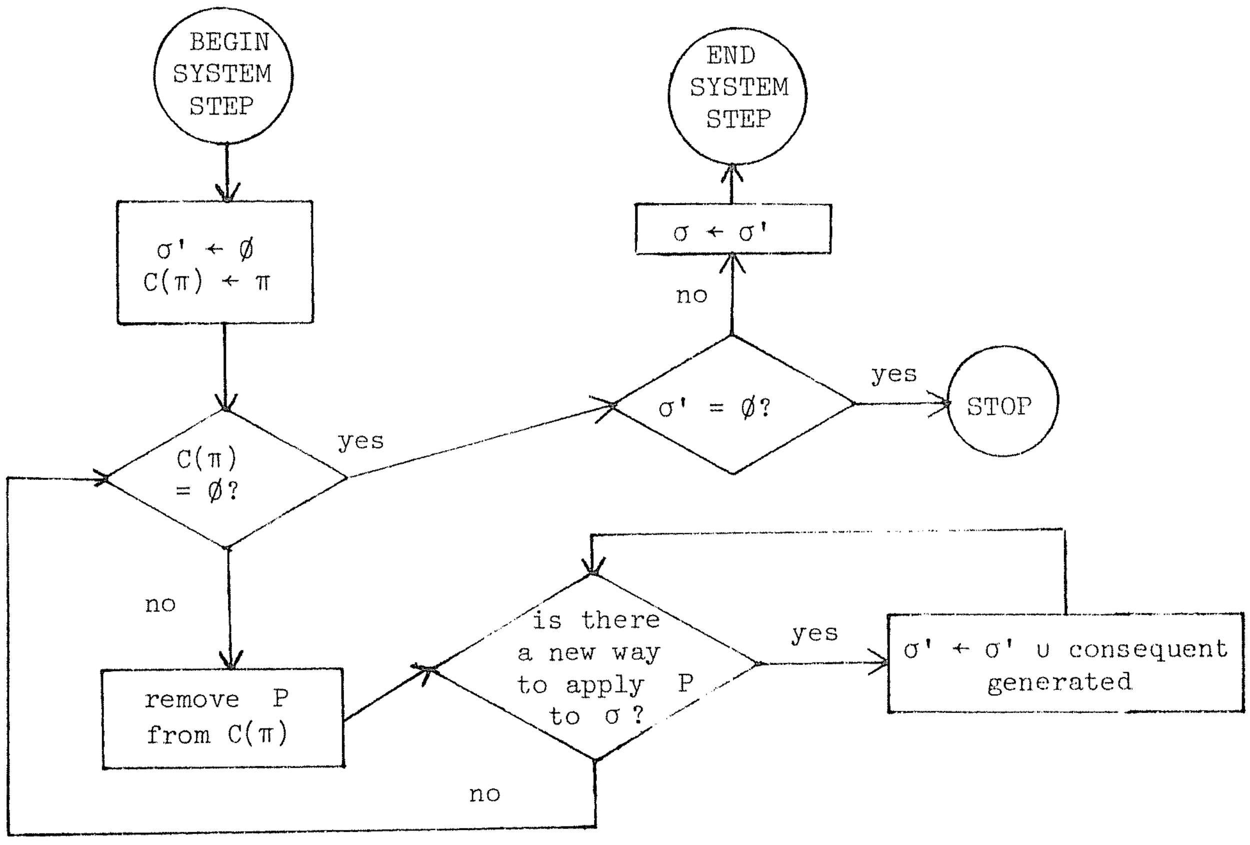


Figure 1.

This flowchart represents an algorithm for simulating what it is not possible for any conventional computer to do, i.e., apply all productions in all ways simultaneously. It is possible to do the computation sequentially and get the same result simply because σ' is kept strictly separate from σ. At the beginning of a step σ' is initialized to the empty set (denoted by "$\emptyset$"), and a consumable copy of π, called $C(\pi)$, is made. "Remove P from $C(\pi)$" indicates the random selection of a production from $C(\pi)$; next there is a loop in which P is matched to the process states in σ in all possible ways. After leaving this loop we go back to test if $C(\pi)$ is exhausted, which would mean that all productions have been tried. If they have not, we select another. If they have, we check for a halting condition, update σ, and end the step.

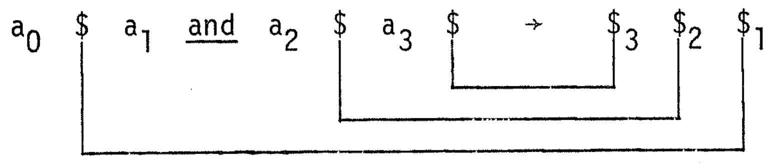
As a simple example we have:

{$\underline{\sigma:}$ STATE0 <u>and</u> THIS-IS-SYSTEM-NUMBER-1
$\underline{\chi:}$ ABCDEFGHIJKLMNOPWRSTUVWXYZ
$\underline{\pi:}$ $\$0 \rightarrow \$_1 1$ <u>or</u>
$\$1 \rightarrow \$_1 0$ }

All that this system does is keep track of whether it has gone through an even or odd number of system steps, as represented by a 0 or 1 appended to the "STATE" process state. At the end of the first system step, σ will have only "STATE1" in it, since the other process state cannot match any antecedent ($ cannot match a string with "-" in it); thereafter σ will alternate between "STATE0" and "STATE1".

Now we can explain some additional features. In the antecedent, a character with a bar over it, e.g., "$\overline{\Delta}$", is a pattern that matches one character different from the given character. This could be just a shorthand for a great number of slightly differing antecedents, but we want the ability to recognize a non-match built into the primitive automaton, for the sake of efficiency. In the consequent, each "not-character" is treated as a different kind of pattern, so that "$\overline{\Delta}_i$" calls for the use of the character that matched the ith occurrence of the particular pattern "$\overline{\Delta}$", and not just any "not-character" pattern. Another such convenience is that a "\$" in the antecedent may have a subscript--indicating the maximum length of the character string this pattern can match. These subscripts have no effect on indexing in the consequent.

Most important of all is that a production can have one <u>or more</u> antecedents (separated by "<u>and</u>"). This allows interaction between processes and a much more powerful, flexible modeling of real systems than would otherwise be possible. A production is used to generate a consequent only when all its antecedents match process states in σ (they could all match the same one, of course). In a multiple-antecedent production, indexing in the consequent refers to the patterns in all the antecedents in left-to-right order:


$a_0 \; \$ \; a_1 \; \underline{\text{and}} \; a_2 \; \$ \; a_3 \; \$ \; \rightarrow \; \$_3 \; \$_2 \; \$_1$

thus we have drawn the lines to show correspondence in variables between antecedents and consequent. Null strings are valid antecedents or consequents; once again, they are recognized by context.

B. Inter-System Communication

A system in isolation is not good enough for us. We want to observe it, we want to make it interact with other systems. For these purposes a message facility was designed.

The collection of all existing systems (in the frame of reference of each user of ABSTRACT) is called the system complex. Each system in the complex runs on its own time scale and is completely asynchronous with respect to all other systems in the complex--it is never possible to make any assumptions about the relative rates of any two systems. The system designer must always keep in mind, when he is planning inter-system communication, the indeterminacy of timing to be contended with; the message facility, however, is sufficient for him to enforce any cooperation that is necessary.

A production may have the form

$$\langle\text{antecedents}\rangle \rightarrow \langle\text{consequent}\rangle \rightarrow \langle\text{consequent}\rangle$$

where the first consequent is a pattern for the generation of a message, and the second consequent is a pattern for the generation of the name of the channel on which it is to be sent. During the system step, the (channel name, message) pairs generated are put in a buffer called β instead of in σ'.

At the end of a system step, the primitive automaton gathers all the pairs in β which have the same channel name into sets, so that we now have (channel name, message set) pairs. Each system has a message-receiving buffer called σ''. The pairs are now transmitted to the σ''-sets of every system in the complex (including the σ'' of the transmitting system), and β is reset to $\emptyset$. This is how messages are sent, once every system step.

The actual transmission of messages must be regarded as a process with some unknown duration. It can also be different for each σ'' to which messages are transmitted. The only certainty is that messages a system sends to itself arrive instantaneously. Therefore, even if we observed a complex of systems relative to a global clock, we could not make any assumptions about the arrival times of messages sent at global time t at the various σ'''s, except that each arrival time would be some number $a \geq t$ on the global clock (and would be equal to t at the sending system).

Each system also accepts its messages once every system step--in fact, accepting messages is the very last thing that the primitive automaton does before completing a system step. This operation is called updating σ from σ'', because the messages will change the contents of σ. First, a consumable copy of σ'' is made, called $C(\sigma'')$, and the buffer is set to $\emptyset$ so that it can begin receiving messages afresh. The seizure of the contents of σ'' is defined to take place at one instant in time, so that σ'' is never unavailable to receive messages. This is not unreasonable, since a real implementation could produce the same logical effect, and it is necessary to prevent message loss. Thus each $C(\sigma'')$ contains all the messages that were received by this system between this seizure of σ'' and the preceding one--with an important exception.

If a set of messages is received under a certain channel name, and σ'' already contains a set of messages sent on that channel, then the previous set of messages is over-written and therefore disappears. This makes it possible to model real-time computations. We assume that there is a conflict-resolver in the mechanism of each σ'', so that it is always clear which of the two message sets arrived first.

Having made $C(\sigma'')$, the primitive automaton updates σ as follows: if a channel name in $C(\sigma'')$ matches a process state in σ exactly, then that process state is replaced by the set of messages associated with the channel name. Messages sent on channels that do not appear in σ are discarded. To carry out this updating, we use an algorithm that flags the original members of σ. Then the channel names can be compared to the process states sequentially, instead of simultaneously, without any risk of confusing a newly-accepted message with an original process state. This is shown on the flowchart of the primitive automaton in Section II.E.

It may seem strange to accept messages at the end of a system step instead of at the beginning. Consider, however, a system which sends a message to another system eliciting a response, and at the same time sends itself a message which is the channel on which the response is to be received. If the other system is very quick, and sends the response before the originating system begins its next step, and if the originating system accepts its messages just before starting a step, then the message to itself will be accepted at the same time as the response (in the same $C(\sigma'')$). Since the channel to receive the response will not be in σ, the response will be lost. In other words, systems accept messages at the end of their steps so that they can communicate to themselves "instantaneously" or "faster than any other system."

We mentioned that the message facility could be used to observe a system. As an example of what was meant, our implementation of the primitive automaton includes in every system complex two informally defined systems, READS and PRINTS, to serve as interfaces between formally defined systems and the environment. A message sent on the channel PRINT is accepted by PRINTS and printed. A message sent on the channel READ is accepted by READS, which reads a message from the input medium, and sends it out on the channel supplied in the message it was sent.

C. Restricted Processors

We consider the concept of a system step to be an important one, because it is the basic measure of time in a system, and ought to be used to create optimum patterns of synchronization and parallelism between systems. To do this, however, it is necessary to provide for system steps of arbitrary complexity. For this reason we introduce restricted processors, which are special invariant systems residing in antecedents as patterns to be matched, and which can be used to perform computational details where they will not be observable outside the system. They are also useful for writing incomplete SR's (an algorithm buried in an RPR can be written later) so that the designer's tasks can be factored conveniently.

A restricted processor representation (abbreviated RPR) appears in an antecedent, and looks exactly like an SR except that, instead of an initial set of process states, it has a single, fixed pattern. This is why we say than an RPR is an invariant system.

When the production containing an RPR is being applied to a process state, the pattern of the RPR functions almost exactly as it would if it were just part of the antecedent, with no RPR attached (except that it cannot usually contain RPR's itself). Indexing of variables in the consequents is not affected by the fact that the pattern segment is in an RPR. The only difference is that the alphabet used to determine which characters a variable can match is that of the RPR rather than that of the SR.

In fact, one is free to attach any alphabet to any variable in an antecedent. The default alphabet is that of the immediately surrounding SR or RPR, but if that is not satisfactory, one can change it by surrounding a particular variable with an RPR that has no productions:

$$\ldots\ XYZ\ \{\$\ \underline{\chi:}\ ABC\ \underline{\pi:}\}\ XYZ \rightarrow \ldots$$

So that the same thing can be done to variables in RPR patterns, we allow RPR's with no productions to appear in patterns.

Between the time that an antecedent containing an RPR matches a process state, and the time that the consequents of the match are formed, the RPR is executed. This "minor cycle" of the primitive automaton is a whole computation, which may require many steps, embedded within the step of the SR--and because it is embedded, its steps are not directly observable outside the SR. An RPR cannot receive messages, and although it can send them, all messages generated by the RPR are put in β of the SR, and not transmitted until the end of the SR's system step.

Now we will describe the execution of an RPR. First the substring of the process state which matched the RPR's pattern is used to create an initial σ with a single member. Then the primitive automaton begins to compute successive steps. There are two main differences between the step-by-step computation of an RPR and an SR: (1) all members of the σ of an RPR must match its pattern, and (2) we believe that when an RPR generates a process state that matches the RPR pattern but no antecedents in the RPR, it is meant as a legitimate result of the RPR, and so it will be saved (in contrast to a process state of an SR, which always disappears in the time of one system step if there is no antecedent that matches it).

What happens when a consequent generates a process state that doesn't match the pattern? This is one of the two ways that an RPR can halt--the other is by having an empty σ' , just like an SR. Remember that each RPR execution should halt, or the surrounding SR will have an infinitely long system step (bad design!), and so this generation of a maverick process state is likely to prove useful. If it happens, σ is updated from σ' minus any process states that don't match the pattern, and σ is returned as part of the result set of the RPR.

To find and save all RPR results that are generated before the RPR halts, we must flag each process state in σ at the beginning of each RPR step. When a process state is used in the successful application of a production (or even when it matches an antecedent of a multiple-antecedent production, but the production is not applied because the other antecedents cannot find matches), its flag is removed, so that at the end of the step all those process states that are still flagged must be saved. We have a buffer called τ to hold them. The contents of τ are included in the result set returned when the RPR halts.

When the consequents of the production containing the RPR are generated, a different consequent is formed for each result of the RPR, i.e., for each distinct correspondence of variables in the consequent to substrings in RPR results. The value of a consequent variable which refers to an antecedent variable in an RPR pattern, is determined by the match between the RPR pattern and the RPR result (which may be quite different from the match between the RPR pattern and the initial σ of the RPR). If an RPR has an empty result set, no consequents to the production are generated, even if its consequent pattern has no variables.

AN RPR can appear in the antecedent of a production in an RPR, and this nesting can go on to any depth. Obviously, execution of the various RPR's will be nested accordingly.

It only remains to mention that an antecedent can contain any number of RPR's. The consequents generated from such an antecedent must use every possible combination of results from the several result sets. For example, if a production

$$* \langle RPR_1 \rangle \quad * \quad \langle RPR_2 \rangle \quad * \quad \rightarrow \quad \$_1 \ \$_2$$

matches a process state (the pattern of each RPR is "$") and produces the result set {A,B} for RPR_1 and {X,Y} for RPR_2, the consequents generated will be {AX, AY, BX, BY} .

A typical situation where one might want to use an RPR is the incrementation of a counter. This little bit of arithmetic is obviously subordinate to the process which contains it, and yet it might take several steps, during which the whole system could be needlessly delayed. The following RPR will increment a binary number by 1 each time it is matched in the application of a production:

$$\{\$ \,.\, \$ \ \underline{\chi:}\ 01\ \underline{\pi:}\ \$\ 0\ .\ \$ \rightarrow .\ \$_1\ 1\ \$_2\ \ \underline{\text{or}}$$
$$\$\ \overline{\Delta}\ 1\ .\ \$ \rightarrow \$\ \overline{\Delta}_1\ .\ 0\ \$_2\ \ \underline{\text{or}}$$
$$1\ .\ \$ \rightarrow .\ 1\ 0\ \$_1\ \}$$

Note that the "." is a marker necessary in the computation. When the RPR pattern is first matched the binary number to be incremented will be on the left of the ".", and when incrementation is complete it will be on the right (so no more antecedents can match, and the RPR halts). Also note that the $\overline{\Delta}$ is used as a pattern which must match one character, hence not the null string. If we abbreviate this RPR by <inc>, we can see how it might fit into a clock system:

$$\{\underline{\sigma:}\ \ 0\ .\quad \underline{\chi:}\ 0\ 1$$
$$\underline{\pi:}\ \ \$\ . \rightarrow \text{time}\ \ \underline{\text{or}}$$
$$\text{<inc>} \rightarrow \$_2\ . \rightarrow \text{time}\}$$

This system keeps a binary number which it increments once each time it takes a system step. Any system in the complex interested in knowing this value need only put the process state "time" in its σ-set.

D. A Grammar for System Representations

Now that we have presented all elements of the description language for SR's, we give for reference an almost-context-free grammar of it. The only reason it is not really context-free is that the language generated is not completely linear--subscripts and bars over not-characters have been incorporated.

<SR>	::=	{σ: <process state set> <processor>}
<process state set>	::=	<process state>
	::=	<process state set> $\underline{\text{and}}$ <process state>
<process state>	::=	
	::=	<process state> <char>
<char>	::=	any non-special character
<processor>	::=	$\underline{\chi:}$ <alphabet> $\underline{\pi:}$ <production set>
<alphabet>	::=	
	::=	<alphabet> <char>
<production set>	::=	<production>
	::=	<production set> $\underline{\text{or}}$ <production>
<production>	::=	<antecedents> $\rightarrow$ <consequent>
	::=	<antecedents> $\rightarrow$ <consequent> $\rightarrow$ <consequent>
<antecedents>	::=	<ante>
	::=	<antecedents> $\underline{\text{and}}$ <ante>
<ante>	::=	
	::=	<ante> <antechar>
<antechar>	::=	<char>
	::=	<variable>
	::=	<RPR>
<variable>	::=	$
	::=	$ <subscript>
	::=	<$\overline{\text{char}}$>
<subscript>	::=	<non-zero digit>
	::=	<subscript> <digit>
<non-zero digit>	::=	$1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
<digit>	::=	$0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
<RPR>	::=	{<pattern> <processor>}

<pattern>	::=	
	::=	<pattern> <patchar>
<patchar>	::=	<char>
	::=	<variable>
	::=	<null-RPR>
<null-RPR>	::=	{<variable> $\underline{\chi:}$ <alphabet> $\underline{\pi:}$}
<consequent>	::=	
	::=	<consequent> <conschar>
<conschar>	::=	<char>
	::=	$ <subscript>
	::=	<$\overline{\text{char}}$> <subscript>

E. A Flowchart of the Primitive Automaton

We present a flowchart (Figure 2.) summarizing the actions of the primitive automaton while computing a system step on an SR. Most of the chart should be satisfactorily explained by the previous text, but we will make some comments on it here.

There are several references to an object called STACK. Because of the necessity for nesting (to an arbitrary depth) system steps of RPR's within this step of the SR, the algorithm will be entered recursively. The variables σ , σ' , χ , π , $C(\pi)$, P , β , β_P (an auxiliary message buffer used to collect (channel name, message) pairs generated by RPR's while RPR execution is still going on), τ , and SR (a Boolean variable: true if computation on the SR is going on, false if an RPR is being executed) are local to each call on the algorithm, and must therefore be kept in a stack. "Push STACK" means "put a new entry, including all the local variables, on top of the stack," and "pop STACK" means to remove the entire top entry.

It is assumed that new entries on the stack are initialized to $\emptyset$. σ , χ , and π are stack entries because they can be superseded by the σ , χ , and π of an RPR, but they are also global characterizations of the SR which must be saved between system steps. For this reason we initialize the stack entries to the saved values on entering the algorithm, and save the values on exit. If exit from the algorithm occurs via "STOP," these values are not saved because there will be no further use for them. S and σ'' are global variables which continue to belong to the SR both during and between system steps.

The "remove" primitive, as in "remove P from $C(\pi)$," means to select a production from $C(\pi)$ at random, and assign it to the temporary variable P . "Remove S from $C(\sigma'')$ works the same way. When the "yes" branch is taken from "is there a new way to apply P to σ ?", we assume that the patterns in the antecedents of P have all been bound to matched substrings of process states.

The "transmit" primitive gathers (channel name, message) pairs into message sets before sending them, as previously described.

Remember that flags are just distinguishing marks on process states which can be added or erased at any time--they have no significance outside the portion of the algorithm they facilitate.

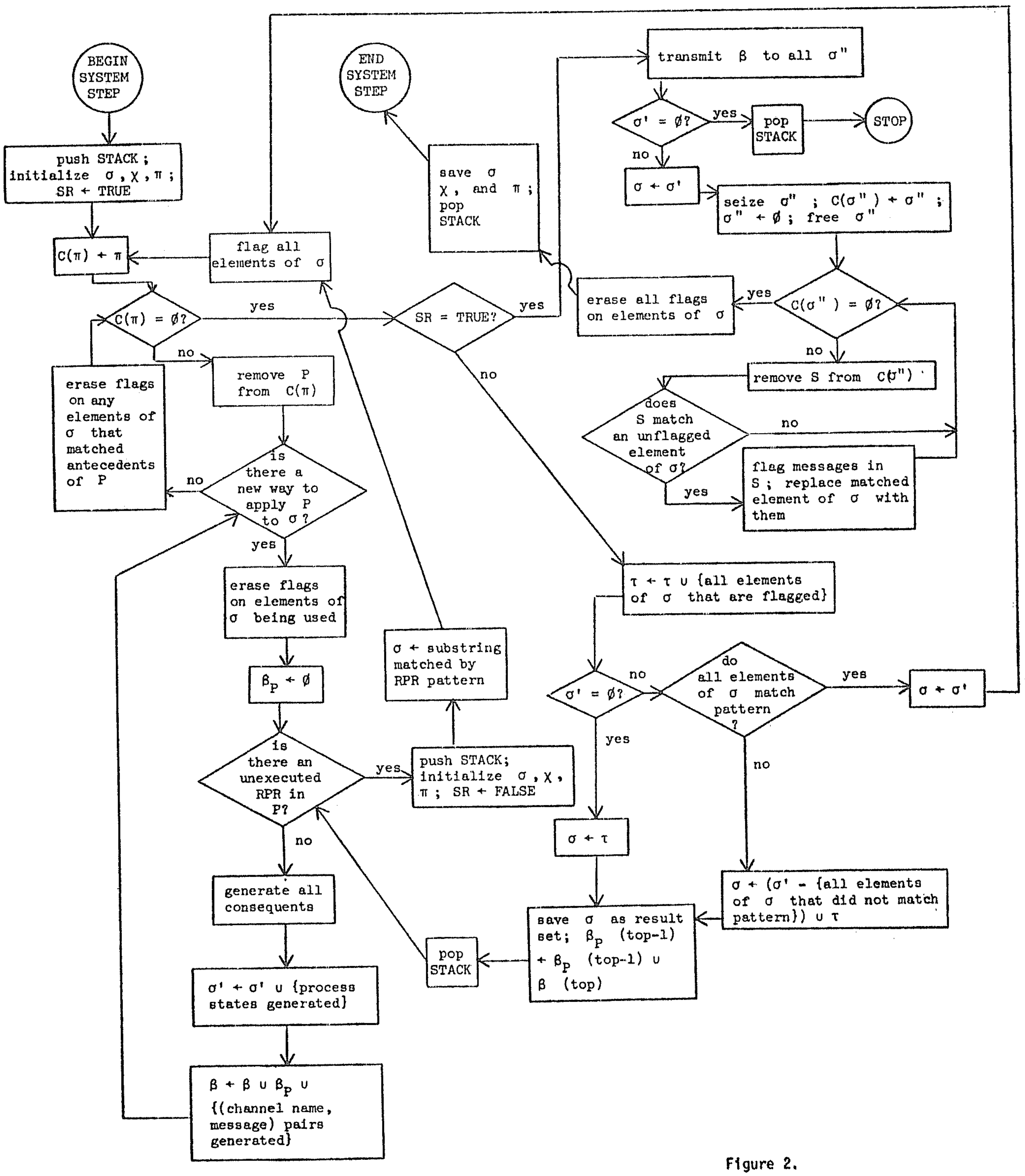

BEGIN SYSTEM STEP
push STACK ; initialize σ , χ , π ; SR ← TRUE
C(π) ← π
C(π) = ∅?
yes
no
erase flags on any elements of σ that matched antecedents of P
remove P from C(π)
is there a new way to apply P to σ ?
no
yes
erase flags on elements of σ being used
β_P ← ∅
is there an unexecuted RPR in P?
yes
no
generate all consequents
σ' ← σ' ∪ {process states generated}
β ← β ∪ β_P ∪ {(channel name, message) pairs generated}
push STACK; initialize σ , χ , π ; SR ← FALSE
σ ← substring matched by RPR pattern
flag all elements of σ
pop STACK
END SYSTEM STEP
save σ χ , and π ; pop STACK
SR = TRUE?
yes
no
transmit β to all σ"
σ' = ∅?
yes
no
pop STACK
STOP
σ ← σ'
seize σ" ; C(σ") ← σ" ; σ" ← ∅ ; free σ"
C(σ") = ∅?
yes
no
erase all flags on elements of σ
remove S from C(σ")
does S match an unflagged element of σ?
no
yes
flag messages in S ; replace matched element of σ with them
τ ← τ ∪ {all elements of σ that are flagged}
σ' = ∅?
no
yes
do all elements of σ match pattern ?
yes
no
σ ← σ'
σ ← τ
save σ as result set; β_P (top-1) ← β_P (top-1) ∪ β (top)
σ ← (σ' - {all elements of σ that did not match pattern}) ∪ τ

Figure 2.

III. EXAMPLES

A. A System Complex

We will illustrate how the formal definition universe can be used to describe a complex of systems which must synchronize some of their actions. Our complex has three systems. One is a parser for the simple precedence expression grammar

```
E  ::=  D
D  ::=  D + U | U
U  ::=  T
T  ::=  T * F | F
F  ::=  ( E ) | I .
```

This parser is supposed to be able to make each reduction within nine units of time, where the units are defined as cycles of a clock system. There is a third system which sends an initiating signal to the parser and the clock simultaneously, and monitors them to decide who won the race.

We hope that this example will point out that processes at all levels of abstraction are easily described and coordinated. We are presenting a binary clock (essentially a bit manipulator), a parser, and a high-level control process, all in the same notation. Here is the simple precedence parser:

$\{\underline{\sigma:}$ ready $\underline{\text{and}}$ $\vdash(I*I+I)*I*I*(I+I)\dashv$

$\underline{\chi:}$ $()*+\vdash\dashv$ DEFTUI

$\underline{\pi:}$ parse $\underline{\text{and}}$ $\$ \rightarrow \Delta\ \$_1$ $\underline{\text{or}}$ [1]

$\{\$\ \Delta\ \overline{\Delta}\ \$\ [RPR_1]\} \rightarrow \$_1\ \overline{\Delta}_1\ \Delta\ \$_2$ $\underline{\text{or}}$ [2]

$\$ \gtrdot \Delta\ \$ \rightarrow \$_1\ \Delta \gtrdot \$_2$ $\underline{\text{or}}$ [3]

$\{\$\ \overline{\Delta}\ \Delta\ \$ \gtrdot [RPR_2]\}\ \$ \rightarrow \$_1\ \Delta\ \overline{\Delta}_1\ \$_2 \gtrdot \$_3$ $\underline{\text{or}}$ [4]

$\$\{\Delta\ \$ \lessdot \$ \gtrdot [RPR_3]\}\ \$ \rightarrow :\$_1\ \$_2\ \$_4$ $\underline{\text{or}}$ [5]

$:\$ \rightarrow$ parsing step completed $\rightarrow$ receive $\underline{\text{or}}$ [6]

$:\$ \rightarrow \$_1$ $\underline{\text{or}}$ [7]

$:\$ \rightarrow$ ready $\underline{\text{or}}$ [8]

ready $\rightarrow$ ready $\underline{\text{or}}$ [9]

ready $\underline{\text{and}}$ $\$ \rightarrow \$_1$ $\}$ [10]

The numbers in [] are just line numbers, and not part of the SR. $[RPR_i]$ stands for the processor part of an RPR, which will be given later, for readability. We give the initial state of the parser as it would be just before starting on a new reduction: there is the process state "ready" and the string to be parsed (the delimiters are required by the parsing algorithm, not our notation; also, the restriction to one-character non-terminals in the string to be parsed is made only to keep the example short). When the control system sends the message "parse" on the "ready" channel, production [1] can be applied. Until then, [9] and [10] will cause the system to preserve its state as it cycles, waiting.

Once the string has a "Δ" marker in it, the parsing production [2] can be applied. The whole RPR_1 is:

$\{\$\ \Delta\ \overline{\Delta}\ \$$ $\underline{\chi:}$ $(\ )\ *\ +\ \vdash\dashv$ D E F T U $I\gtrdot$

$\underline{\pi:}$ $\$\ D\ \Delta\ )\ \$ \rightarrow \$_1\ D\ \Delta\gtrdot\ )\ \$_2$ $\underline{\text{or}}$ $\ldots.\}$

The RPR is essentially a table of the relation $\gtrdot$: the production shown belongs because $D \gtrdot)$, and the RPR contains a similar production for every pair of symbols i , j such that $i \gtrdot j$. The purpose of production [2] is to move the scan pointer "Δ" to the right, looking for the leftmost $\gtrdot$ relationship between two characters, which will define the end of the handle. If the production is applied when the "Δ" is between an i and j which do not have $i \gtrdot j$, the RPR does nothing, and the production moves the scan pointer one character to the right. If the scan pointer has reached the right place, however, the RPR will introduct the marker "$\gtrdot$" at the end of the handle, and the resultant consequent will have the scan pointer just to the right of the "$\gtrdot$" . Production [3] straightens that situation out by moving "Δ" to the left of "$\gtrdot$" , where it can begin to move leftward looking for the other end of the handle, via production [4].

In production [4], RPR_2 is:

$$\{\$\ \bar{\Delta}\ \Delta\ \$\ \gtrdot\ \underline{\chi:}\ (\ )\ *\ +\ \vdash\ \dashv\ D\ E\ F\ T\ U\ I\ \lessdot$$
$$\underline{\chi:}\ \$\ (\ \Delta\ D\ \$\ \gtrdot\ \rightarrow \$_1\ (\ \lessdot\ \Delta\ D\ \$_2\ \gtrdot\ \ \underline{or}\ \ \ldots.\}$$

It works much the same as production [2]. RPR_2 has a production for every pair of symbols i , j such that $i \lessdot j$, and if the scan pointer is between any such pair, the resulting consequent will have the form "$\$\ \Delta \lessdot$ handle $\gtrdot \$$" . Otherwise the pointer will just move one character to the left.

Production [5] actually makes the reduction. RPR_3 is:

$$\{\Delta\ \$\ \lessdot\ \ \$\ \gtrdot\ \ \underline{\chi:}\ (\ )\ *\ +\ \vdash\ \dashv\ DEFTUI$$
$$\underline{\pi:}\ \Delta\ \lessdot\ \ T * F \gtrdot \rightarrow \Delta\ T\ \lessdot\ \ T * F \gtrdot\ \ \underline{or}\ \ \ldots.\}$$

with one production for every reduction rule of the grammar. The string that the handle reduces to is placed between the "Δ" and the "$\lessdot$"; the consequent generated has only relevant parts--markers and the old handle disappear. The ":" is the signal that the reduction has been made, so a response is sent to the control system. Then the parser goes into a waiting state until the next "parse" command comes.

The clock system is similar to the one we showed before, except that it overflows on the eighth cycle (after one initialization cycle) and sends a message to the control system. It, like the parser, is initiated by the command "parse" coming in on the "ready" channel:

```
{σ:  ready  χ: 0 1
 π:  ready → ready                                         or
     parse → 0 0 0 .                                       or
     {$ . $  χ: 0 1
             π: $ 0 . $ → . $1 1 $2  or
                $ 1 . $ → $1 . 0 $2  or
                .000 → overflow } → $2 .                   or
     111. → timeout → receive                              or
     111. → ready}
```

Notice that the third production in the RPR generates a consequent not matching the pattern on the same system step that the clock starts at "111." and sends the "timeout" message. Because the result set of the RPR is empty, no consequents are generated. This clears the system of the clock register until it gets another "parse" command.

Finally we come to the control system:

```
{σ:  startparse  χ:
 π:  startparse → parse → ready                            or
     startparse → receive                                  or
     receive → receive                                     or
     timeout → parse failed → to whom it may concern       or
```

parsing step completed → receive <u>or</u>
parsing step completed → wait for clock <u>or</u>
receive <u>and</u> wait for clock → wait for clock <u>or</u>
timeout <u>and</u> wait for clock → parse → ready <u>or</u>
timeout <u>and</u> wait for clock → receive}

The system sends out the "parse" command, then puts itself in a receiving state and cycles until another system responds. The resolution time of this system, in judging the outcome of races, is its own system step time. Since this would be so regardless of whether the competing systems reported on the same channel or not (the system only looks at messages once every system step), here the competing systems use the same channel. This means that if both respond during the same step of the control system, the later one will actually win, by over-writing the other message--but if time units smaller than the system's resolution time were important to the designer, he should have made the resolution time smaller.

If the "timeout" message wins, a message that the parse failed will be broadcast. At the end of the step, σ' is empty, and the system dies. If the parse succeeded, however, another reduction is to be initiated. The control system waits for the response from the clock (otherwise a stray "timeout" message, generated after the parser finished, could ruin the next parsing step), then sends the "parse" command and again goes into the "receive" state.

B. Comments on "Programming"

One useful operation that may seem difficult to express in the formal definition system is comparison. For instance, suppose we had a process state "X <integer$_1$> Y <integer$_2$> Z" and we wanted something to happen if and only if the two integers were the same. Assuming the digits are in the alphabet, and that the action we want is to erase the integers, this can be accomplished by:

$$X \ \$ \ Y \ \$ \ Z \rightarrow X \ Y \ Z \rightarrow X \ \$_1 \ Y \ \$_1 \ Z$$

The message will not be accepted unless its channel name is present, which will be so only if the integers are equal.

Perhaps it does not seem right that one should need to know such tricks, when we claim that our formal definition universe is so abstract and general. Clearly the fact that some transformations are easier to make than others is the price you have to pay for having any representation at all. If we have chosen our representation well, the transformations that are hard to make will be the ones that are inherently complex from some meaningful point of view, and this seems to be the case.

Finally, it may seem that designing a large system in this unfamiliar notation would be hopeless. We hope we will convince you, in the next section, that it would be worthwhile. As to any programming difficulties, a design tool like this is exactly as much convenient as there is effort put into making it so. The addition of a macro processor to the ABSTRACT implementation, for instance, would be a tremendous help to the user, as would a library of RPR's. After enough features of that kind are added, any programming system becomes a pleasure to use.

IV. RESULTS, PRESENT AND PROMISED

A. Methods of Proving Assertions

We believe that this formal definition universe affords limitless possibilities for studying system structures. We will attempt to demonstrate this by presenting some of the methods which can be used to formulate and prove assertions about systems. The remainder of the paper will be a brief outline of current results and ideas for future research, indicating how we hope to apply the results to problems of system verification, system equivalence, implementation efficiency, and the iterative design process.

Syntactic analysis of an SR is only the first source of feedback to its designer, but we can offer him a good deal of information from it. The same type of syntax checking, formatting, and generation of diagnostics that is performed by any good compiler should be carried out; however, the simplicity of representation in an SR makes it possible to discover much useful execution information besides. For instance, by examining the pattern of an RPR and all the consequents inside it, we can decide whether the RPR can ever generate a process state which does not match the pattern. Another example, particularly important, is described in IV.B.1.

The next thing we can do with a complex of SR's is simulate it. This is just normal testing, using sets of trial data, but when the design is expressed in the formal universe it can be tested _before_ it is implemented--and at many levels of abstraction!

Exhaustive testing of any system design, implemented in any form, is seldom a finite procedure. Thus our most powerful tool for proving assertions about SR's is the one that gives us finite characterizations of the potentially infinite computations of an SR or SR complex.

The set of strings matched by any antecedent is a regular language, as is the set of strings which can be generated from any consequent. The regular language is an excellent characterization of a set of process states, finite or infinite--its descriptive power can be impressive. Most important, though, is that regular languages are the largest 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 on pairs of languages.

Because we have these algorithms, our assertions can be proved without any necessity for theorem provers based on unbounded search trees. It is true that operations on general regular expressions can grow exponentially, which could be worse than potential unboundedness, but the regular expressions one derives from an antecedent or consequent are greatly restricted. Computations on them are of reasonable size.

The essence of the testing problem is that a computation of a non-trivial system or complex is likely to consist of an infinite sequence of process state sets, a form of description not amenable to proof. If we can cast an assertion about the computation in the form of a statement about the regular languages the process states belong to, however, proof or disproof of the assertion will be algorithmic. It seems that the class of assertions that can be expressed in this way is very rich.

For the computation of such proofs, we have the R-model interpreter or simulator (as opposed to the F, or finite, model interpreter which has been described up to now). The R-model interpreter executes SR's which have regular languages instead of strings as their representation of state information. The result of the application of a production to a regular language L_1 is another regular language L_2 containing all the strings which would have been generated as consequents by the F-model interpreter applying that production if it were given all the strings in L_1 to start with. The R-model interpreter is described in detail, and its validity is proved, in Johnson's thesis [JO].

B. Some Early Results

1. The Antecedent Language Assertion

Examining the text of an SR provides a basic assertion about the control flow of processes in the system. Since each antecedent and consequent has a corresponding regular language, we can compute the intersections of all pairs of antecedent languages and consequent languages. A production P_1 can apply to a process state generated by the consequent of production P_2 only if the intersection of the antecedent language of P_1 and the consequent language of P_2 is not empty.

The assertion for a particular SR can be expressed as a directed graph in which each node is a regular language. An arc from node L_1 to node L_2 indicates that there is a production with an antecedent that can match a string in L_1 and a consequent that can produce a string in L_2. The assertion is "sequence-dependent" because it characterizes the sequences in which process states can appear during computation.

We are using this assertion to optimize F-model simulation. An analysis of the SR is carried out before execution begins, so that the number of unsuccessful attempts to match antecedents to process states can be reduced. This is a good example of the expected usefulness of such assertions. In the typical SR this optimization will eliminate most unsuccessful match attempts, leading us, hopefully, toward a simulation time which is fairly independent of the size of the SR. Also, since channel names and messages have their own regular languages, we can apply the same technique to inter-system communication.

2. The System State Language

A system designer may be interested in properties of the process state languages which hold throughout the entire history of the processes. A typical instance would be an assertion (a sequence-independent one) that, regardless of any transformations under productions, the process states are always contained in a given regular language which we call a system state language.

With the R-model interpreter, this assertion is proved about an isolated system very simply: we initialize the SR's σ-set to the asserted regular language, and apply the R-model interpreter for one step. Then we check to see if the resulting regular language is a sublanguage of the asserted language. If it is, we have proved the assertion, because no number of system steps will generate a process state outside the asserted language--the R-model interpreter is designed to guarantee this.

If the resulting language is not a sublanguage of the asserted language, then there are two possibilities: either the assertion is false, or there are some strings in the asserted language which cannot exist in any process state, but which caused the test to fail. In either case, the regular language chosen is lacking in some respects, and needs adjustment. Of course, if the assertion is false because the system can generate some process states that the designer did not expect and does not want, then the trial will have served as a valuable debugging aid.

3. The State Graph of a System Complex

We would like to extend the concept of a state language--now understood as a regular language containing all the process states in a system at one instant--to an asynchronous complex of communicating systems. The state of a complex of n systems could be represented by a vector of n regular languages such that the ith component was a state language for the ith system, containing its σ-set at the instant described by the vector.

Assume that we have a finite set of state languages for each system such that each possible state of the system is in one of these state languages, and that the amount of time that the state of any system is undefined (while σ is replaced by σ' as updated from σ'' is unobservably small. Then the state of the complex, at any instant, can be represented by one of these vectors, with all the system state languages in it selected from the finite sets mentioned above. Unless each finite set has only one member, there is more than one possible vector; they can be arranged as the nodes of a state graph whose arcs indicate successor relationships. The usefulness of this approach is suggested by the work of Gilbert and Chandler [GC], who use a state graph to analyze co-operating sequential processes.

Difficulties arise, in computing the successor relation, because the next state of the complex is dependent on which system completes a step next, <u>and</u> on the (unknown) transmission time of each message that is sent. This intrinsic indeterminacy is troublesome, but it can be overcome by associating with each state vector a vector of σ'''-sets, containing, for each system, all the messages sent to it minus the messages that have "arrived" by being received into σ'' . The details are worked out in [JO].

The resultant graph is called rate-dependent because it contains information about the relative rates of the systems in the complex. To specify the relative rates of all the systems and the transmission times of all the messages is to select a subgraph describing the action of the complex under those conditions. If only certain subgraphs are acceptable to the designer, he can formulate his conditions for correct functioning of the complex as a rate-dependent assertion about it, based on what he has seen in the state graph.

C. Finite Process Structures and System State Graphs

1. A Classification of Graphs

The directed graph whose nodes represent regular languages turns out to be an important tool for organizing information given by the R-model interpreter. Such a graph, made by analyzing an isolated SR or a complex of SR's, is in itself an assertion about the system--and a model of it.

There are a number of ways of forming such graphs; presumably each will prove useful in the study of some system characteristic or another. A major division is between graphs called finite process structures and system state graphs.

A finite process structure of an isolated SR consists of a collection of nodes representing different regular languages, such that the union of these languages contains the system state language. A directed arc from node L_1 to L_2 means that there is a single-antecedent production in the SR whose antecedent can match a string in L_1, and when it does, the consequent generated will belong to L_2. Multiple-antecedent productions require interaction between processes. An n-antecedent production thus generates n-arcs in the graph: an n-arc has n labeled components, one for each antecedent, each component originating at a node-language the antecedent could match, and leading to the node-language of the consequent. The components of an n-arc share a label unique to the arc. The general interpretation of an arc in the finite process structure of an SR is that, if a σ contains a process state in each of the arc's origin nodes, the succeeding σ can have process states contained in its destination node.

If a finite process structure was derived using a particular regular language as the initial state of the SR, then it is initial-state-dependent; an initial-state-independent process structure would be the result of considering all strings to be possible initial process states. Most process structure graphs will be sequence-dependent, meaning that information about sequencing of process states is present, but a graph with only one node can be called sequence-independent. It is worth looking at because the language of the single node is a state language of the system.

With a little attention to detail, it is possible to fuse process structures of communicating SR's into one finite process structure, so that the processes which proceed across SR boundaries are represented in a uniform manner with those that do not. The details of the fusion procedure determine whether the resulting graph is rate-dependent or independent, i.e., contains or does not contain information about the relative rates of the systems.

A system state graph can represent exactly the same information as a finite process structure, but the information is organized by a scheme orthogonal to that of the process structure. Each node of the system state graph for an isolated SR is a language containing all of σ at some possible point in the life of the system. Thus the system state graph is a finite automaton representing the succession of states that the SR can pass through; it reduces the SR to a finite automaton by grouping a possibly infinite number of real states into a finite number of regular languages. To find a language containing all of some possible σ in a finite process structure, we would have had to take the union of the languages of a scattered collection of nodes.

System state graphs fall into categories much as finite process structures do. The graphs of individual communicating SR's can also be fused into a single state graph, but in this case it seems more useful to make the nodes represent vectors of state languages, one for each SR in the complex. The rate-dependent form of these graphs has occupied our attention up to now: the current version is the state graph described in IV.B.3.

2. The Concept of Resolution

The following discussion applies to any particular kind of finite process structure or any particular kind of system state graph; for convenience we will use the general term "process structure."

The first thing to notice about process structures is that any non-trivial system can be described by an infinite number of different ones. When any specific property is being studied, however, only a finite subcollection will be of interest--those outside the subcollection will differ from those in it only in ways that do not provide any information about the property (or in ways that provide additional information by singling out, one by one, an infinite sequence of special cases).

Within the finite subcollection, process structures vary in their degree of resolution; they range from the graph of least resolution, which has only one node (representing a system state language), to (possibly) a canonical maximum-resolution process structure, which has enough nodes and enough detail to convey all the information about this property in the system which is capable of finite characterization. To explore the variety of process structures, we define resolution operators that transform process structures themselves. An operator for raising the resolution of a finite process structure defines a procedure for "splitting" some nodes of the process structure, i.e., making new nodes by partitioning the language of the old node, and re-computing the arcs to make a new process structure. This is done in such a way that the new process structure offers some more detailed information about the property under study than the old one did. An operator to lower resolution does just the opposite: node-languages are fused to form a simpler, less detailed, finite process structure.

If we group these operators into pairs (one lowering and one raising in each pair, each the inverse of the other) we will find that some pairs can be used to define a partial order on the set of process structures describing an SR or SR complex, and at least one pair that is known can be used to define a lattice. Thus we have a scheme for organizing all finite characterizations of the system which give any information about the property we wish to study.

It is easy to see how useful this could be. Canonical process structures (minimum-resolution, maximum-resolution, and perhaps others, for example, one in which all parts of the graph are at a uniform given level of resolution) could be generated simply. We know that there is a graph which describes any part or parts of the system in any level of detail; the partial or lattice ordering should make it possible to arrive at any desired one algorithmically.

3. Initial Investigations

Finite process structures can have some properties which would be extremely valuable in verifying characteristics, detecting flaws, and evaluating the designs of the systems they model. We want to study these properties so that we can find the process structures which have them: these will undoubtedly become the most useful system characterizations.

The most obvious one is determinism, i.e., whether an arc in the graph will be followed whenever there is a process state in each of its origin nodes. Determinism is fundamental to a number of inquiries, and it may also lead into the area of parallelism: we would like to feel free to re-arrange the implementations of finite process structures (the exact assignment of processes to RPR's and SR's) so as to maximize some notion of least time constraints on a physical implementation, but not all such re-arrangements preserve the computation that the process structure was extracted from. A knowledge of deterministic paths through the graph may illuminate the interactions that permit some alterations and forbid others.

Time bounds on processes are absolutely necessary for any verification of real-time properties. We can identify (loosely) a process with a path through the graph. Suppose we could prove--based on the essential simplicity of the matching operation--that any step (a single arc) can be completed in a bounded time in any reasonable implementation, as long as the production causing it contains no RPR's that do not halt. Any path will then take a bounded amount of time as long as all its steps are bounded, and the length of the path itself is bounded. A proof that a path is time-bounded is a strong statement about the real-time capabilities of the system!

Space being the other fundamentally bounded resource in computer systems, we might also wish to search out steps and paths that are space-bounded. A space-bounded path is one with only space-bounded steps (it must also be finite, if we are assuming that the history of the process is saved). A space-bounded step is one which requires a provably finite amount of space to complete, including scratch storage and space for the newly-generated σ-set.

At the date of this writing these ideas have not been studied in detail, but they indicate that the finite process structure is a very promising research tool. We hope to compute process structures in the study of sizeable system complexes, which calls for a remark on the problem of combinatorics--and significant demands for computer time--we will face. It seems that the combinatoric problem could be solved if the process structure analysis could be factored, perhaps by using as system state languages subsets of a true system state language which leave large parts of the graph unvisited. A number of such partial analyses, done serially, could be put together to make the whole process structure, as long as it could be proved that such a procedure was valid. Another possible approach would be to factor the analysis by hiding computations in RPR's which the R-model can deal with separately. Eventually we may discover design principles using which the designer can avoid such problems entirely--surely the most effective solution.

D. Total System Design

The ultimate goal of this research effort is to make the formal definition universe into a total design tool, one with facilities to aid or automate every aspect of system development. We are beginning to see how this might be done.

The designer will need to make many F-model simulation runs, for testing various stages of the design, and many R-model runs, for computing process structures. This should provide feedback about what design traits facilitate efficient simulation, easy computation of process structures, and maximum observation of provable system characteristics. Considering how valuable all these things are likely to be to the designer, he would do well to follow the design constraints that develop from the feedback. We can also look forward to having sophisticated pre-simulation processors; these could be used to provide macro facilities or personalized syntax to the user, to do static program checking and enforce design constraints, and to analyze and re-arrange SR's for the optimization of the simulation.

We offer to the designer many ways of representing his design: process structures, system state graphs, SR complexes completely or incompletely specified, etc. (the final design step, the implementation of an SR complex in hardware and software, is beyond the scope of this research because it is machine-dependent). We must also offer him the means to pass from any one to any other, because a design cycle may begin with any representation. Design is an iterative process, which must proceed at many levels simultaneously; our variety of representations, offering a complete range of resolution and information content, can perhaps provide him with useful tools at each stage of his work.

Finally, we hope to arrive at some solutions to the problem of system equivalence. Recognizing equivalence classes of SR complexes and process structures is extremely important in itself, as a means of system verification, but it is also a step toward design automation. If we knew, for instance, that certain transformations of process structures yielded equivalent process structures (where equivalence must be defined, perhaps as "performing the same computations"), then we could describe an equivalence class, closed under those transformations. If we further understood how such general criteria as parallelism and redundancy operate to make one process structure in an equivalence class better than another, we would be able to take any process structure a designer gave us, and return to him the equivalent process structure best suited to his needs! Finding the best complex of SR's to implement a process structure appears to be a similar problem. Here all the members of an equivalence class will be implementations of the same process structure, and the best will be chosen according to some different criterion, perhaps even efficiency on the F-model interpreter--if we can establish our claim that this is related to the efficiency achievable in a physical implementation.

REFERENCES

[FH] Fitzwater, D. R., and Hintz, C. A. "A System for the Formal Definition of Digital Systems." Comp. Sci. Tech. Report 141, University of Wisconsin, Madison, Wisconsin, 1971.

[GC] Gilbert, Philip, and Chandler, W. J. "Interference Between Communicating Parallel Processes." Comm. ACM 15, 6 (June, 1972), 427-437.

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

[MI] Minsky, M. Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, N.J., 1967.

[SI] Simon, Herbert A. The Sciences of the Artificial. The M.I.T. Press, Cambridge, Massachusetts, 1969.

BIBLIOGRAPHIC DATA SHEET	1. Report No. WIS-CS-184-73	2.	3. Recipient's Accession No.
4. Title and Subtitle A FORMAL DEFINITION UNIVERSE FOR COMPLEXES OF INTERACTING DIGITAL SYSTEMS			5. Report Date June 1973
			6.
7. Author(s) D. R. Fitzwater and Pamela Z. Smith			8. Performing Organization Rept. No.
9. Performing Organization Name and Address The University of Wisconsin Computer Sciences Department 1210 West Dayton Street Madison, Wisconsin 53704			10. Project/Task/Work Unit No.
			11. Contract/Grant No.
12. Sponsoring Organization Name and Address			13. Type of Report & Period Covered
			14.

15. Supplementary Notes

16. Abstracts

A language is presented for describing any asynchronous complex of interacting digital systems at any level of abstraction. The resulting representation is a definition of the complex of systems which is both formal, and effective, meaning that it is interpretable by a deterministic automaton. Thus it can be used to study system structures and develop practical design tools.

After the explanation of the formal definition universe, annotated examples of system representations are given. Their usefulness for proving assertions about systems is established in an outline of present and future research, with the ultimate goal of providing assistance for the total design process.

17. Key Words and Document Analysis. 17a. Descriptors

digital systems
formal systems
language semantics
asynchronous communication
design automation
process structuring
extensions to Post systems
debugging aids

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement Available to public	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 49
	20. Security Class (This Page UNCLASSIFIED	22. Price