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

A SYSTEM FOR THE FORMAL DEFINITION OF DIGITAL SYSTEMS

by

D. R. Fitzwater
C. A. Hintz

Computer Sciences Technical Report #141

November 1971

A SYSTEM FOR THE FORMAL DEFINITION OF DIGITAL SYSTEMS

by

Donald R. Fitzwater and Cynthia A. Hintz
University of Wisconsin*
Madison, Wisconsin

ABSTRACT. The structure and meaning of each operating system, and the programming languages it interprets are normally defined only in the context of a specific computer. This makes the recognition of computer independent structures more difficult, and inhibits the development of both general design principles and computer independent language and operating system structures. A formal system for the definition of computer independent structures is required.

A formal system for representing general digital systems, based on extensions to Post string manipulation systems, is developed and defined. Very general forms of digital systems, interacting in real time can be represented. The formal system allows definition of equivalent systems at many levels of abstraction without biasing design decisions by a particular way of doing things.

Examples of the use of this formal system for defining and comparing circuit elements, automata, and programming language semantics are presented.

---

*Computer Sciences Department. This research was supported in part by the Wisconsin Alumni Research Foundation.

KEY WORDS AND PHRASES: Digital systems, formal systems, language semantics, abstract real time systems.

CR CATEGORIES: 4.0, 4.1, 4.2, 5.2, 6.0, 6.2

1. INTRODUCTION

At the present time nearly all digital systems work is carred out in the almost arbitrary environment of a particular machine. The meaning of the vocabulary used to define the system is explained only in terms of the particular structures found in that machine. We have not developed a well-defined vocabulary, a general way of defining or comparing systems, or a way of predicting the intrinsic properties of a system from its definition. Although inter-system compatability is a desirable goal, we have no way to define it or to achieve it. Lacking these capabilities, system design work is largely ad hoc with continual re-invention of concepts and with the more general concepts which should have guided all such designs being substantially obscured. Perhaps the most significant obstacle to achieving these capabilities is the lack of a practical way to define all such systems in a common, non-biasing way. We need to define a consistently described formal universe of systems, large enough to contain all digital systems of interest, within which we may study invariancies and equivalents, and may develop design tools. In effect we must create a computer science. An introduction to some of the concepts involved in such a science can be found in a fascinating book by H. A. Simon [1] on The Sciences of the Artificial.

1.1 Formal Universe of Systems

For us to have a computer science we must have some formal universe with intrinsic properties which we may describe, modify, and predict. Our science currently has as many universes as there have been computers built or dreamed of. Some portions of our science, those having to do with computability and decidability, have been extensively and rather successfully studied using Turing machines, Post systems, automata theory and recursive function theory. We are concerned here with other questions which may not be answered with the same independence of any physical discovery. We are interested in those aspects which may be described as a physical science.

The physical sciences have a common but incompletely described universe called our physical universe. Each such science, however, selectively abstracts certain properties of the common physical universe and tries to describe their nature and relationships. Each such science, in effect, hypothesizes a different formal abstraction of the real physical universe but validates its conclusions by real measurements on the physical universe. This abstraction then becomes the formal hypothetical or theoretical universe of the science. Progress in science consists of validating and extending this formal hypothetical universe.

Progress in computer science has been slow because no such hypothetical, machine independent formal universe of computer systems has been developed or commonly agreed upon. Each computer system has many intrinsic properties, such as color or weight, which are irrelevant when we want to consider what the system is doing or what it represents. Our description of the system must be made at an abstract level which leaves out details which are not essential. The main difficulty lies in deciding which details are essential.

If one is interested in comparing systems with respect to computability, the details irrelevant to Turing machines may be left out of our universe. Most of the formal work on computing systems is so abstract as to leave computability as almost the only intrinsic property of the system. Most of the systems of interest to us can simulate, until some bound is reached, a Turing system and thus are all equivalent (within their bounds) with respect to computability. This level of abstraction has left out nearly everything of interest in computer systems. We must start with a system universe as abstract as possible so as to afford a hope of gaining sufficient insight to reach a more detailed level, but not so abstract as to eliminate time and space. Our systems must at least have a size and operate in time.

Since we can not expect to create immediately an appropriate hypothetical universe within which all interesting questions may be studied, we must start by specifying a hypothetical formal universe which, at least, is general enough to contain some interesting abstraction for all existing computers. An "interesting abstraction" is one that gives further insight into possible systems and which makes possible some useful predictions and manipulations of the intrinsic properties of the system.

We will restrict our attention, at present, to "digital systems" which are characterized as being in a well-defined state or in a transition requiring at least a finite time between such states. We will consider interacting complexes of such digital systems in a very general form.

Such a formal systems universe becomes a hypothesis of our science. If a better way of structuring a system is found and it is not in our formal universe, we should modify our hypothesis to contain it. With a given hypothesis, we should be able to establish useful laws governing relationships of system properties. We need not claim a unique or absolutely proven hypothesis; we need only claim our current hypothesis as the best we have yet. Thus we may only speak of "scientific truth" which is dependent on a changing technology rather than "mathematical truth" which is invariant to any possible discoveries.

1.2 Representation of Systems

Much effort has gone into formal specification of abstract procedures in a representation independent fashion, in order to ensure exact implementability on different systems. However, the advantages of such abstractions are offset both by the difficulty of such specifications and by our current inability to design systems which can interpret such specifications in general. Such specifications, being representation independent, are too abstract to be directly useful to the implementers of the systems, since real system processors, which manipulate the representation itself, are always representation dependent. The systems designer is interested in these manipulations, not the abstract notions they might represent.

For example, a mathematician may define an abstract number system and operations on it in a formal, representation independent fashion. To implement such a system, the designer must choose a suitable representation which allows the appropriate transformations to be carried out in an efficient manner. This indeed is a major problem for the systems designer and not one that we are close to delegating to a computer to solve. In other words, a representation independent definition of a system may be useful for proving theorems about the system but it is not suitable as an operational definition of the system. An operational definition is one

for which we can construct a universal system which will interpret (i.e. run) any system so defined.

The usual representation of an operational definition of a computer system has been an implementation of the system itself. For example, if we strictly define the meaning of Fortran to be the execution of the translator output in machine language on the IBM 704, another system must simulate an IBM 704 executing its machine language in order to interpret Fortran. Strictly speaking, Fortran was defined in this fashion. Consequently other system designers had to reject this definition and thus Fortran programs, in general, will not do the "same thing" on different machines.

An operational definition should be an efficient representation of the system structure and should also be abstract enough to eliminate the need to include extraneous structure, or unintended details of transformations. We can, therefore, profitably use an abstract representation medium whose objects have only the properties which are explicitly defined. This is such a common and necessary thing to do that it is almost unnoticed that the actual physical representation of a computer is rarely described in a programming manual or even in a system definition for the engineers who service it.

In order to define our formal systems universe and to secure operational definitions of systems in our universe, we must choose a suitable representation

medium and describe a primitive automaton which can operate on objects in that medium. Note that, although we constrain ourselves to a single abstract representation medium, a given system may have, in our formal systems universe, many equivalent representations at various levels of the abstraction. Indeed the translation from one representation to another, often into different representation media, is an important part of the work normally done by a system or a systems designer. We would like to be able to study such transformations in our systems universe. Possible abstract representation media include integers (Gödel numbers*), strings, and directed graphs. Each such medium could be used to construct a formal systems universe.

We can consider integers as a point medium, strings as a line medium, and graphs as a plane or three-dimensional medium. Trees are a restricted form of directed graph that may be represented linearly in a parenthesized notation. Three-dimensional systems are difficult to document and immensely complex, far too complex to start with. They do exist; a beautiful example is the universe of living organisms with operational definitions in genetic material and a primitive system for their interpretations based on chemical transformations. The similarities with the simpler linear universe we will define are remarkable.

Gödel numbers would make a poor representation medium for many reasons. Their simple transformations do not correspond to very useful ones

---

*See Minsky [2] Section 14.3 for an explanation of Gödel numbers.

and one must do elaborate transformations to form expressions which must then be elaborately decoded. The decoding efficiency is very poor and structural insights are not easily obtained.

A bit string, a reasonably conventional medium, has many advantages but two very serious disadvantages. A bit has such a small number of attributes and values (i.e. one attribute with two values), that an object in an abstract process will generally require a bit pattern rather than a bit for its representation. This means that object transformations may only be expressed in terms of very primitive operators on bits. This forces us to express higher level object manipulations in unintended detail which any other system for doing the "same thing" must copy exactly. The other serious disadvantage of a bit string is that when all object representations and transformations have been laboriously forced into an undesirably detailed sequence to define the system, the system designers no longer have as much choice or freedom to adapt to technological or social changes without discarding previous designs. This is what has been happening with our system "generation gaps". System languages have been such low level languages that applications written in those languages must be stated in unintended detail which must be simulated by the next generation unless the application is to be rewritten.

A directed graph-based representation medium clearly allows more general objects and transformations. However, most current computer memory structures are basically linear rather than the more complex forms made possibly by a graph structure. Simple operations such as copying an object become very complex when it is a graph-structured object. It may be that a full graph structure would be nearly as unbalanced a choice as Gödel numbers.

The fact that current systems surely are technologically reasonable representations of themselves, in addition to current work on languages as strings, suggests that our starting point should be a linear representation of trees of character string values. If we then define an object as a character string, and a structured object as an object containing substrings, we have a system which is represented by trees. More generality could only be obtained by allowing a full graph structure. While we are still trying to sort out our ideas and ways of discussing them, an elaboration to full graphs would be an encumbrance. There is also less hope for immediately useful designs because their implementation on basically linear machines would be complex and inefficient.

Our formal universe must be defined as an abstract formal system. It must be abstract so that we can know just what properties are being transformed and it must be formal so that we know that no other properties are relevant. An informal system would require consideration of other properties and

relationships not explicitly defined in the system in order to define different but equivalent systems.

When we look for abstract formal systems for string manipulation to use as a starting point for our formal systems universe, we find one outstanding formal system, that defining the Post production systems. As we will see later, the Post systems are so beautifully suited to our purpose that scarcely any other candidate need be considered. If we can move freely in concept between our systems and Post systems, we will establish a bridge between our "real" systems and the formal work that has already been done.

The Post system formalism is, however, not sufficient for our purposes for three primary reasons: there are no concepts of time, of space, or of inter-system communication left in the abstract system. We must extend the Post system to include these concepts in such a manner that we preserve Post system properties while enabling us to define and discuss our "real" systems.

1.3 Desirable Properties

Since we would like to use our hypothetical systems universe to study possible system structures, it is essential that our universe not already contain implicitly a particular form of the structures we wish to study. It would otherwise be difficult to avoid biasing the study in favor of the built-in

very sophisticated notion of substring matching and manipulation, and their unrestricted order of production application, are at a sufficiently high level that in explication in Post systems would serve to illuminate rather than obscure real language concepts.

Since one of the purposes of creating our formal systems universe is to study ways to design and implement efficient systems, there must be some useful relationships between our abstract systems and the efficiency and cost of implementation of systems so structured. We should at least be able to compute some bounds valid for such implementations. We can thus study various logical design principles and structures in terms of their effect on such bounds, e.g. we can compare relative system efficiencies. To do this we must introduce the concepts of some hardware technology which provides efficiency constraints for the design of our abstract systems. We will not pursue this further here, but will only comment that the formal systems universe defined here seems well suited to this purpose.

The most desirable property of our formal universe would be something like: "the simplest and most elegant abstract systems in our formal universe have the most effective and efficient implementations". It is of course impossible to show that our formal systems universe has this property; a more precise formulation of it must await the insights gained from a study similar to that

structures. For example, because we are interested in control structures, our formal systems universe should allow all forms of control without having them restricted in their relationships due to inherent formal universe properties. Without design constraints all possible transformations must occur simultaneously, so that less parallel control concepts must be explicitly defined as restrictions in a systems design. For example, the order of production application imposed by Markov algorithms [3] creates implicit control in a way that biases the study of control, while the (effectively) simultaneous application of all possible productions in Post systems requires the system designer to explicitly build in any control he desires. Similarly, because we are interested in relationships between programming languages and their semantics, our formal universe should allow the interpretation of all forms of programming languages. Restricting a system to interpret a particular programming language should require explicitly imposing the appropriate design constraints.

Although the primitive concepts in our formal universe must not presume to contain the concepts we wish to discuss and develop, they must be relatively "high level" concepts with substantial complexity so that explications of system structure will not be more complex and difficult to understand than are the structures to be explicated. A Turing machine which interprets Algol is not a good place to study Algol concepts. However, Post systems, with their

mentioned just above. It is clear that many abstract simplifications would have their counterparts in simplifications of any implementation.

2. A SYSTEMS UNIVERSE AND DESCRIPTION LANGUAGE

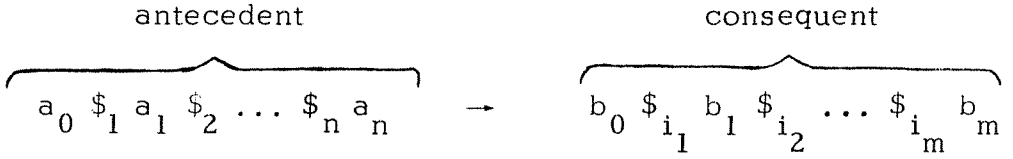
In order to define our formal systems universe, we will describe a "primitive automaton" which can be used for the formal definition of any system in our formal universe. These system definitions will be written in a "system definition language" which our primitive automaton can interpret, i.e. transform into a sequence of definitions of the successive states of the given system. We will develop the rules that our primitive automaton must follow as a sequence of modifications to the rules for interpreting Post systems.

2.1 Post Systems

In 1942 Emil Post introduced a formal system for manipulating strings of symbols. We will briefly discuss his system here; Minsky [2] gives a more detailed explanation.

Post's formal system has three parts: the alphabet, the axioms, and the productions. The alphabet is a finite set of symbols specifying the character set. The axioms are a finite set of finite strings of symbols from the alphabet. The productions are a finite set of rules of inference by which strings may be transformed.

A simple production is of the form:



where $a_0, a_1, \ldots a_n$ and $b_0, b_1, \ldots b_m$ are literal strings of symbols, and $1 \leq i_j \leq n, \forall_j$. These strings and all of the axioms contain only symbols from the alphabet. Any of these literal strings can be null.

A production implies that we may take something of the form of the antecedent and produce a result patterned after the consequent. A literal string in the antecedent, of course, can only be matched by an identical literal string in the axiom. The $'s in the antecedent can match any substring of an axiom (including the null string). The $'s in the consequent refer to the value that was matched by one of the $'s in the antecedent, which of the $'s of the antecedent being indicated by the index following the $ in the consequent. (The $'s in the consequent need not appear in the same order in which they appeared in the antecedent, and $'s in the antecedent need not appear in the consequent at all.) We can also have multiple-antecedent productions, where the single antecedent is replaced by two or more antecedents separated by a given delimiter.

For a production to be applicable, each antecedent must match some entire axiom, and all antecedents of a multiple-antecedent production must be matched*. Each different way the production could match the

---

*Minsky's proof that multiple-antecedent productions can be reduced to single-antecedent systems assumes that no two antecedents in a multiple-antecedent production may match the same axiom. We will remove that restriction.

axioms is counted as a separate possible application. The result of one application, based on the pattern of the consequent and using the value of the $'s from the match of the antecedent, would then be considered derivable in the system containing the production.

For example, suppose we have the following Post system:

Alphabet: The single symbol 1

Axiom: The string 1

Production: $\$_1 \rightarrow \$_1 1$

In this system, the antecedent $\$_1$ can match the axiom "1". The result would then be the "1" matched by the $\$_1$ concatenated with a literal "1", so that the string "11" is derivable in this system. The antecedent $\$_1$ can also match "11"; the string "111" is thus also derivable. Other possible derivations are "1111", "11111", etc. If we view these strings as representing unary numbers, then all positive integers are derivable in this system, and we could show a possible derivation sequence for any given positive integer.

Formally, Post canonical systems only specify a set of strings by recursively specifying how to find things in that set; they do not define a process. One can show that a certain string is derivable in a system by showing a (possible) sequence of steps in which the given string is the last string derivable, and all previous strings are derivable from the initial axioms

or from previously derivable strings via a sequence of applications of the productions. In Post systems the emphasis is on derivability, not on the actual derivation.

At this point we will make an extension to the Post systems. Since the recognition of a character match implies the ability to recognize a non-match, we will permit the use of $\bar{x}$ (read "not x") in productions, where x is any character in the alphabet. $\bar{x}$ will match any single character in the alphabet other than x. In effect, $\bar{x}$ becomes a string variable of length one (i.e. it cannot match the null string). Thus, for Post systems, a production containing a $\bar{x}$ is equivalent to a shorthand convention for a set of productions, each of which contains a different, non-x character from the character set in the position of the shorthand $\bar{x}$. For the sake of efficiency, however, we want this ability to recognize a non-match to be built into the primitive automaton we are developing; thus it will become an extension rather than a shorthand notation.

2.1.1 Post System Language

We want to implement our Post systems so that we may see the actual derivations which take place. In order to do this, we need a system representation and an appropriate primitive automaton which interprets that representation. Since the actual representation is quite arbitrary and a

string of characters is a natural representation medium, we will define a Post System Language (PSL) of which each sentence is a Post System Representation (PSR). We may define a PSL by specifying a vocabulary of terminal characters $V_T = \{\underline{i},\ i = 1,2,\ldots n\}$ and the grammar given below.

<PSR> ::= $\underline{1}$ <axiom part> <processor part> $\underline{2}$

<axiom part> ::= $\underline{3}$ | $\underline{3}$ <axiom string>

<axiom string> ::= <axiom> | <axiom string> $\underline{7}$ <axiom>

<axiom> ::= <$\underline{j}$> | <axiom> <$\underline{j}$>

<processor part> ::= <char part> <production part>

<char part> ::= $\underline{4}$ | $\underline{4}$ <concatenation of a subset of $\{\underline{i} \mid 20 \leq i \leq n\}$>

<production part> ::= $\underline{5}$ | $\underline{5}$ <production string>

<production string> ::= <production> | <production string> $\underline{6}$ <production>

<production> ::= <mult. antecedent> $\underline{9}$ <consequent> | <mult. antecedent> $\underline{9}$

<mult. antecedent> ::= <antecedent> | <mult. antecedent> $\underline{7}$ <antecedent>

<antecedent> ::= <ante char> | <antecedent> <ante char>

<ante char> ::= $\underline{8}$ | <$\underline{j}$> | <$\bar{\underline{j}}$>

<consequent> ::= <cons char> | <consequent> <cons char>

<cons char> ::= <$\underline{j}$> | <$\bar{\underline{j}}$> <index> | $\underline{8}$ <index>

<index> ::= <nz subscript digit> | <index> <subscript digit>

<nz subscript digit> ::= $\underline{11}$ | $\underline{12}$ | $\underline{13}$ | $\underline{14}$ | $\underline{15}$ | $\underline{16}$ | $\underline{17}$ | $\underline{18}$ | $\underline{19}$

<subscript digit> ::= $\underline{10}$ | <nz subscript digit>

$\langle \underline{j} \rangle ::= \underline{20} | \underline{21} | \underline{22} | \ldots | \underline{n}$

$\langle \bar{\underline{j}} \rangle ::= \bar{\underline{20}} | \bar{\underline{21}} | \bar{\underline{22}} | \ldots | \bar{\underline{n}}$

Note that we can define many different PSLs simply by changing the set V_T. The terminal characters $\underline{i}$ may be given any desired unique character representations. However, because of the tedium of the $\underline{i}$ notation (evident in the grammar above), we shall henceforth use the print equivalents given in table 1. If we define $\langle \underline{20} \rangle$, $\langle \underline{21} \rangle$, $\langle \underline{22} \rangle$, etc.

Table 1: Print Equivalents of $\underline{i}$ in PSR

$\underline{i}$ $1 \leq i \leq 9$ are meta characters used in defining a system.

$\underline{i}$ $10 \leq i \leq 19$ are special subscript characters used as meta "integers" in defining a system.

$\underline{i}$ $20 \leq 1 \leq n$ are assigned as desired for a particular system.

$\underline{i}$	print equivalent	$\underline{i}$	print equivalent	$\underline{i}$	print equivalent
$\underline{1}$	{	$\underline{10}$	0	$\underline{20}$	
$\underline{2}$	}	$\underline{11}$	1		
$\underline{3}$	$\underline{6:}$	$\underline{12}$	2		
$\underline{4}$	$\underline{\chi:}$	$\underline{13}$	3	.	
$\underline{5}$	$\underline{\Pi:}$	$\underline{14}$	4	.	
$\underline{6}$	$\underline{\text{or}}$	$\underline{15}$	5	.	
$\underline{7}$	$\underline{\text{and}}$	$\underline{16}$	6		
$\underline{8}$	$	$\underline{17}$	7		
$\underline{9}$	→	$\underline{18}$	8		
		$\underline{19}$	9	$\underline{n}$	

as print equivalents of the corresponding members of V_T we may rewrite the <PSR> grammar in a more readable form using print equivalents where known.

<PSR> ::= {<axiom part><processor part>}

<axiom part> ::= $\underline{\sigma:}$ | $\underline{\sigma:}$ <axiom string>

<axiom string> ::= <axiom> | <axiom string> and <axiom>

<axiom> ::= <$\underline{j}$> | <axiom><$\underline{j}$>

<processor part> ::= <char part><production part>

<char part> ::= $\underline{\chi:}$ | $\underline{\chi:}$ <concatenation of a subset of $\{\underline{i} \mid 20 \leq i \leq n\}$>

<production part> ::= $\underline{\pi:}$ | $\underline{\pi:}$ <production string>

<production string> ::= <production> | <production string> or <production>

<production> ::= <mult. antecedent> → <consequent> | <mult. antecedent> →

<mult. antecedent> ::= <antecedent> | <mult. antecedent> and <antecedent>

<antecedent> ::= <ante char> | <antecedent><ante char>

<ante char> ::= $ | <$\underline{j}$> | <$\bar{j}$>

<consequent> ::= <cons char> | <consequent><cons cher>

<cons char> ::= <$\underline{j}$> | <$\bar{j}$> <index> | $<index>

<index> ::= <nz subscript digit> | <index><subscript digit>

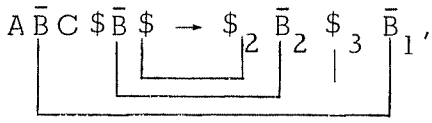
<nz subscript digit> ::= $_1 | _2 | _3 | _4 | _5 | _6 | _7 | _8 | _9$

<subscript digit> ::= $_0$ | <nz subscript digit>

<$\underline{j}$> ::= <$\underline{20}$> | <$\underline{21}$> | <$\underline{22}$> | ... | <$\underline{n}$>

<$\bar{j}$> ::= <$\overline{20}$> | <$\overline{21}$> | <$\overline{22}$> | ... | <$\bar{n}$>

The semantics associated with the PSR are obvious with the possible exception of <cons index>. If we consider each occurrence of $\bar{i}$ and $ to be indexed from left to right (starting at 1) in the antecedent(s), then the <cons index> in a <cons char> references the indexed occurrence of the particular $\bar{i}$ or $. If the integer value of the <cons index> is greater than the number of $\bar{i}$'s or $'s in the antecedent(s), the value associated with the <cons char> is null. An example of this correspondence is:



, where the lines connect corresponding variables.

2.1.2 Post Primitive System

Without changing the properties of Post systems, we are free to decide how our primitive automaton will operate to produce the derivation sequences, and to relax certain restrictions.

a) In Post's systems, axioms and productions could contain only characters found in the alphabet. Here the purpose of the alphabet has been changed to that of designating the possible symbols which can be matched by a $ or a $\bar{i}$.

b) Our primitive automaton will ONLY accept a string which it recognizes as a PSR, as defined above.

c) Two or more antecedents in a multiple-antecedent production may match the same axiom.

d) Finally, and most important, a method will be given for actually generating consequents if the antecedent(s) of a production can be applied. Thus any string which is derivable in a system will actually be derived. Moreover, our primitive automaton must not progress on a particular derivation sequence without progress on all possible derivation sequences. First all strings which can possible be derived from the axiom(s) will be derived; then all strings which can be derived from the axioms and the previously derived strings which can be derived from the axioms and the previously derived strings will be derived, etc. This is done by keeping newly derived strings in a separate set until all the currently derivable strings are formed, after which these derived strings are added to the axiom set.

These modifications do not alter the set of strings derivable in a Post system or their derivation sequences. They do, however, lay a foundation for some desirable extensions to Post systems.

We must now take a look behind the scenes in the Post systems, to see how our primitive automaton will transform any system definition of the proper form. Consider what would have to be done to interpret any arbitrary system of this form. Clearly one would go through the productions

one at a time and see if the antecedent(s) match any (non-null) axioms. If so, the results must be saved. After all productions have been tested, we will make our results into axioms, and repeat the process.

Figure 1 gives an informal flow chart for a primitive automaton to interpret our PSR. This description is not unique but is so arranged as to display a maximum of continuity with the primitive automaton we will later develop.

The "valid PSR" test checks to see if the Post System Representation is in the proper form.

σ is the set of system axioms. σ' is a set whose contents vary with time, used to collect the results of applying the productions to the axioms in σ. At the beginning of every process step σ' is set to null (i.e. the empty set).

π is the set of system productions. π' is a consumable set of productions whose contents will change with time. At the beginning of each process step, copies of all of the productions in π are put into π'.

"Null (π')" checks to see if all productions have been removed from π'. If they have, we know that all productions have been applied and that we have generated all possible consequents.

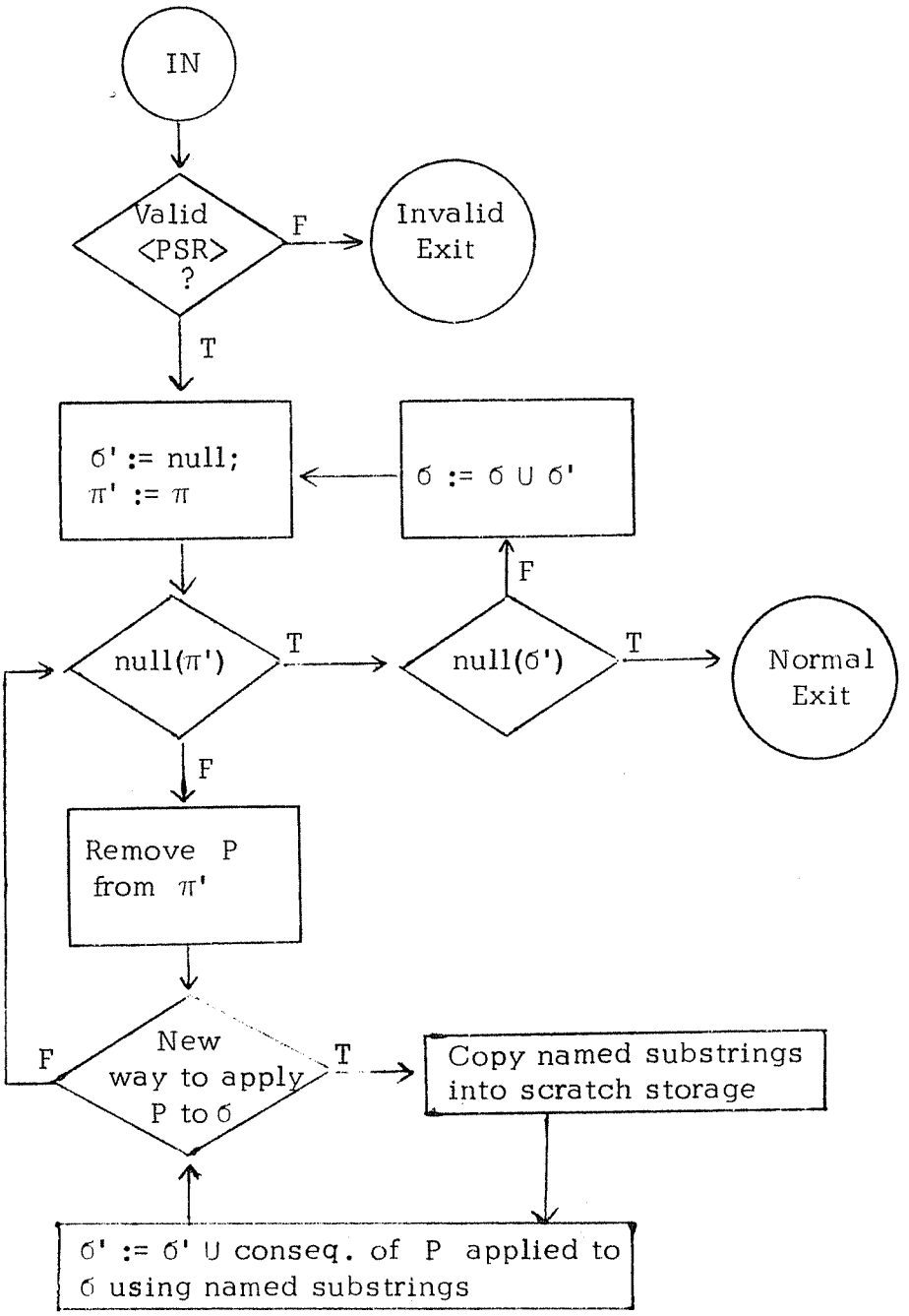


Figure 1: Primitive system for executing <PSR>.

π = set of system productions
σ = set of system axioms
π' = consumable copy of π
σ' = set to collect consequents during system step
P = current production drawn at random from π'
Null = null string
Null (σ') = null predicate

"Remove P from π'" takes, by arbitrary choice*, some production out of π'. P will then refer to this production, and π' will then name the set of the remaining productions.

"New way to apply P to σ" tests whether the antecedent(s) in production P match any of the (non-null) axioms in σ. If not, we go back to see if all productions have been checked. If the antecedent(s) do match, we go on to calculate the result of this production. Each time we re-enter this box, we will look for a unique way of matching, not used in previous tests since P was selected. Note that P may match the axioms in σ in many ways and the purpose of this test is to find all the ways and use the True exit once for each of them. When all possible matches have been handled, we will go back to see if there are any further productions to be checked.

"Copy named substrings into scratch storage" saves all those parts of the matched axiom(s) which correspond to a $ or a $\bar{i}$ in the antecedent(s) of production P. These may be used to construct the new result patterned after the consequent of production P. For example, one possible match of the antecedent $\$\Delta\bar{\Lambda}\ \$\Delta\$$ to the axiom $00\Delta 10\Lambda 10\Lambda$ would result in $\$_1$ matching 00, $\$_2$ matching $0\Delta 10$, $\$_3$ matching the null string, and $\bar{\Lambda}_1$ matching 1 (assuming that 0, 1, and Δ are in the character set).

---

*Actually, in any given implementation of the system the choice will not be arbitrary, but the method of selection will have absolutely no effect on the results.

"σ':=σ' $\cup$ consequent of P applied to σ using named substrings" calculates the result of P using the consequent of P as a pattern with the named substrings from scratch storage defining the values of the variables ($'s and $\bar{i}$'s). This result is added to σ'. Since σ' is a set, identical results from different productions will appear only once in σ'. We then go back to check for a new way to match the antecedent(s) of P with the axioms in σ.

"Null (σ')" tests to see if any result(s) were calculated. If there were none, the system halts, since it can never produce any further results.

"σ:= $\sigma \cup \sigma'$ " adds to the axiom set the results of all the possible applications of the productions which are different from the axioms which are already there. This completes one system process step. We then go back to see if any new result(s) can be derived using the new axiom set. This system will never stop since the results derived on the first process step will always be derived again, so σ' will never be null at the end of a process step.

The primitive automaton transforms a PSR, during one system process step into a unique successor PSR in which the axiom set has been changed. During the next process step the new PSR will be transformed into its

successor PSR. However, only the <axiom set> is changed. The <char set> and <production set> are invariant to these transformations; together they can be regarded as an invariant processor which will produce different results depending on the data (i.e. the axiom set) it is applied to, but which will perform the same kinds of transformations on any type of data. Thus a system representation consists of a processor (which is invariant through time) and a particular axiom set (which undergoes transformations in time).

We now have a deterministic, discrete state, formal system in which interpretation sequences are defined even though the sequences of production applications are not defined. Since none of the sets σ, σ', π, or χ are ordered sets, PSR's which differ only in the ordering of their respective sets are considered to be equivalent. It is also not possible to associate axioms in the successor PSR with corresponding axioms in the initial PSR if the axiom could have been a transformation of more than one original axiom.

2.2 System Definition System

The primitive automaton just described will interpret systems which still have only the basic character of Post systems. We will now extend our systems universe to include systems which are more general. This will require an expanded System Representation (SR as defined in section 2.2.4)

to reflect these extensions, and modifications of the primitive automation (as shown in figure 2) so that it can interpret the extended SR's. Our final System Definition System will be formed by adding to our modified Post systems three things: the ability to discard irrelevant axioms, the ability for systems to communicate with each other, and the generalization of antecedents to permit the use of a restricted processor as an <ante element>.

2.2.1 Forgetting Irrelevant Axioms

As defined, a PSR will preserve all of its original axioms and any new axioms generated. This means that by the time several transformations have been made, the axiom set of a complex PSR will be getting unwieldy. Because of all the irrelevant information accumulated it may be very difficult to pick out what we consider to be the relevant axioms--usually the last transformations that were made.

We may also want certain productions to be applied conditionally--only if certain other conditions have been met. This can be done initially by using the conditions as multiple antecedents of the production, so the production would not be applicable until all the conditions had been satisfied. However, once these conditions have been satisfied, they will be satisfied forever after. They are like switches which can be turned on but not off.

Both of these problems have one cause: we cannot get rid of an axiom once it has been formed. At some point we want to be able to discard all irrelevant or unwanted axioms. A logical place to do this is at the end of a system process step. However, who is to determine what is irrelevant or unwanted? Certainly the primitive automaton cannot know; it must be the designer of the system who decides. We can give the system designer this powerful tool by simply "forgetting" the old contents of σ before a new set σ is formed from σ' (see figure 2). In order to simulate a Post system, i.e. retain all axioms ever produced, one could merely add the production $\$ \rightarrow \$_1$ (which would be redundant on the original primitive automaton shown in figure 1). Since the antecedent $\$$ would match all axioms*, a copy of all axioms would be put into σ' and thus be carried along to the next process step, i.e. all axioms would be remembered.

2.2.2 Inter-system Communication

Since our systems now have interpretation sequences which define system time, we may be interested in examining, as non-interacting observers, their behavior. We may also be interested in the behavior of a system complex consisting of interacting (communicating) systems. Such interaction between systems (including observers) can be accomplished only via a new concept in our primitive automaton, namely the concept of intersystem communication. This is accomplished by having a system send

*This is true only in Post systems or other systems where all axioms and productions are strings over the system alphabet.

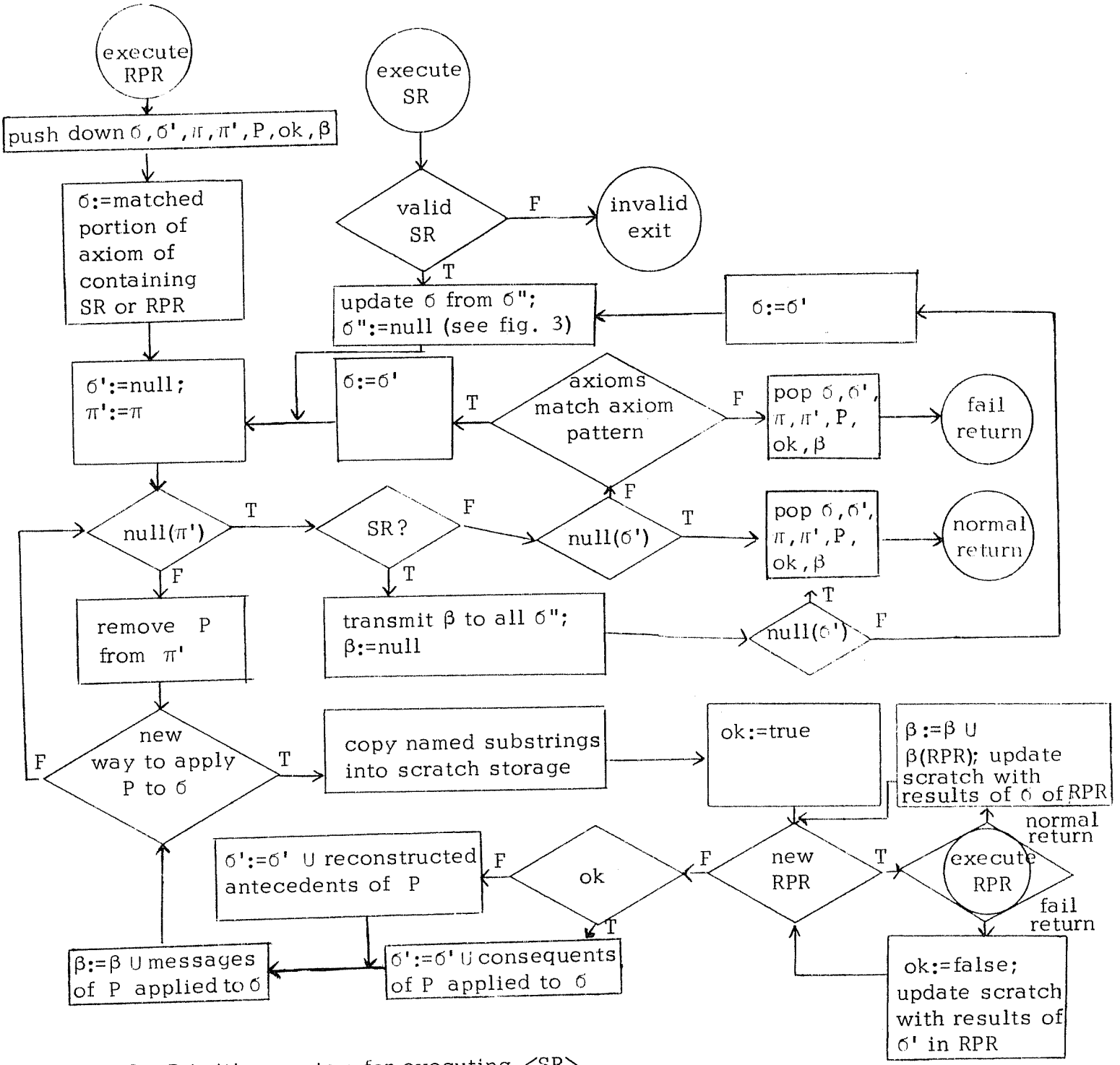


Figure 2: Primitive system for executing <SR>.

π=set of system productions
σ=set of system axioms
π'=consumable copy of π
σ'=set to collect consequents during system step
σ"=system buffer for receiving messages
P=current production drawn at random from π'
β=set to collect messages generated during system step
β(RPR)=the β of the contained RPR

"messages" (axioms) on named "channels" (destination names) to each system in the interacting complex, where they will be received in a set σ".

Sending messages: Every production has an arrow ($\rightarrow$) separating the antecedent(s) from the consequent. Another arrow following a consequent indicates that the result associated with the consequent before the arrow is a message which is to be sent (at the end of the system process step) on the "channel" named by the result associated with the consequent following the arrow. Thus a production can be of the form <mult. antecedent> $\rightarrow$ <cons. element> $\rightarrow$<cons. element> where the first <cons. element> is a pattern for the generation of the message and the second <cons. element> is a pattern for the generation of the "channel" name.

Whenever a message is constructed as above, it is added to the message buffer, β, of the source system under its designated channel name. This happens even if a given channel name already has other messages to be sent (i.e. messages produced in the same process step by other productions or other ways of applying the current production). The β message buffer will collect all messages produced in one process step of an SR. At the end of the process step, all of these messages will be sent to the σ"s of all the systems in the system complex, after which the message

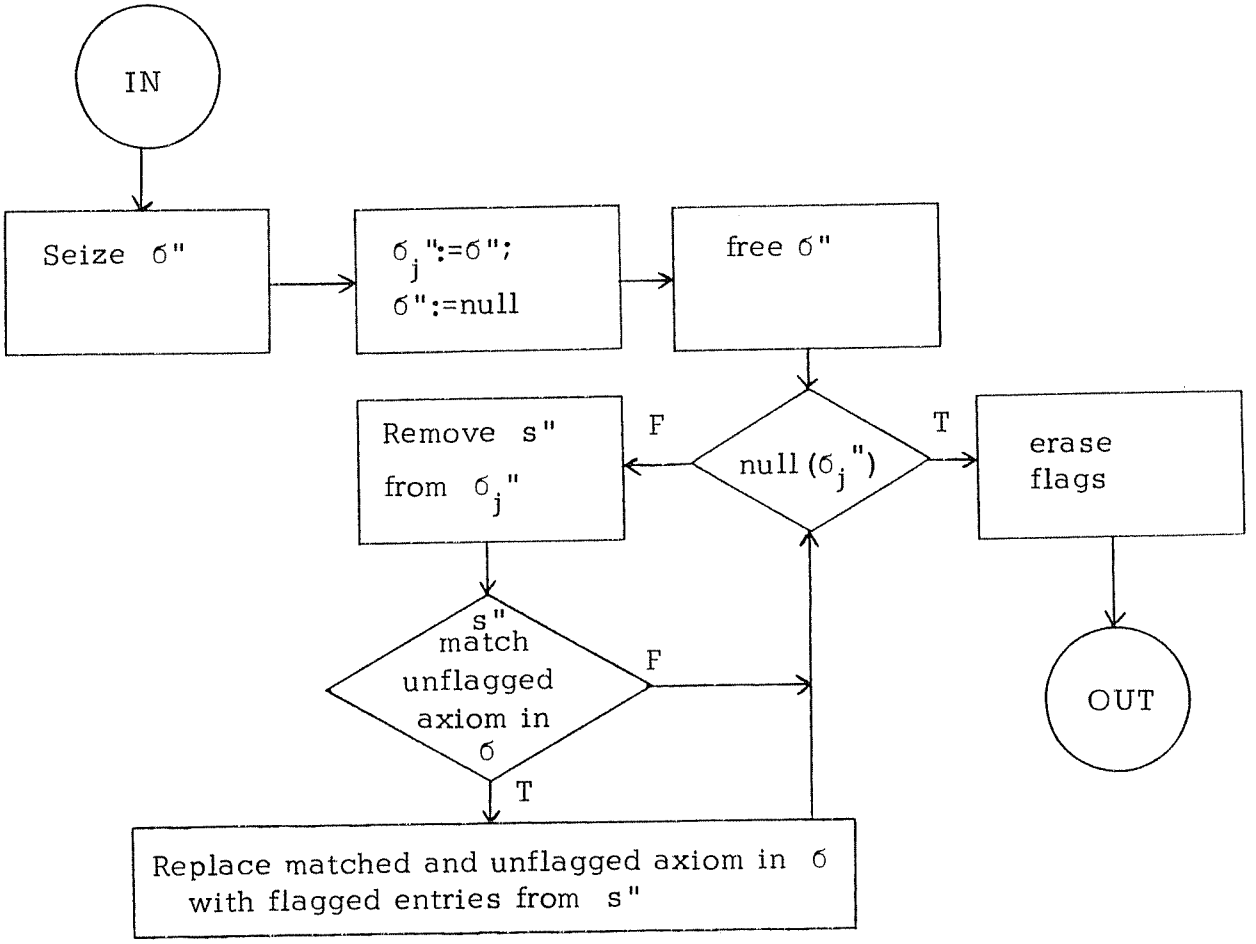


Figure 3: update σ from σ''; σ'' := null

σ_j'' = consumable copy of σ''

s" = labeled set of messages which forms an entry in σ''

buffer will be set to null. A similar buffer β(RPR) is associated with the execution of an RPR (discussed in the next section) to collect any generated messages. This buffer will be returned to the containing SR or RPR when execution of the RPR is completed. Note that RPR generated messages are transmitted by the system only upon completion of an SR process step.

Receiving messages: As a new process step for an SR is started, σ is updated from the σ'' of the system as shown in figure 3. The set σ'' is seized (locked against new entries) and a consumable copy of σ'' is made, leaving σ'' null so that it may start collecting messages from other systems again.* σ'' is then freed to accept new entries. An entry, s", (consisting of a channel name and the set of messages that it names) is removed from this copy of σ''. If the name of this entry is the same as some unflagged axiom† in σ (all axioms are initially unflagged), that axiom is replaced by the messages associated with that name which are then flagged. If the name of the entry does not match any axiom in σ, the entire entry is discarded. This is done for all the entries in the copy of σ'', and then all flags on axioms in σ are removed.

Because σ'' is set to null at update time, if an entry's name does not match any of the axioms in σ at the time of the update, these messages will be lost. They will not exist in σ'' at the time of the next update unless the sending system (or some other system) repeats them.

---

*The consumable copy is needed to prevent a possible infinite update phase where another system may be sending messages faster than the system we are examining can process them.

†If the channel name pattern is a $ which matches the null string, the messages cannot be picked up since the null name cannot match any axiom.

Collecting messages in σ": At the beginning of a process step, σ" is set to null. When another system completes its process step, it sends the messages in its message buffer under their given channel names. If the receiving σ" does not contain a particular channel name, that channel name and its associated set of messages will be placed in σ". If the receiving σ" already has a set of messages under that channel name (either sent from another system or sent by the same system on a previous process step), the earlier set of messages is discarded and replaced by the current set of messages.

For example, suppose that there is a system A whose system process steps are faster than those of a system B. Suppose that A sent a message under a given channel name at the end of A's process step. If A completes another process step and sends another message under the same channel name before B completes a process step and updates its σ from its σ", the earlier message on the channel is lost. Thus on any update, a system receives only the most recent messages sent on any given channel. This makes real time systems possible (see section 2.2.5 for an example).

2.2.3 Restricted Processors.

We would like to define process steps of arbitrary complexity and to restrict the values which a given variable may match. We can do this without destroying the desirable properties of our modified Post systems by adding a modified form of SR as a new type of <ante element>.

These modified forms of SR's willbe called RPR's (Restricted Processor Representation). They differ from SR's in three basic ways: 1.) RPR's are restricted in that they cannot receive messages (and therefore need no σ") 2.) RPR's can send messages only indirectly through the containing SR or RPR, and 3.) RPR's have an axiom pattern rather than an axiom set σ, which makes them invariant processors.

The axiom pattern and the character set χ of an RPR determine the axiom substrings of the containing SR or RPR which will initially match the RPR as an <ante element>. When determining if a production P which contains an RPR can be applied in a "new way" to the axioms in σ, all variables in the antecedents take on certain values: a $ will match zero or more characters from the set χ of the containing SR or RPR;* $\underline{\bar{i}}$ will match a single character from the character set χ; and an RPR must match its axiom pattern to an axiom substring where any $'s or $\underline{i}$'s in the axiom pattern can be matched only by characters in the character set of that RPR. (This gives us a simple way to restrict the values that the variables may take on.) There may be many ways in which the axiom pattern can be matched; each of these will be matched in subsequent "new ways to apply P to σ ". For example, suppose we are matching the antecedent of the production $\$\wedge\{\$\bar{\wedge}\$\ \underline{\chi}\colon 01\ \underline{\pi}\colon 1\rightarrow 2\}\ \$ \rightarrow \$_1\bar{\wedge}_1\wedge\$_3\$_4$ to the axiom 0010^10x01.

*If a $ is the entire antecedent, it will match all the axioms consisting of only characters from the character set. Since the null string is not considered to be an axiom, the "named value" of the $ cannot be null in this case.

Assuming that the χ of the SR contains 0, 1, and x, the named substrings in scratch storage for each possible way of matching would be:

	$\$_1$	0010	0010	0010
(RPR)	$\$_2$	null	null	1
(RPR)	$\bar{\Delta}_1$	1	1	0
(RPR)	$\$_3$	null	0	null
	$\$_4$	0x01	x 01	x 01

If the antecedents do match, each RPR in the antecedent(s) will result in a recursive execution of the primitive automaton, with $\sigma, \sigma', \pi, \pi'$, P, ok, and β being pushed down. (This interrupts the present execution until the RPR returns.) The σ of the RPR will be initialized to the portion of the axiom which matches its axiom pattern. While the RPR will initially have only this one axiom, it may generate any number of axioms. At the end of each process step of the RPR all of the generated axioms will be checked against the RPR axiom pattern. It they all match the axiom pattern, a new process step of the RPR is begun. If any do not match, the RPR will take the fail return, and any messages which had been collected in its message buffer will be discarded without being sent. Note that if all the axioms match and another RPR process step is begun, the message buffer is <u>not</u> emptied; it will continue to collect all messages that it creates during all of its process steps. If the RPR finally takes the fail return, these messages will be discarded. If it takes a normal return, however, these messages will be put into the message buffer of the SR or RPR which contains this RPR. Since an RPR

may contain other RPR's, this means that all messages created at any level will eventually be returned to the SR (assuming that the RPR's return normally) where they will be sent at the end of the process step of that SR. If an RPR takes a fail return, no messages from that RPR or any RPR's that it contained will be transmitted. However, messages from an RPR which contains a "failing" RPR will still be sent if it and all RPR's containing it do not fail (i.e. do return normally.)

Any number of RPR's may exist in one production. They will all be evaluated, even if the first RPR takes a fail return. If an RPR fails, none of the messages it formed will be sent; however, messages from other RPR's on the same level will not be affected by the fail return.

A normal return from an RPR implies that all the axioms in σ matched the axiom pattern at the end of the last process step, and that no productions were applicable on the current process step. The values of the variables in the axiom pattern of the RPR in scratch storage will be replaced by these matches. A consequent will be constructed for each of the possible ways in which the variables in the axiom pattern can be defined using these axioms.

For example, assume that the second set of named substrings in the previous example were being used. The σ of the RPR would be initialized to 10. Since the only production does not apply (the antecedent will match only the single character 1), the RPR will return normally. There are two ways the axiom 10 can match the axiom pattern $\$\bar{\Lambda}\$$

$\$_2$:	null	1		null
$\bar{\Delta}_1$:	1	0	replace	1
$\$_3$:	0	null		0

The consequents constructed from this application of the RPR will be 00101Δ0x01 and 00100Δx01.

A fail return from a RPR indicates that an axiom in the RPR has been produced which does not match the RPR's axiom pattern. Thus the variables in the axiom pattern are potentially undefined with respect to that axiom, so that the consequent results using these variables cannot be formed. Therefore we will put into σ', of the RPR or SR containing the failing RPR, the reconstruction of the antecedents using each of the axioms in σ' of the RPR as the value associated with the RPR axiom pattern. This reconstruction can always be made since variables in the axiom pattern need not be defined.

Consider the previous example with the first set of named substrings in scratch storage being used. The only axiom of σ, namely 1, will match the only production, putting the axiom 2 into σ'. At the end of the process step, the RPR takes a fail return (since 2 is not in its character set, it cannot match $\bar{\Delta}$). Since the axiom cannot be segmented according to the axiom pattern, and since the substring which originally matched the axiom pattern is lost, the entire axiom of the RPR is substituted for the originally matching substring. The reconstructed antecedent would be 0010Δ20x01.

If the production had multiple antecedents, they may not all have failed to match; however, all antecedents will be reconstructed and will go into σ'. Thus failing RPR's may have side effects on other results. Note that a fail return from an RPR does not imply an error. An RPR may be written so that it purposely takes a fail return.

2.2.4 System Description Language

The following grammar for System Representations reflects the above modifications to PSR's. For convenience in reading, the grammar is presented informally, using print equivalents instead of the formal $\underline{i}$ notation. This grammar and a set of terminal characters $V_T = \{\underline{i}, i = 1,2,...n\}$ define a System Description Language.

<SR>::= {<axiom part><processor part>}

<axiom part>::= $\underline{\sigma:}$|$\underline{\sigma:}$ <axiom string>

<axiom string>::=<axiom> |<axiom string> and <axiom>

<axiom>::= <$\underline{j}$> | <axiom> <$\underline{j}$>

<processor part>::=<char part><production part>

<char part>::= $\underline{\chi:}$|$\underline{\chi:}$ <concatenation of a subset of $\{\underline{i}|20 \leq i \leq n\}$>

<production part>::= $\underline{\pi:}$|$\underline{\pi:}$<production string>

<production string>::=<production> |<production string> or <production>

<production>::=<mult. antecedent> → <consequent> |<mult. antecedent> →

<mult. antecedent>::=<antecedent> |<mult. antecedent> and <antecedent>

<antecedent>::= <ante element> |<antecedent> <ante element>

<ante element>::= <ante char> |<RPR>

<ante char>::= $|<$\underline{j}$>|<$\bar{\underline{j}}$>

<RPR>::= {<pattern><processor part>}

<pattern>::=<ante char>|<pattern><ante char>

<consequent>::=<cons element>|<cons element>→<cons element>

<cons element>::= <cons char>|<cons element><cons char>

<cons char>::= <j>|<$\bar{j}$> <index>|$<index>

<index>::=<nz subscript digit>|<index><subscript digit>

<nz subscript digit>::= $_1|_2|_3|_4|_5|_6|_7|_8|_9$

<subscript digit>::= $_0$|<nz subscript digit>

<$\underline{j}$>::=<$\underline{20}$>|<$\underline{21}$>|<$\underline{22}$>|...|<$\underline{n}$>

<$\bar{\underline{j}}$>::=<$\overline{\underline{20}}$>|<$\overline{\underline{21}}$>|<$\overline{\underline{22}}$>|...|<$\bar{\underline{n}}$>

2.2.5 A System Example

In order to illustrate these new concepts in our primitive automaton, let us examine the transformations of the following system which will act as a 4-bit binary clock systems which transmits the current time, measured in cycles of the clock system, to other systems.

<clock>::= {$\underline{0}$: time 0000. $\underline{\chi}$: 01

$\underline{\pi}$: time $. → time. or

time {$.$ $\underline{\chi}$: 01

$\underline{\pi}$: $0.$ → $._1 1\$_2$ or

$1.$ → $\$_1.0\$_2$} → time $\$_2$. → time.}

For additional clarity, we may want to use a syntactic notation whereby we define RPR's separately rather than in the body of the SR. The above system would then be defined using the RPR <scalar> as:

<clock>::= {σ: time 0000. χ: 01

π: time $. → time . or

time<scalar> → time $$_2$$. → time.}

<scalar>::= {$.$χ: 01

π: $0.$ → .$$_1$$1 $$_2$$ or

$1.$ → $$_1$$.0$$_2$$}

When the system begins to run, the system time is 0, as indicated by the axiom "time0000.". The first production is applicable, so the result "time." is put into σ'. The second production is also applicable, as the RPR's axiom pattern "$.$" will match the axiom substring "0000.".

When the RPR is executed, only its first production is applicable, with the first $ matching "000" and the second $ being null. The result will be ".0001". Since no other productions are applicable, and our only result does match the axiom pattern, we begin a new process step. Neither production can now be applied, however, so we take a normal return from the RPR. There is only one way the only axiom ".0001" can match the axiom pattern, i.e. $$_1$$ is null and $$_2$$ matches "0001". The two arrows in the production containing the RPR indicate that a message is to be sent, so the message "time0001." is formed and put

into the message buffer under the channel name "time.". (If there were another way to match the axiom pattern, resulting in a different value for $\$_2$, a message constructed using this value would also be put into the message buffer under the channel name "time.".) The single message in the message buffer is sent to the σ" of each of the systems in the system complex--including the σ" of the clock system itself. σ is replaced by σ', namely the single axiom "time.". This completes a system process step for our clock system.

When σ is updated from σ" at the beginning of the next process step, σ" will contain at least the message just sent by this system. The channel name "time." matches exactly the only axiom in σ, so the message "time 0001." will replace that axiom. Application of the first production will cause "time." to be put in σ'. When the second production is applied, the RPR will take three process steps before it returns normally (with "0001." being transformed to "000.0", then to ".0010") when no productions apply.

Thus a "tick" of the clock happens at the end of each process step of the clock system, with the current system time always in σ" under the channel named "time.". The message "time0001." in σ" will be replaced by "time0010." etc., so that only the most recent time is available when updating σ from σ" at the beginning of a process step. The clock will be

reset to 0000 when it overflows from 1111. Any system wishing to know the time must have "time." as an axiom in $\mathcal{O}$. At the beginning of each process step, this will be replaced by time concatenated with the current scalar value followed by a period.

We have the freedom in the design of our SR to make a natural factorization which not only makes structural characteristics easy to identify but also easy to change. Note that we could easily change the size of the clock "register" by changing the number of zeroes in the initial axiom of the <clock> system, and we could change the clock from binary to a different base by changing only the RPR <scalar>.

2.3 System Structure Invariancies

In previous sections we have presented a system Definition System which serves to define the formal universe of systems in which we are interested. To aid in discussion of structures in our formal universe, we will give names to some of its useful invariant properties. For example, the interpretation sequences of a given SR (produced by our primitive automaton) have certain properties which remain invariant to the SR transformations. The most basic invariancies are that an SR will transform only into an SR, and that all parts of the SR except for <axiom set> are invariant under all transformations.

2.3.1 System Invariancies

Many of the names we will give to invariancies in our universe are identical to terms commonly used to talk about computer systems. In

order to be able to make finer distinctions and to study more subtle relationships, we will re-define these terms here. A little thought will show that these new definitions are intuitively reasonable and do not appreciably alter the conventional notions.

Primitive processor: The System Description System primitive automaton for execution of SR's as described in section 2.2. The primitive processor is invariant for all systems.

System Description Language: The language defined by the grammar in sec. 2.2.4 and a choice of n for the vocabulary of terminal characters V_T. The sentences of the language are SR's. The System Description Language is invariant for a particular systems universe. Languages which differ only in the assignments of print equivalents are considered to be equivalent.

System Processor: An SR with <axiom part> deleted. A system processor is invariant to all transformations of an SR by the primitive processor. Processors which differ only in the ordering in the <char part> and <production part> components are considered to be equivalent.

System operator: A <production> of a processor.

System sub-processor: An <RPR> in a system operator.

System State: An <SR>.

System: A subset of a System Description Language consisting of an initial SR and all the SR's formed in the interpretation sequences

of the initial SR. The initial SR is formed by combining a processor and a system state.

System Complex: A set of SR's in the same universe consisting of at least one SR and all SR's which send messages to or receive messages from members of the set.

Process state: The <axiom string> in a set σ of an SR or RPR.

Process: A valid interpretation sequence of process states.

Simple process: A process in which each <axiom string> contains only one <axiom>.

2.3.2 Process Invariancies

In the last section we suggested that the <axiom string> in an SR may be considered a "state" of the described system. Manipulations of the axiom string by the SR operators may thus be viewed as "changes in the state" of the described system. It is possible for us to write SR's in which each such change produces a new state which has no noticable similarity to the old state. Nonetheless we are particularly interested in axioms which retain certain properties as they are transformed into new axioms. Such properties are known as invariancies. Invariancies may also occur in the <axiom string> as a whole. These are known as invariancies in the state of the described system.

For example, suppose we had an SR which described a conventional computer with a word-organized main memory. The words of memory

remain as invariant components of the machine even though their contents may change. We need some notational convention for stating this fact, because sentences like the preceding one become too cumbersome when describing complicated systems. Let us choose to delimit the words in our hypothetical machine by "*"s in an axiom. As part of our axiom set we might find something like:

*00**00**00*

Here we have described a three word memory with two digits in each word. In order to suggest the desired invariancy we might say something like:

<word>::= *<digit><digit>*

We might then say:

<digit>::= 0|1

which effectively binds the word contents to a number system of base 2.

In order to describe structural invariancies of a system's states, we will assign syntactic entities to the relevant substrings of the axioms, as above. Such syntactic entities may even be generalized to the form of a complete grammar defining a State Language whose sentences are process states.

The State Language is in no way part of the formal definition of the system. For example, we might expand the above State Language by adding the assignments:

<byte>::=<bit><bit>

<bit> ::=<digit>

These alterations to the State Language require no change in the SR to which they apply. Nonetheless these changes may improve our understanding of the SR. The State Languages which we devise for a given system are informal aids to understanding. They give us a more intuitive way to describe the possible development of a system process, given its initial definition.

3. EXAMPLES OF USE

Without attempting to exploit the system definition language we have now developed, we may illustrate its versatility and simplicity by using it to define several kinds of systems at varying levels of abstraction.

3.1 Circuit Element Systems

We will define a family of simple and familiar circuit element systems to show that simple systems can have simple SR's. In addition, since these circuit elements are more than sufficient to construct any combinatoric or sequential circuit, we show that we can define a system complex which represents a rather detailed implementation of digital systems at the circuit element level. A useful description of such circuit elements is given in the PDP manual [4].

3.1.1 Representation

We must first choose a representation of a wire carrying a signal. The discrete nature of our system representation dictates a digital rather

than analogue representation. For example, if the wire were to be considered as a delay line SR in our universe, the signal would have to be discretely delayed. We will here choose our level of abstraction so as to ignore all properties of wires except their names and a binary state of carrying a signal or not carrying a signal.

<wire>::=<wire name> . <state>

<wire name>::=<non. char>|<wire name><non. char>

<non. char>::=<any character except a period>

<state>::=<signal> |<no signal>

<signal>::=1

<no signal>::=0

For example, a wire, x, will be represented while carrying a signal by "x.1", and while not carrying a signal by "x.0". The representation of a wire will occur as an axiom in all connected systems. It will be assumed that all interacting systems "connected" to the same wire will initially have the wire in the same state. Each change in wire state will be transmitted at least once to all "connected" states.

Rather than defining particular circuit elements with specific names we will parameterize wire names by use of a notation which will generate

the unparameterized SR if actual parameters are supplied. If we have a syntactic definition of the form <a:z = $\bar{x}$> where "a" names the defined entity and x and z are wire names, we will consider x and z as "formal parameters" in the defined SR to be replaced by corresponding actual wire names. Thus our definitions are made in a generative form; the parameter replacement will produce the desired SR.

3.1.2 Combinatorial Circuits

Although many types of combinatorial circuits are possible, the circuit elements "not", "or", "and", and "stroke" are sufficient to synthesize any combinatorial circuit. The output wire state will be transmitted by each element system at the end of its cycle. If the wire z is not already in the proper state, it will be changed to the proper state by the following systems:

<not:z=$\bar{x}$> ::= {$\underline{\sigma}$: x.0 $\underline{\chi}$: x.01 $\underline{\pi}$: x.0 → z.1 → z.0 or x.1 → z.0 → z.1 or \$ → $\$_1$}

<or:z=x ∨ y> ::= {$\underline{\sigma}$: x.0 and y.0 $\underline{\chi}$: xy.01 $\underline{\pi}$: $\bar{\Delta}$.1 → z.1 → z.0 or x.0 and y.0 → z.0 → z.1 or \$ → $\$_1$}

<and:z=x ∧ y> ::= {$\underline{\sigma}$: x.0 and y.0 $\underline{\chi}$: xy.01 $\underline{\pi}$: $\bar{\Delta}$.0 → z.0 → z.1 or x.1 and y.1 → z.1 → z.0 or \$ → $\$_1$}

<stroke:z=x|y> ::= {$\underline{\sigma}$: x.0 and y.0 $\underline{\chi}$: xy.01 $\underline{\pi}$: $\bar{\Lambda}$.0 → z.1 → z.0 or x.1 and y.1 → z.0 → z.1 or \$ → $\$_1$}

The structural similarity between "stroke" and "and" systems is readily apparent. Although our SR structures are far more powerful than is required for such simple systems, note that these simple systems can be described without obscuring structural similarities.

The sequential circuit corresponding to the Boolean expression $C = A \wedge \bar{B}$ may thus be implemented by the two systems

$$\langle :c=a \wedge \bar{b}\rangle ::= \langle \text{not}:i=\bar{b}\rangle \langle \text{and}:c=a \wedge i\rangle$$

where wires a and b are inputs and c is the output. An "internal" wire, i, is required to connect the two systems. The corresponding SR's are:

$\{\underline{\sigma}: b.0 \underline{\chi}: b.01 \underline{\pi}: b.0 \rightarrow i.1 \rightarrow i.0$ or $b.1 \rightarrow i.0 \rightarrow i.1$ or $\$ \rightarrow \$_1\}$

$\{\underline{\sigma}: a.0$ and $i.0 \underline{\chi}: ai.01 \underline{\pi}: \bar{\Delta}.0 \rightarrow c.0 \rightarrow c.1$ or $a.1$ and $i.1 \rightarrow c.1 \rightarrow c.0$

or $\$ \rightarrow \$_1\}$.

3.1.3 Sequential Circuits

As an example of the wide variety of sequential circuit elements that may be designed, the following should be illustrative and are sufficient for system synthesis.

A real time "clock pulse" system can be defined as

$\langle \text{pclock}:x\rangle ::= \{\underline{\sigma}: x.0 \underline{\chi}: 01 \underline{\pi}: x.0 \rightarrow x.1 \rightarrow x.0$ or $x.1 \rightarrow x.0 \rightarrow x.1$ or $x.\bar{\Delta} \rightarrow x.\bar{\Delta}_1\}$

which broadcasts a change of state on wire x at the end of each system process step. The character Δ is assumed to be in V_T and is used as a one character variable. A real time "clock" system that always broadcasts the current time in unary notation is given by

$\langle \text{uclock}:x\rangle ::= \{\underline{\sigma}: x. \underline{\chi}: 1 \underline{\pi}: x\ \$. \rightarrow x\ \$_1 1. \rightarrow x.$ or $x\ \$. \rightarrow x.\}$.

Any system that requires current time to the resolution possible in the <uclock:x> system may receive it by adding an axiom "x." and a production "x$.→x." which causes the latest time at the start of each system cycle to be available. A similar clock using binary notation was defined in section 2.2.5.

A simple "RS" flipflip may be defined as two interacting systems:

<RS:x,r,s>::=<ffr:x,r><ffs:x,s>

<ffr:x,r>::={σ:r.0 χ: r.01 π: r.1→x0.1→x0.0 or r.1→x1.0→x1.1 or $→$$_1$}

<ffs:x,s>::={σ: s.0 χ: s.01 π: s.1→x1.1→x1.0 or s.1→x0.0→x0.1 or

$→$$_1$}

or as one system:

<RS:x,r,s>::={σ: r.0 and s.0 χ: rs01

π: r.1→x0.1→x0.0 or r.1→x1.0→x1.1 or

s.1→x1.1→x1.0 or s.1→x0.0→x0.1 or $\bar{\Lambda}.\bar{\Lambda} \rightarrow \bar{\Lambda}_1.\bar{\Lambda}_2$}

where r and s are reset and set input wires. x0 and x1 are the reset and set output wires. r and s should not be in the ".1" state simultaneously or the flipflop will oscillate.

The state of the flipflop (i.e. set or reset) does not appear in this system since, by our representation of a wire, each connected system will "remember" the current state or a wire until it is changed.

A "toggle" flip flop that simply changes state when a pulse is received via wire t is given by:

<tff:x,t>::={σ: t.0 χ: t.01

π: t.1 and t → x0.$\bar{1}$ → x0.0 or t.1 and t → x0.0 → x0.1 or

t.1 and t → x1.1 → x1.0 or t.1 and t → x1.0 → x1.1 or

t.0 → t}

where t must go to state ".0" before a new toggle will occur. Similarly, a "JK" flipflop which toggles if reset (j) and set (k) signals occur simultaneously may be defined as

<JK:x,j,k>::={σ: j.0 and k.0 χ: jk01

π: j.1 and j → x0.1 → x0.0 or j.1 and j → x1.0 → x1.1 or

k.1 and k → x1.1 → x1.0 or k.1 and k → x0.0 → x0.1

or $\bar{\Delta}$.0 → $\bar{\Delta}_1$ or $\bar{\Delta}.\bar{\Delta}$ → $\bar{\Delta}_1.\bar{\Delta}_2$}

which is identical to <RS:x,r,s> except for the production ($\bar{\Lambda}$.0 → $\bar{\Lambda}_1$) of conditioning signals for the recognition of new j.1 and k.1 signals.

The following "pulse extender" system will, upon receipt of a t signal, set x and remain set for four cycles after the end of the t pulse. The system will then reset unless a new t pulse is received first. A new t pulse reinitiates the cycle.

<pe4:x,t>::={σ: x.0 and t.0 χ: tx01

π: t.1 → x.1 → x.0 or t.1 → t1111 or $\bar{\Lambda}.\bar{\Lambda}$ → $\bar{\Lambda}_1.\bar{\Lambda}_2$ or

t.0 and t$1 → t$$_1$ or t.0 and t → x.0 → x.1}

3.2 Automata Systems

Although the representation of our systems in terms of automata is, in general, quite complex, the converse is not true. Most forms of

automata can be translated trivially into systems of our universe. The added power of our systems frequently provides far simpler equivalent systems.

3.2.1 Finite State Machines

A finite state machine (FSM) may be defined in many ways,* one of which is by

a) a set of input signals $s_j \in S$, $j = 1, \ldots, n_s$

b) a set of internal states $q_j \in Q$, $j = 1, \ldots, n_q$

c) a set of output signals $r_j \in R$, $j = 1, \ldots, n_r$

d) a state function $G(q_i, s_j) = g_{ij} \in Q$

e) a response function $F(q_i, s_j) = f_{ij} \in R$

where S, Q and R need not be disjoint sets. A FSM may be represented by a set of quadruples $(q_i, s_j, g_{ij}, f_{ij})$.

A FSM may also be defined as the SR

<FSM x, y> ::= {σ: x and Δ q_0 Δ χ: x <state symbols><input symbols>

π: $\bar{\Delta} \rightarrow$ x or x and $\Delta\bar{\Delta}\Delta \rightarrow \Delta\bar{\Delta}_1\Delta$ or

$\Delta q_i \Delta$ and $s_j \rightarrow \Delta g_{ij} \Delta$ or $\Delta q_i \Delta$ and $s_j \rightarrow f_{ij} \rightarrow$ y}

<state symbols> ::= <concatenation of elements of Q>

<input symbols> ::= <concatenation of elements of S>

where x is the input channel name and output is transmitted to y. The shorthand notation of the last line implies the set of productions obtained by substituting the appropriate values of q_i, s_j, f_{ij}, and g_{ij} from each

*See sec. 2 of Minsky [2] for more details.

quadruple. After an input signal s_j is received on the channel x, the output signal r_j with be transmitted on the channel y at the end of the next process step. The system idles after output until a new input signal appears. Note that a new input signal must not be sent until the previous output is received because of the unknown delay prior to initiation of the next process step.

An equivalent system (except for delay in output) formed by encoding F and G as RPR's is given by:

<FSM:x,y> ::= {$\underline{\sigma:}$ x $\underline{\text{and}}$ $\Delta q_0 \Delta$ $\underline{\chi:}$ <state symbols><input symbols><output symbols>

$\underline{\pi:}$ $\bar{\Delta} \rightarrow$ x $\underline{\text{or}}$ x $\underline{\text{and}}$ $\Delta\bar{\Delta}\Delta \rightarrow \Delta\bar{\Delta}_1\Delta$ $\underline{\text{or}}$ $\Delta\bar{\Delta}\Delta$ $\underline{\text{and}}$ $\bar{x} \rightarrow \Delta\bar{\Delta}_1\bar{x}_1\Delta$ $\underline{\text{or}}$

Δ<G>$\Delta \rightarrow \Delta\ \$_1\Delta$ $\underline{\text{or}}$ Δ<F>$\Delta \rightarrow\ \$_1 \rightarrow$ y}

<G> ::= {$ $\underline{\chi:}$ <state symbols><input symbols> $\underline{\pi:}$ $q_i s_j \rightarrow g_{ij}$}

<F> ::= {$ $\underline{\chi:}$ <state symbols> <input symbols> <output symbols> $\underline{\pi:}$ $q_i s_j \rightarrow f_{ij}$}

<output symbols> ::= <concatenation of elements of R>

where the same shorthand notation is used. Two process steps of the SR are required to generate the output.

The first form of SR above corresponds to a matrix representation of a FSM while the second form represents a functional representation. FSM's are also represented in state diagram form and we should be able to find a similar representation in our universe. We may do this by defining a separate system for each state, q_i, for $i \neq 0$ as

$\langle state{:}q_i,x,y\rangle ::= \{\underline{\sigma}{:}\, q_i\ \underline{\chi}{:}\langle state\ symbols\rangle\langle input\ symbols\rangle\langle output\ symbols\rangle$

$\underline{\pi}{:}\ \bar{x} \rightarrow q_i\ \underline{or}\ \ x\$ \rightarrow x\ \underline{or}\ x\bar{\Delta} \rightarrow \bar{\Delta}_1 \rightarrow y\ \underline{or}\ s_j \rightarrow xf_{ij} \rightarrow g_{ij}\}$

using the same notation as above. The system for state q_0 is identical except that the initial axiom is "x" rather than "q_0". Output is delayed by two process steps, one in the old state and one in the new state. This allows the new state to be prepared for the next input symbol before it appears. The input "channel", x, is passed to the next state so that the next input will appear there.

We might find it convenient to define a given FSM as a system complex involving all three of the above forms. A much more interesting possibility is to use representation dependent systems. Each of the above systems is independent of the choices for S, Q, and R. If we define a "two moment" delay machine by $g(q_{ij},s_k) = q_{jk}$ and $f(q_{ij},s_k) = r_i$ where i,j, and k are 0 or 1 we have the SR:

$\langle 2M{:}x,y\rangle ::= \{\underline{\sigma}{:}\, x\ \underline{and}\ \Delta q_{00}\Delta\ \underline{\chi}{:}\ xq_{00}q_{01}q_{10}q_{11}s_0s_1$

$\underline{\pi}{:}\quad \bar{\Delta} \rightarrow x\ \underline{or}\ x\ \underline{and}\ \Delta\bar{\Delta}\Delta \rightarrow \Delta\bar{\Delta}_1\Delta\ \underline{or}$

$\Delta q_{00}\Delta\ \underline{and}\ s_0 \rightarrow \Delta q_{00}\Delta\ \underline{or}\ \Delta q_{00}\Delta\ \underline{and}\ s_0 \rightarrow r_0 \rightarrow y\ \underline{or}$

$\cdots$

$\Delta q_{11}\Delta\ \underline{and}\ s_0 \rightarrow \Delta q_{10}\Delta\ \underline{or}\ \Delta q_{11}\Delta\ \underline{and}\ s_0 \rightarrow r_1 \rightarrow y\ \underline{or}$

$\Delta q_{00}\Delta\ \underline{and}\ s_1 \rightarrow \Delta q_{01}\Delta\ \underline{or}\ \Delta q_{00}\Delta\ \underline{and}\ s_1 \rightarrow r_0 \rightarrow y\ \underline{or}$

$\cdots$

$\Delta q_{11}\Delta\ \underline{and}\ s_1 \rightarrow \Delta q_{11}\Delta\ \underline{or}\ \Delta q_{11}\Delta\ \underline{and}\ s_1 \rightarrow r_1 \rightarrow y\}.$

The Δ's delimit what is essentially a shift register. Input signals are shifted into the right side, and the leftmost signal is sent as a response. If we assume $s_i = r_i = i$ and $q_{ij} = ij$ we may define the same system by the SR

$\langle 2M{:}x, y\rangle ::= \{\underline{\sigma}{:}x \text{ } \underline{\text{and}} \text{ } \Delta 00\Delta \text{ } \underline{\chi}{:} \text{ } 01$

$\underline{\pi}{:} \bar{x} \rightarrow x \text{ } \underline{\text{or}} \text{ } \bar{x} \text{ } \underline{\text{and}} \text{ } \Delta\bar{\Delta}\bar{\Delta}\Delta \rightarrow \Delta\bar{\Delta}_2\bar{x}_1\Delta \text{ } \underline{\text{or}} \text{ } x \text{ } \underline{\text{and}} \text{ } \Delta\bar{\Delta}\bar{\Delta}\Delta \rightarrow \bar{\Delta}_1 \rightarrow y\}.$

Although the FSM representation independent forms get much more elaborate for an "n moment" delay machine, using our representation dependent form we have the SR

$\langle n\,M{:}x, y\rangle ::= \{\underline{\sigma}{:}$ x and $\Delta\langle$string of n zeroes$\rangle\Delta$ $\underline{\chi}{:}$ 01

$\underline{\pi}{:} \bar{x} \rightarrow x \text{ } \underline{\text{or}} \text{ } \bar{x} \text{ } \underline{\text{and}} \text{ } \Delta\bar{\Delta}\$\Delta \rightarrow \Delta\$_1\bar{x}_1\Delta \text{ } \underline{\text{or}} \text{ } \bar{x} \text{ } \underline{\text{and}} \text{ } \Delta\bar{\Delta}\$\Delta \rightarrow \bar{\Delta}_1 \rightarrow y\}$

which is scarcely more complicated than the previous two moment form and makes the notion of a shift register more apparent.

3.2.2 Turing Machines

A Turing machine, TM, may be defined by*

a) a set of input symbols $s_j \in S, j = 0, \ldots, n_s$

where s_0 is the blank symbol

b) a set of internal states $q_j \in Q, j = 0, \ldots, n_q$

c) a set of output symbols $r_j = s_j, j = 1, \ldots, n_s$

d) a state function $G(q_i, s_j) = g_{ij} \in Q, \quad j \neq 0$

e) a response function $E(q_i, s_j) = f_{ij} \in S, \quad j \neq 0$

f) a move function $D(q_i, s_j) = d_{ij} \in \{d_0, d_1\}$

where $d_0 \supset$ left shift and $d_1 \supset$ right shift

j) an $\langle$initial tape$\rangle ::= \langle$string$\rangle$ s_0q_0 $\langle$input symbol$\rangle\langle$string$\rangle$

$\langle$string$\rangle ::= \langle$null$\rangle \mid \langle$string$\rangle\langle$input symbol$\rangle$

where S, Q, R, and D need not be disjoint sets.

---

*See Sec. 6 of Minsky [2] for details.

The TM may be represented by an <initial tape> and a set of quintuples $(q_i, s_j, g_{ij}, f_{ij}, d_{ij})$ in a form similar to the quadruples for an FSM.

A TM may also be represented by an SR as follows:

<TM:>::= { $\underline{\sigma:}$ <initial tape> $\underline{\chi:}$ <input signals> $\underline{\pi:}$ <TM productions> }

<TM productions>::=<TM quint> | <TM productions> <TM quint>

where <TM quint> is a pair of productions for each quintuple of the TM. If the quintuple moves left, the productions are

$\$\bar{\Delta} s_o q_i s_j \$ \rightarrow \$_1 s_o g_{ij} \bar{\Delta}_1 f_{ij} \$_2$ or $s_o q_i s_j \$ \rightarrow s_o g_{ij} s_o f_{ij} \$_1$ and if it moves right the productions are

$\$ s_o q_i s_j \bar{\Delta} \$ \rightarrow \$_1 f_{ij} s_o g_{ij} \bar{\Delta}_1 \$_2$ or $\$ s_o q_i s_j \rightarrow \$_1 f_{ij} s_o g_{ij} s_o$

A TM which checks an initial tape, <pstring>, consisting of a sequence of left and right parenthesis to decide if they are well formed (i.e. may be paired off from inside to outside) is easy to construct, using 22 productions, as an SR of the form above. By making better use of our primitive automaton we may define a parenthesis checking system SR, using only one production, as

<pc:>::= { $\underline{\sigma:}$ <pstring> $\underline{\chi:}$ x( ) $\underline{\pi:}$ { $) $\underline{\chi:}$ x ($\underline{\pi:}$ $([$\underline{\chi:}$ x $\underline{\pi:}$]) $\rightarrow \$_1$ x $\$_2$ x } $ $\rightarrow$ }

which will terminate with a string of x's if well formed or, if not well formed, either a string of x's and unmatched left parenthesis or a null string.

3.3 Programming Language Semantics

A system for the definition of simple precedence languages and their semantics was described by Wirth [5], who illustrated his method by defining

a parser/interpreter for a simple language. We will define an equivalent system.

3.3.1 Parsing/interpreting System

The parsing/interpreting system for simple precedence grammars and languages, shown below, displays very clearly the essential notions of simple precedence parsing. The first production simply scans to the right for the leftmost $\gtrdot$ relation. The second production then scans to the left to find the rightmost $\lessdot$ relation. <operation i,j> then reduces the $\lessdot \ldots \cdot>$ delimited phrase and the first production resumes scanning. The system will halt if the <sentence> is not well formed.

<p/i:> ::= {σ: Δ<sentence>Δ<stack>Δ χ:<non Δ>

π: {$ΔΔ̄Δ$ χ:<non Δ> π: <g matrix>} Δ $ Δ → $$_1$ Δ̄ $_1$ Δ $$_2$ \ $$_3$ \ or

{$Δ̄ΔΔ $ Δ $ χ:<non Δ> π: <l matrix>} Δ $ Δ → $$_1$ Δ Δ̄ $_1$ $$_2$ \ $$_3$ \ $$_4$ \ or

<operation i,j>}

<sentence> ::= sentence of language to be interpreted. Left and right terminating symbols are assumed.

<stack> ::= null if only parsing, otherwise defining the current state of interpretation. It is implicitly defined by <operation i,j>

<non Δ> ::= concatenation of language vocabulary which must not contain Δ character. x_i will represent the i^{th} character.

<g matrix>::= <g elmt> |<g matrix> or <g elmt>

<g elmt>::= $ $x_i \Delta x_j$ $ \rightarrow $ $_1 \Delta x_i \Delta x_j$ $ $_2$

One <g elmt> is generated for every pair of values of i,j such that x_i has greater precedence than x_j.

<l matrix>::= <l elmt> |<l matrix> or <l elmt>

<l elmt>::= $ $x_i \Delta x_j$ $ Δ $ \rightarrow $ $_1 x_i \Delta\Delta x_j$ $ $_2 \Delta$ $ $_3$

One <l elmt> is generated for every pair of values of i,j such that x_i has lower precedence than x_j.

<operation i,j>::= <reduce i,j>|<interpret i,j>

<reduce i,j>::= $$\Delta\Delta<p_j>\Delta$ Δ $\Delta \rightarrow$ $ $_1 x_i \Delta$ $ $_2 \Delta$ $ $_3 \Delta$

One <reduce i,j> is generated for each x_i phrase having a null interpretation rule where x_i::=<p_j>

<p_j>::= right hand side of an x_i production in grammar

<interpret i,j>::= {$$\Delta\Delta<p_i>\Delta$ Δ Δ χ:<non Δ> π: <x_i ops>} $\rightarrow$ $ $_1 x_i \Delta$ $ $_2 \Delta$ $ $_3 \Delta$

One <interpret i,j> may be generated for each x_i::=<p_i> phrase having a non-null semantic interpretation rule. Generalizations leading to a smaller number of non-standard operators are frequently possible.

<x_i ops>::= <production string>

which defines appropriate transformations of <stack> upon the reduction of <p_j> to x_i.

Since the initial axiom Δ <sentence> Δ <stack> Δ contains just three Δ's, only the first production is applicable. This production will move the leftmost Δ to the right until it creates $x_i \Delta x_j$ where $x_i \gtrdot x_j$. It then inserts an extra Δ before x_i to delimit this point, causing the RPR to take a fail return.

Now that the axiom contains four Δ's, only the second production is applicable. This production moves the extra Δ (inserted by the first production) to the left until it finds $x_i \Delta x_j$ where $x_i \lessdot x_j$. This is marked by inserting another Δ after x_i.

There are now five Δ's in the axiom, so only <operation i,j> can possibly be applied. The leftmost reducible phrase of the sentence is now delimited by $\Delta\Delta$ on the left and by Δ on the right. The only <operation i,j> which is applicable is the one where j, the righthand side of a production in the grammar, matches the delimited phrase exactly. <operation i,j> will reduce this phrase to x_i, the lefthand side of the production, and will remove the $\Delta\Delta$, so the axiom contains only three Δ's again. If the sentence is being interpreted instead of simply being parsed, the appropriate manipulations on the <stack> will also take place.

3.3.2 Simple Language Definition

The simple precedence language defined by Wirth [5] can be defined in terms of an SR, <p/i:>, generated by the additional definitions below.

<non Δ>::= ; <no;Δ>

<no;Δ>::= $x_0 x_1 \ldots x_{18} x_{19} \lambda, \leftarrow + - x / ( )$ 0123456789

where a mapping from Wirth's multisymbol vocabulary elements as in Table 2 has been made.

<g elmt>::= } defined from precedence matrix

<l elmt>::= } of the simple language

<stack>::= <null>

<sentence>::= x_0 <program in simple language> x_0

<operation i,j>::= as defined in Table 3.

Just as in the Wirth definition, certain operations are assumed to be available as "service" operations and are not explicitly defined; thus they may be considered as "primitive" operations. For a complete definition these primitives should (and could) be defined as additional operations in our system. The remaining primitive operations are defined in Table 4.

Facilities for the easy implementation of the primitives have been provided in the <stack>. Only <stack> sentences as produced below will ever occur during the execution of the system. This is an assertion about the system, not a part of its definition.

<stack> ::= <null> |; <elmt> |<stack>; <elmt>

<elmt> ::= <v> |<i> |<d>

<i> ::= λ

<d> ::= λ :<v>

<v> ::= number in internal representation

<stack> will be transformed by the primitives as a <stack> of identifier (<i>), declaration (<d>), and value (<v>) entries. The primitives operate only on the top one or two entries of the <stack> and leave their result on the top of the <stack>. The resulting form is suitable for the application of the first operator of <p/i:>. The initial form (prior to primitive execution) is always Δ<op>$\Lambda$$\Delta$$\Lambda$$\Lambda$ and the final form (just after primitive <op> execution) is always $\Lambda$$\Delta$$\Lambda$. The operator will reduce the original phrase and transform the <stack> appropriately.

Table 2: Character equivalents for multisymbol vocabulary elements.

Char.	Equiv.	Char.	Equiv.	Char.	Equiv.	Char.	Equiv.
x_0	⊥	x_5	decl	x_{10}	term	x_{15}	digit
x_1	program	x_6	statement	x_{11}	term-	x_{16}	new
x_2	block	x_7	statlist	x_{12}	factor	x_{17}	begin
x_3	body	x_8	expr	x_{13}	var	x_{18}	end
x_4	body-	x_9	expr-	x_{14}	number		

Table 3: Semantic Rules

<phrase i>	<operation i , j>
x_5::= $x_{16}\lambda$	$\$\Delta\Delta x_{16}\lambda\Delta\ \$\Delta\ \$\Delta \rightarrow \$_1 x_5\ \Delta\ \$_2\ \Delta\ \$_3;\lambda:\Delta$
x_{15}::=0|...|9	$\$\Delta\Delta\{\bar{\Delta}\ \underline{\chi:}\ 0123456789\ \underline{\pi:}\}\Delta\$\Delta\ \$\Delta \rightarrow \$_1\Delta x\ \text{to}\ v\Delta x_{15}\Delta\$_2\Delta\$_3;\bar{\Delta}_1\Delta$
x_{14}::=$x_{14}x_{15}$	$\$\Delta\Delta x_{14}x_{15}\Delta\$\Delta\$\Delta \rightarrow \$_1\Delta\times 10 + \Delta x_{14}\Delta\$_2\Delta\$_3\Delta$
x_{13}::=λ	$\$\Delta\Delta\lambda\Delta\$\Delta\$\Delta \rightarrow \$_1 x_{13}\ \Delta\ \$_2\ \Delta\ \$_3;\lambda\Delta$
x_{12}::=x_{13}	$\$\Delta\Delta x_{13}\Delta\$\Delta\$\Delta \rightarrow \$_1\Delta\text{Id}\Delta x_{12}\Delta\ \$_2\Delta\ \$_3\Delta$
x_{11}::=x_{11}/x_{12} x_{11}::=$x_{11}\times x_{12}$	$\$\Delta\Delta x_{11}\{\bar{\Delta}\ \underline{\chi:}\ \times/\ \underline{\pi:}\}\ x_{12}\Delta\ \$\Delta\ \$\Delta \rightarrow \$_1\Delta\bar{\Delta}_1\Delta x_{11}\Delta\ \$_2\Delta\ \$_3\Delta$
x_9::=-x_{10}	$\$\Delta\Delta -x_{10}\Delta\ \$\Delta\ \$\Delta \rightarrow \$_1\Delta\text{neg}\Delta x_9\Delta\ \$_2\Delta\ \$_3\Delta$
x_9::=x_9 + x_{10} x_9::=x_9 - x_{10}	$\$\Delta\Delta x_9\{\bar{\Delta}\ \underline{\chi:}\ +\ -\ \underline{\pi:}\}\ x_{10}\Delta\ \$\Delta\ \$\Delta \rightarrow \$_1\Delta\bar{\Delta}_1\Delta x_9\Delta\ \$_2\Delta\ \$_3\Delta$
x_6::=$x_{13} \leftarrow x_8$	$\$\Delta\Delta x_{13} \leftarrow x_8\Delta\ \$\Delta\ \$\Delta \rightarrow \$_1\Delta\text{st}\Delta x_6\Delta\ \$_2\Delta\ \$_3\Delta$
x_4::=$x_5;x_4$	$\$\Delta\Delta x_5;x_4\Delta\ \$\Delta\ \$;\{\$\ \underline{\chi:}\ <\text{no};\Delta>\ \pi:\}\Delta \rightarrow \$_1 x_4\Delta\ \$_2\Delta\ \$_3\Delta$
All others	use form for <reduce i,j> as shown below
x_i::=$<p_j>$	$\$\Delta\Delta<p_j>\Delta\ \$\Delta\ \$\Delta \rightarrow \$_1 x_i\Delta\ \$_2\ \Delta\ \$_3\Delta$

Table 4: Unrepresented primitives

Operator	Interpretation
xtov	Convert digit to internal number representation
×10+	$v_{t-1} := 10 \times v_{t-1} + v_t$; t:=t-1
ld	$v_t :=$ current value associated with λ on top of <stack>
st	store v_t in current value associated with λ in v_{t-1}; t:=t-1
×/+-	$v_{t-1} := v_{t-1}$ <op> v_t; t:=t-1
neg	$v_t := -v_t$

4. ACKNOWLEDGEMENTS

The formal system described here was developed as a part of a project for the design of high level hardware-independent systems and languages, and as a tool for studying and teaching system structures. It has been used in systems courses for three years and has profited from many discussions with students in these courses. Although individual acknowledgements of all these students is not feasible, special acknowledgements should be given to Mr. Russell Blake and Mr. Stephen Schleimer for discussions of systems communication and equivalence.

REFERENCES

1. Simon, H. A. The Sciences of the Artificial. M.I.T. press, Cambridge, Massachusetts, 1969.

2. Minsky, M. Computation Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, N. J., 1967.

3. Galler, B. and Perlis, A. A View of Programming Languages. Addison-Wesley, Reading, Mass., 1970.

4. Digital Logic Manual, Digital Equipment Corporation, Maynard, Mass., 1970.

5. Wirth, N. and Weber, H. "EULER: A Generalization of ALGOL, and its Formal Definition: Part 1", CACM 9 #1 Jan. 1966.