COMPLEXITY OF EXPRESSIONS ALLOWING CONCURRENCY

W. F. Ogden[1]
Case Western Reserve University

W. E. Riddle[2]
University of Colorado

W. C. Rounds[3]
University of Michigan

0. SUMMARY

We study some consequences of the formal language approach to modelling software system behavior for the case of asynchronous, concurrent subsystems. We use the formal language shuffle operation to give an "algebraic" definition of semantics for a simple (structured) concurrent programming language and prove that the use of this operation is necessary. Having established this necessity, we investigate other types of behavioral expressions which use the operation and show that the analysis problem for these expressions is either undecidable or intractable. The results provide some limitations, for example, on the path expression method of system behavior analysis. Our lower bound proofs involve the use of synchronization symbols, which seem to be a formal language analogue of semaphores.

1. INTRODUCTION

The parbegin - parend and fork-join control structures express concurrent program behavior. They implicitly contain the idea of interleaving or shuffling streams of operations from component processes. This interleaving is also implicit in the path expression formalisms (Campbell and Habermann [2]) and event expressions (Riddle [14]) and, of course, the various Petri net schemes (Peterson [11]). This paper is concerned with algebraic expressions which explicitly use the shuffle operation [4, p. 108] to express concurrency. We give an example of this use by providing a language-theoretic semantics for simple structured parallel programs. We then study the recognition problem for languages defined by such expressions. Our conclusion is, in brief, that shuffling is complex.

This paper has two main sections. The first part, after preliminary definitions, illustrates the use of shuffling in giving semantics for concurrent programs. Many authors, of course, have given such semantics: among them are Horning and Randell [7], Owicki [10], and Lipton [8], for example. Our approach differs from these authors' by using algebraic expressions with explicit concurrency. We separate control from data in the manner of program schemata, so that a program expression defines a certain set of abstract execution sequences. These sequences of operations are then interpreted over a state space. The (I-O) behavior of such a program is given by choosing an interpretation and picking off pairs of states associated with the beginnings and ends of all legitimate execution sequences, thus obtaining a binary relation on the state space. In contrast, Pratt [12] views a program as being defined by such a relation. We show, that for expressing concurrency, the execution sequence approach is somehow necessary, because the I-O behavior of the result of shuffling two programs cannot be defined from the I-O behavior of the components using the

[1]Department of Computer & Information Science, Case Western Reserve University, Cleveland, Ohio 44106

[2]Department of Computer Science, University of Colorado, Boulder, Colorado 80309

[3]Department of Computer & Communication Sciences, University of Michigan, Ann Arbor, Michigan 48109

standard relational operators (Codd [3]). Our result uses a theorem of Bancilhon [1] on relational definability which is itself a form of Beth's definability theorem [15, page 81].

The second part of this paper deals with the complexity of the recognition problem for languages defined by expressions with concurrency. Riddle [14] defines event expressions, which denote behavior of programs written in a software description language. Similarly, Campbell and Habermann [2] use path expressions as definitions of desired program behavior. Riddle's scheme is intended for off-line analysis of global system behavior, while the path expressions are to be compiled into code which checks allowable activity during execution. In both cases, however, it is natural to ask whether or not a string of symbols is in the set denoted by the expression. We prove that event expressions in their full generality can denote any recursively enumerable set, and thus are not amenable to algorithmic analysis. We then show that extensions of path expressions to the context-free case (standard path expressions denote regular sets) will have to be defined with some care, because there exist (deterministic) context-free languages whose shuffle is NP-complete.

Our intractibility results use the technique of synchronization symbols, introduced by Riddle in event expressions. These symbols are a formal language analogue of semaphores. We use them to force concurrent systems to simulate Turing machine behavior. The technique is reminiscent of one used by Lipton [9] to show an exponential space lower bound on the vector addition reachibility problem and, for the NP result, an extension of the methods used by Greibach [6] to exhibit NP-complete quasi-realtime languages. There appear to be many different ways to use these symbols in such proofs.

2. SHUFFLING

Let L_1 and L_2 be subsets of Σ^*. Then

$$L_1 \Delta L_2 = \{x_1y_1\ldots x_ny_n \mid x_1\ldots x_n \in L_1;\ y_1\ldots y_n \in L_2\}$$

where the x_i's or y_i's can be null. (An early reference to this operation is Ginsburg and Spanier [5].) It is easy to show that regular sets are closed under shuffle and that the shuffle of a regular set and a CFL is a CFL. The CFL's are not closed under shuffle. (Theorem 3 shows how bad things can get.)

We also consider the unary dagger operation on languages. Dagger is the Kleene closure of shuffle:

$$L^{\dagger} = \bigcup_{n \geq 0} L^{(n)}$$

where $L^{(n)} = L\Delta L\Delta|\ldots\Delta L$ (n times). The regular sets are not closed under dagger.

Structured parallel programs

We give an application of shuffling to programming language semantics. We consider a language for which expressions with shuffling form a concise way of defining meanings. Figure 0 is an example.

```
sem:=1;
result:=null;
parbegin
   writer: repeat-indefinitely
      if sem=0 then
         if result ≠ null then
            begin
               buffer:=result;
               write(buffer);
               sem:=1
            end
   end writer;
   read & process: repeat-indefinitely
      if sem=1 then
         begin
            read(buffer);
            sem:=0;
            <code to compute result>
         end
   end read & process
parend
```

Fig. 0

This code represents overlapped execution of printing with processing, using a shared buffer, with a semaphore to enforce mutual exclusion of reading and writing. We are not concerned with efficiency or even correctness here, however; only with giving a meaning to the statements.

The atomic operations in this program are the assignments, reads, and writes; the atomic tests are the conditions mentioned in the if-then

statements. We assign to each atomic operation a unique symbol from an alphabet Σ and each test a symbol from an alphabet B (these alphabets need not be finite). The control structures (if-then, sequencing, repeat, etc.) are then regarded as operators on sets of strings of symbols and so give rise to program expressions. A program expression thus denotes a certain language (set of strings of operation and test symbols). By interpreting each symbol, and hence string, as a real operator or test, we then get the meaning of the program. This is of course the method of program schemata for a class of programs involving concurrency.

Formally, the class of program expressions over Σ and B is given by a recursive definition:

(i) each element of Σ is a program expression;

(ii) if π_1 and π_2 are program expressions, and b ε B, then the following are program expressions:

$\pi_1; \pi_2$

if b then π_1

if b then π_1 else π_2

while b do π_1

π_1 or π_2

repeat-indefinitely (π_1)

parbegin $\pi_1; \pi_2$ parend

To illustrate, let us give the tree structure of our example program regarded as a program expression (Fig. 1). This tree emphasizes the role of control structures as operators (on the results of subtrees).

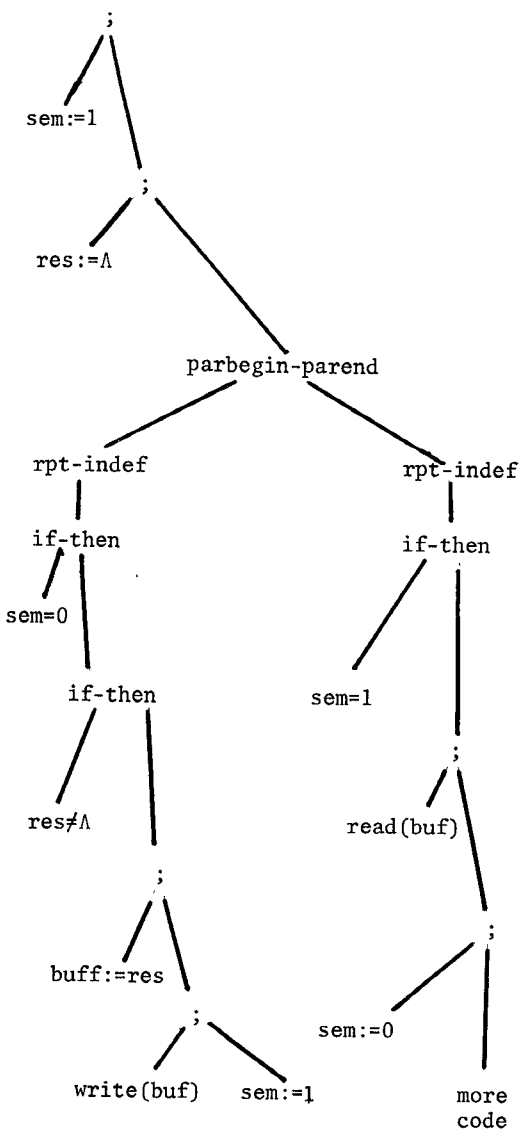

Fig. 1

The first stage in computing the meaning of a program expression is to define its associated language; the second is to interpret this language over a data space. Thus the language of an expression π is defined, following the recursive definition of π.

(i) if $\pi \in \Sigma$ then $L(\pi) = \pi$;

(ii) if $L(\pi_1)$, $L(\pi_2)$ are defined, and b ε B, then

$L(\pi_1;\pi_2) = L(\pi_1)\cdot L(\pi_2)$;

$L(\text{if b then } \pi_1) = b\cdot L(\pi_2) \cup \{\bar{b}\}$;

$L(\text{if b then } \pi_1 \text{ else } \pi_2) = b\cdot L(\pi_1) \cup \bar{b}\cdot L(\pi_2)$;

$L(\text{while b do } \pi_1) = (b\cdot L(\pi_1))^*\cdot\bar{b}$;

$L(\pi_1 \text{ or } \pi_2) = L(\pi_1) \cup L(\pi_2)$;

$L(\text{repeat-indefinitely}(\pi_1)) = L(\pi_1)^*$;

$L(\text{parbegin } \pi_1;\pi_2 \text{ parend} = L(\pi_1) \Delta L(\pi_2)$.

Comments: (1) $\bar{b}$ is a single symbol denoting the negation of the predicate b; (2) the use of $\bar{b}$ in the definition of if-then reflects the desire to do the null operation when b is false; (3) we do not employ infinite sequences of operations - thus "repeat-indefinitely" is not the same as "do-forever" or "while true do".

For the second stage, we define the input-output

relations denoted by a program expression. We are given a set D of states; for each operation symbol σ in Σ a relation $R(\sigma) \subseteq D \times D$; and for each $b \in B$ a subset $S(b)$ of D. Then for each $x \in (\Sigma \cup B)^*$ we define a relation $I(x)$:

$I(\sigma) = R(\sigma)$ for $\sigma \in \Sigma$;

$I(b) = id_D \uparrow S(b)$ for $b \in B$;

$I(xy) = I(x) \circ I(y)$ otherwise

where $\circ$ is relational composition, and $\uparrow$ is restriction of a relation to a set.

Finally,

$I(\pi) = \cup \{I(x) \mid x \in L(\pi)\}$.

so that $I(\pi)$ is a binary relation on D, the I-O relation of π.

Pratt [12], among others, regards this relation as the definition of a program. He can then build up compound programs using relational operators; for example, relational composition for sequencing, and relexive transitive closure for indefinite iteration. Intuitively, however, there is no fixed combination of standard relational operators (union, composition, projection, etc.) which corresponds to shuffling. Our task in the next section is to prove this.

Undefinability of shuffle

Let D be a set and $R_1,\ldots,R_m$ given relations on D. The arity of these can be arbitrary. A relation R is definable from $R_1,\ldots R_m$ if it can be written as a relational expression over $R_1,\ldots,R_m$ using the following operations:

union

complement

projection

expansion

permutation

where the last three operators are defined as follows:

Let $R \subseteq D^n$; then

$\text{Proj}_i\ R(d_1,\ldots d_{i-1},d_{i+1},\ldots,d_n) \Longleftrightarrow$

$\exists (d_i)\ R(d_1,\ldots,d_i,\ldots,d_n)$

$\text{Exp}_i\ R(d_1,\ldots,d_i,\ldots d_n) \Longleftrightarrow$

$R(d_1,\ldots,d_{i-1},d_{i+1},\ldots,d_n)$

$\text{Perm}_\sigma\ R(d_1,\ldots,d_n) \Longleftrightarrow R(d_{\sigma(1)},\ldots,d_{\sigma(n)})$

where σ is a permutation of $\{1,\ldots,n\}$.

Theorem 1. There exist two programs π_1 and π_2 and an interpretation I such that $I(\pi_1 \Delta \pi_2)$ is not definable from $I(\pi_1)$, $I(\pi_2)$ and the identity relation.

Proof. Let Σ = a,b,c,d and consider the programs π_1 = a;b and π_2 = c;d. The interpretation I is defined over $D = \{0,1\}$. We set

$I(a) = \{(0,0), (1,0)\}$

$I(b) = \{(1,1)\}$

$I(c) = \{(0,1),(1,1)\}$

$I(d) = \{(0,1)\}$

Clearly $I(a;b) = I(c;d) = \emptyset$. and it is easy to check that I(a;b c;d) is the constant function $K = \{(0,1), (1,1)\}$. Our problem is to show that K is not definable from $\emptyset$ and id, the identity relation on $\{0,1\}$. We adopt an approach from logic. Let $R_1,\ldots,R_m$ be given relations on D, and let $\Gamma(R_1,\ldots,R_m)$ be the set of bijections of D leaving these relations invariant; i.e., $f:D \to D$ is in Γ iff for each R_i

$R_i(d_1,\ldots,d_n) \Leftrightarrow R_i(f(d_1),\ldots,f(d_n))$.

Bancilhon [1] shows that a relation R is definable from $R_1,\ldots,R_m$ if and only if each f in $\Gamma(R_1,\ldots,R_m)$ leaves R invariant. (In fact, we need only the easy direction of his result.) There are only two bijections of $\{0,1\}$ and both leave $\emptyset$ and id invariant. But the function f interchanging 0 and 1 does not leave K invariant, and so K is not definable. This proves our theorem.

3. LOWER BOUND RESULTS

We turn to the analysis of expressions involving shuffle. Theorem 2 shows that the class of event expressions [14] denotes the full clss of r.e. sets. Event expressions, introduced in [13] and there called message transfer expressions, were intended as a behavioral characterization of the system description language PPML. That language was able to define Petri-net types of behavior. Theorem 2, however, shows that event expresions are considerably more general.

Theorem 3 asserts the existence of deterministic context-free languages L_1 and L_2 such that $L_1 \Delta L_2$ is NP-complete. Thus, unless P=NP, there is no polynomial time algorithm for "decoding" possible shuffled strings from general CFL's. This means that one should be careful when extending path-expression definitions [2] to cases where component processes exhibit recursive or stack-like behaviors.

Event expressions

Event expressions denote formal languages in the manner of regular expressions, except that an extra denotational process excludes some strings

from the language. The idea is to use special synchronization symbols to control concurrent combinations. Execution sequences not having a specified pattern of synchronization symbols are excluded from the language.

Let Δ and S be two finite disjoint alphabets (terminal and synchronization alphabets respectively). We assume $S = A \quad \bar{A}$ where A is a finite set and $\bar{A} = \{\bar{a} \mid a \in A\}$, the set of inverses of symbols in A. Then the class EXP(Σ,S) of event expressions is given by:

$\Sigma \cup S \subseteq EXP(\Sigma,S)$;

$\{\emptyset,\lambda\} \subseteq EXP(\Sigma,S)$;

if $e_1, e_2 \in EXP(\Sigma,S)$, then

$e_1 \cup e_2$, $e_1 \cdot e_2$, e_1^*, $e_1 \Delta e_2$, $e_1^\dagger$

are all members of EXP(Σ,S).

Language definition from an EE is a two-phase process; the first is similar to that for regular expressions; the second is the intersection process mentioned above. Thus we define an intermediate language L_0:

$L_0(\sigma) = \{\sigma\}$ for $\sigma \in \Sigma \cup S \cup \{\lambda\}$;

$L_0(\emptyset) = \emptyset$

$L_0(e_1 \cup e_2) = L_0(e_1) \cup L_0(e_2)$;

$L_0(e_1 \cdot e_2) = L_0(e_1) \cdot L_0(e_2)$;

$L_0(e_1^*) = L_0(e_1)^*$;

$L_0(e_1 \Delta e_2) = L_0(e_1) \Delta L_0(e_2)$;

$L_0(e_1^\dagger) = L_0(e_1)^\dagger$.

For the second phase, define the "cancellation grammar" G with productions

$$\{a\,\bar{a} \to \lambda,\ \bar{a}\,a \to \lambda \mid a \in A\}$$

Then

$$L(e) = \{w \in \Sigma^* \mid (y \in L_0(e))(y \overset{*}{\underset{G}{\Rightarrow}} w)\}$$

Example - Let $\Sigma = \{a,b\}$, $A = \{\alpha,\beta\}$. Then the language $a^n b^n$ is L(e) where e is the expression

$$((a\alpha)^* (b\beta)^*) \Delta (\bar{\alpha}\bar{\beta})^\dagger$$

In this example, the dagger operation introduces $\bar{\alpha}$ and $\bar{\beta}$ symbols in equal numbers; the language $\bar{\alpha}^n\bar{\beta}^n$ is a subset of $(\bar{\alpha}\bar{\beta})^\dagger$. Shuffling $(\bar{\alpha}\bar{\beta})^\dagger$ into $(a\alpha)^* (b\beta)^*$ and cancelling in effect eliminates all strings except those of the form $a^n b^n$. Similarly, the language

$$a^m b^n \# b^n a^m$$

is generated by the expression

$$((a\alpha)^*(b\beta)^*\#(b\gamma)^*(a\delta)^*) \Delta (\bar{\alpha}\bar{\delta})^\dagger \Delta (\bar{\beta}\bar{\gamma})^\dagger.$$

Notice that the β and γ symbols control the matching of b's, while the α and δ symbols control the matching of a's.

An easy fact about cancellation is given by

Lemma 1. Let $\sigma_i \in \Sigma$ and $w_i \in S^*$. If the string $w_1\sigma_1 w_1\sigma_2 \ldots\ldots \sigma_n w_{n+1}$ cancels (to $\sigma_1\sigma_2 \ldots\ldots \sigma_n$) then each w_i cancels to λ.

Proof. Observe that the cancellation grammar cannot remove symbols in Σ; thus cancellation sequences must be independently applied to each w_i.

Theorem 2. For any r.e. subset L of Σ^*, we can find an event expression e such that L=L(e).

Proof. We show that an EE can be constructed in such a way that the cancellation activity simulates the computation of a two-counter automaton. We equip the automaton with output instructions and let the terminal symbols of the EE correspond to the output symbols of the 2CM; the only trace of the computation is then the successive output symbols of the machine. Minsky's theorem states that for any r.e. set L, there is a 2 CM M such that the output language of M is just L; this gives us the result.

The main difficulty with the proof is ensuring that every cancellation sequence which occurs represents a computation of the machine. This is one of the reasons for Lemma 1: output symbols do not cancel, so the cancellations can be localized to the w_i. By itself, however, Lemma 1 will not suffice because each w_i will be composed of a string over S* essentially of the form $y_1,\ldots,y_p$ where each y_j is an instantaneous configuration of the 2CM followed by the "inverse" of the next configuration. Cancellation of these y's is supposed to represent computation of the 2CM between output steps. In order to control cancellation within each string $y_1,\ldots y_p$ we introduce auxiliary cancellation symbols "[" and "]", inserting them into the expression in such a way that if each [cancels a], then the whole string $y_1,\ldots,y_p$ will collapse to λ and correctly represent a computation in doing so.

The two counters of our machine contain a string of a's and a string of b's respectively; thus a configuration has the form

$$a^m q_i b^n$$

where q_i is a state of the machine. We make up an expression for each q_i so that (if the a,b,q symbols are regarded as terminal for the moment) the associated language would be

$$\{\bar{a}^m \bar{q}_i \bar{b}^n \cdot [\cdot b^{n'} q_j a^{n'} \mid m,n \geq 0\}$$

where m' and n' are the new contents of the a and b counters of the machine, and q_j is the correct new state.

This expression can easily be constructed using the idea in the example preceding Lemma 1. We then union together expressions for each q_i and star the result, thus obtaining our expression for the whole language.

We now give the construction process in some detail. First we describe two-counter machines, then give the associated EE's, and finally show language equality.

Two-counter machines

The 2CM is a finite set M of instructions having one of the following forms:

q_i: if $\begin{matrix}\text{counter-A}\\ \text{counter-B}\end{matrix}$ = 0 go to q_j else go to q_k

q_i: write (δ); go to q_j

q_i: $\begin{matrix}\text{counter-A}\leftarrow\text{counter-A}\\ \text{counter-B}\leftarrow\text{counter-B}\end{matrix}$ ± 1; go to q_j

q_i: HALT

One instruction, say q_o, is designated as the start instruction; the machine may start with counter-A = p, counter-B = r for any natural numbers p and r.

A configuration of a 2CM is just a triple (k,ℓ,q_i) where k and ℓ are natural numbers and q_i is a state (output is ignored here). In the usual way we define a computation sequence of m to be a sequence of configurations each correctly following from its predecessor according to machine instructions, starting with (p,r,q_o) for some p and r, and ending with some (k,ℓ,q) where q is a HALT instruction. The output string $w(\xi)$ is the string of symbols obtained by concatenating left to right all output symbols produced during the sequence, and then

$L(M)=\{w(\xi) \mid \xi$ is a computation sequence of M.$\}$

For technical reasons, we will modify M's finite control as follows: the state set is partitioned into two disjoint subsets Q(odd) and Q(even); and transitions always go from odd to even or from even to odd states. Any 2CM can be modified this way without changing the language generated.

Construction of an EE from a 2CM

We let Σ be the output alphabet of M; then Σ is also the terminal alphabet of e(M). We now give the auxiliary alphabet S. The "positive part" A of S has several components:

$G(\text{even}) = \{\alpha_e,\beta_e,\gamma_e,\sigma_e\}$

$G(\text{odd}) = \{\alpha_e,\beta_e,\gamma_e,\sigma_e\}$

$T = \{a,b\}$

$Q = \{q_1,\ldots,q_n\}$ (the state set of M)

$P = \{[\}$

Then we have

$A = G(\text{even}) \cup G(\text{odd}) \cup T \cup Q \cup P$

and $S = A \cup \bar{A}$ where $\bar{A}$ consists of all inverses of symbols in A. We write the inverse of "[" as "]".

Corresponding to each state of M we give an EE. Our understanding is that any Greek letter auxiliary symbols in the EE are taken from G(even) or G(odd) depending on whether the state is in Q(even) or Q(odd). We will thus drop subscripts on Greek auxiliaries.

Case 1.

q_i: $C_A \leftarrow C_A + 1$; go to q_j

e_i: $(\bar{a}\alpha)^*\cdot\bar{q}_i(\bar{b}\beta)^*\cdot[\cdot(b\gamma)^*\cdot q_j\cdot a\cdot(a\delta)^* \Delta (\beta\gamma)^\dagger \Delta (\alpha\delta)^\dagger$

To understand this expression, momentarily regard all symbols except the Greek letters as terminal symbols. The dagger introduces $\bar{\beta}$ and $\bar{\gamma}$ in equal numbers, and $\bar{\alpha}$ and $\bar{\delta}$ in equal numbers. These barred symbols migrate to the left and cancel their inverses, thus imposing a correspondence between a's and $\bar{a}$'s, and between b's and $\bar{b}$'s. After cancelling Greek symbols we are left with strings of the form

$$\bar{a}^m\bar{q}_i\bar{b}^n\cdot[\cdot b^n q_j a^{m+1}$$

which represent successive configurations of the 2CM

Now similarly we have an expression for each state of the 2CM.

Case 2.

q_i: $C_A \leftarrow C_A - 1$; go to q_j

e_i: $(\bar{a}\alpha)^*\bar{a}\bar{q}_i(\bar{b}\beta)^*\cdot[\cdot(b\gamma)^*q_j(a\delta)^* \Delta (\bar{\beta}\bar{\gamma})^\dagger\Delta(\bar{\alpha}\bar{\delta})^\dagger$

Case 3.

q_i: WRITE(σ); go to q_j

e_i: $(\bar{a}\alpha)^*\bar{q}_i(\bar{b}\beta)^*\cdot]^*\cdot\sigma\cdot(b\gamma)^*q_j(a\delta)^* \Delta (\bar{\beta}\bar{\gamma})^\dagger\Delta(\bar{\alpha}\bar{\delta})^\dagger$

Case 4.

q_i: if $C_A = \Lambda$ go to q_j else q_k

e_i: $((q_i(b\beta)^*\cdot[\cdot(b\gamma)^*q_j) \Delta (\bar{\beta}\bar{\gamma})$

$\cup (\bar{a}\alpha)^*\bar{a}q_i(\bar{b}\beta)^*\cdot[\cdot(b\gamma)^*q_k(a\delta^*)\cdot a)$

$\Delta (\bar{\beta}\bar{\delta})^\dagger\Delta(\bar{\alpha}\bar{\delta})^\dagger$

Similar expressions are constructed for the instructions involving counter-B.

Define

$$\text{START} = b^*q_0a^*$$

$$\text{HALT} = \bar{a}^*q_h\bar{b}^* \cdot]^* \text{ , where h is the HALT state.}$$

For the program M define

$$e(M) = \text{START} \cdot (\cup_{i\neq h} e_i)^* \quad \text{HALT}$$

We would like to show L(e(M) = L(M). The difficult part of this result is showing that $L(e(M)) \subseteq L(M)$. In the expression e(M) there are many ways to shuffle strings and therefore many possibilities for cancellation. We must show that every cancellation to terminals corresponds to a correct computation of the machine. The brackets control the order of cancellations, as do the restrictions on M and the fact that no cancellations can "cross" terminal symbols.

The function of the brackets is given precisely by the next lemma.

Lemma 2. Given a string of the form

$$w = \sigma z_1 \cdot [\cdot z_2 \cdot [z_3 \cdot \ldots \cdot [\cdot z_p]^{p-1}\tau$$

where σ and τ are in Σ, $z_i \in (A \cup \bar{A})^*$ all bracket-free; if w cancels (to στ) then each z_i cancels to λ.

The proof is by induction on p, and is omitted.

Assertion. L(e(M)) = L(M).

We will show only that $L(e(M)) \subseteq L(M)$, as explained above. Consider a string in $L_o(e(\pi)$ which cancels to terminals. From the form of e(M) this string will have an internal structure

$$\ldots x_o x_1 \ldots x_p \ldots$$

where x_o and x_p are strings introduced by expressions e_o and e_p representing two consecutive "write" statements, and $x_1,\ldots,x_{p-1}$ come from expressions representing non-writing statements. Thus x_o has the form

$$x_{o\ell}]^j \sigma x_{or};$$

each x_i has the form

$$x_{i\ell} \cdot [\cdot x_{ir}$$

and x_p has the form

$$x_{p\ell} \cdot]^k \cdot \tau \cdot x_{pr}$$

where $\sigma,\tau \in \Sigma$, and

$$x_{i\ell} \in (\bar{a}\alpha)^*\bar{Q}(\bar{b}\beta)^* \Delta \bar{\alpha},\bar{\beta}\ ^*$$

$$x_{ir} \in (b\gamma)^*Q(a\delta)^* \Delta \bar{\gamma},\bar{\delta}\ ^* .$$

We can write the string

$$\sigma x_o\ x_{1\ell} \cdot [\cdot x_{1r}x_{2\ell} \cdots [x_{(p-1)r}x_{p\ell}]^k\tau$$

in the form

$$\sigma z_1[z_2 \cdots\ [z_p]^k\tau .$$

By Lemma 1, this string must cancel to στ and so by Lemma 2 each z_i must cancel. But, $z_i = x_{ir}x_{(i+1)\ell}$ which thus must cancel for each i, and so computations can be simulated for more than one step.

As an example, suppose that x_i is generated by an expression representing instruction

$$q_i:\ C_A \leftarrow C_A + 1;\ \text{go to } q_j$$

and x_{i+1} is generated by an expression representing

$$q_k:\ C_A \leftarrow C_A - 1;\ \text{go to } q_\ell .$$

Then (ignoring the $\bar{\alpha},\bar{\beta},\bar{\gamma},\bar{\delta}$ symbols)

$$x_i = (\bar{a}\alpha)^m \bar{q}_i (\bar{b}\beta)^n\ [(b\gamma)^s \cdot q_j \cdot (a\delta)^t$$

$$x_{i+1} = (\bar{a}\alpha)^{m'}\bar{q}_k(\bar{b}\beta)^{n'} \cdot [\cdot (b\ )^{s'} \cdot q_\ell (a\delta)^{t'} .$$

The string

$$z_i = (b\gamma)^s q_j (a\delta)^t \cdot (\bar{a}\alpha)^{m'}\bar{q}_k(\bar{b}\beta)^{n'}$$

must cancel. Hence $q_j = q_k$, $s = n'$, and $t = m'$. Also, there must be exactly s $\bar{\gamma}$ symbols in z_i t $\bar{\delta}$ symbols, s $\bar{\alpha}$ symbols, and t $\bar{\beta}$ symbols. $\bar{\gamma}$ and $\bar{\delta}$ symbols in z_i must have been inserted in the expression for q_i, and any $\bar{\alpha}$ and $\bar{\beta}$ symbols in z_i by the expression for q_j since q_i and q_j have different "parity". The $\bar{\alpha}$ and $\bar{\beta}$ bymbols introduced for q_i must thus cancel the α and β symbols in the left part of x_i and the $\bar{\gamma}$ and $\bar{\delta}$ symbols introduced for q_j must cancel the γ and δ symbols in the right half of x_{i+1}. Thus m + 1 = t, n = s′, m′ = t′ + 1, n′ = s′, and the computation is advanced two steps. By induction, the strings $x_1,\ldots,x_p$ thus represent a correct computation of the machine between two output steps, and again inductively, the whole string represents a correct computation of the machine. The example just above represents a typical inductive step of the proof.

Theorem 2 follows because the trace left upon making all cancellations is just the sequence of output symbols produced by the machine.

We remark that without the daggering operation, Theorem 2 does not hold; only regular sets can be generated [14]. The dagger thus introduces all the "counting" ability into our expressions.

The complexity of pure shuffle

Our final theorem shows that even when restricted to the class of context-free languages, shuffling is a complex operation. We find deterministic CFL's L_1 and L_2 such that $L_1 \Delta L_2$ is NP-complete. This means that a general efficient algorithm which recognizes scrambled message sequences from parallel sources with context-free behavior will probably not exist (unless P = NP). It is possible, however, to design such an algorithm in case the message sources give finite-state behavior. Thus an interleaved sequence of messages coming from sources whose behavior satisfies a regular path expression will be recognizable efficiently, because regular sets are closed under shuffling. The intended use of path expressions does not even raise this issue; each process simply checks whether the subsequence of the shuffled message stream consisting only of the letters mentioned in its path expression in fact is in the language denoted by the expression. Our result implies only that in extending path-expression analysis to systems with non-regular behavior, some care should be taken with semantics of the expressions.

A language $L \subseteq \Sigma^*$ is in NP iff there is a nondeterministic TM accepting L within p(n) time, where p is a polynomial. L is NP-complete iff $L \in NP$ and for each $L' \in NP$, $L' \leq L$; i.e. there is a polynomial-time computable function ϑ: $\Sigma^* \to \Sigma^*$ such that for all x, $x \in L'$ iff $\vartheta(x) \in L$.

Theorem 3. There exist deterministic context-free languages L_1 and L_2 such that $L_1 \Delta L_2$ is NP-complete.

Proof. We show that for all $L \in NP$, there exist L_1 and L_2 (depending on L) such that $L \subseteq L_1 \Delta L_2$. By choosing L to be a known NP-complete language, we get a specific $L_1 \Delta L_2$ which is NP-complete. (Note: trivially $L_1 \Delta L_2 \in NP$.)

Our proof is another Turing machine simulation argument. Let $L \in NP$ and let M be a Turing machine accepting L within p(n) time. We may assume that M has a single tape and that the input head of M has a predictable head motion (consisting of, for example, successive sweeps across the entire tape). A configuration of M is a string in $\Sigma^*Q\Sigma^*$ where Q s the state set and Σ is the tape alphabet of M. (we will take $\Sigma = \{0,1\}$ for convenience.) In the usual way, if x and x' are configurations of M, we write $x \Rightarrow x'$ to indicate that x' follows from x by means of M's program. Define the following languages:

$$K_1 = \{x\$(x')^R\$ \mid x \Longrightarrow x'\}$$

where $(z)^R$ is the reversal of z, and \$ is a new symbol.

$$L_1 = g_1(K_1^*)$$

where g_1 is a string homomorphism that replaces each 0 by [0 and each 1 by [1, leaving any other symbols unchanged.

$$K_2 = \{\phi z \phi z^R \mid z \in (\Sigma \cup Q)^*\}$$

$$L_2 = g_2(K_2^*)$$

where g_2 is a string homomorphism replacing 0 by 0] and 1 by 1]. L_1 and L_2 are deterministic CFL's; we claim that $L_1 \Delta L_2$ is the required language.

To show that this is the case, we introduce some temporary notation. Let $Q = \{q_o,\ldots,q_n\}$ be the state set of M. For a string $x_i \in \Sigma^*Q\Sigma^*$, define the complement $C(x_i)$ to be the homorphic image of x_i given by

$C(0) = 1$;

$C(1) = 0$;

$C(q_i) = q_o q_1 \cdots q_{i-1} q_{i+1} \cdots q_n$.

Let

$$w = x_1\$(x_1')^R\$x_2\$(x_2')^R\$\ldots x_R x_k'^R\$$$

be a string in K_1, and let

$$v = \phi z_1 \phi z_1^R \phi z_2 \phi z_2^R \ldots \phi z_k \phi z_k^R$$

be a string in K_2. We say that v is the shifted complement of w just in case

$$z_1 = C(x_1')^R,\ z_1^R = C(x_2),\ldots,z_k = C(x_k')^R.$$

Notice that if $w \in K_1$ has a shifted complement in K_2 then w represents a complete computational history of m starting from the configuration x_1, because $x_2 = x_1'$, $x_3 = x_2'$, etc. Our idea is to shuffle together $g_1(w)$ with $g_2(v)$ to produce a very uniform-looking string and to use this form as the definition of the reducing function ϕ which shows that $L \leq L_1 \Delta L_2$. The brackets and markers are used to show that if $\phi(x) \in L_1 \Delta L_2$, then the only way to unshuffle $\phi(x)$ will be to have $\phi(x) \in g_1(w) \Delta g_2(v)$ where w and v are shifted complements, and w represents an accepting computation of x.

It is useful to regard all configurations as having the same length, so if $x \in \Sigma^*$, $|x| = n$, let I(x) be x followed by p(n) - n zeros. All computa-

tions last for p(n) steps, and an accepting configuration consists of tape entirely zero and head at the end of the tape, in state q_f. Let h(t) be the position of the head at step t of the computation. By our assumption h can be calculated very easily.

Set

$$(x) = g_1(q_o\ I(x)) \cdot \prod_{t=1}^{2\cdot p(n)} y(t) \cdot \$\phi g_2(1^{p(n)} q_f)$$

where $y(t) = \$\phi[01]^{h(t)} \cdot Q(t) \cdot [01]^{p(n)-h(t)}$

$$Q(t) = \begin{cases} q_o \ldots q_n & t \text{ even} \\ q_n \ldots q_o & t \text{ odd.} \end{cases}$$

Thus ϕ is an encoding of the initial configuration, followed by a large number of strings of the form

$\$\phi[01][01]\ldots[01]q_o\ldots q_n[01][01]\ldots[01]$

all of which have the same length, and then the final string which is the complement of the final configuration.

Observe that if ϕ is a shuffle of strings from L_1 and L_2, then each portion of ϕ between any $\$\phi\ldots\$\phi$ markers must consist of a contribution from L_1 shuffled with a complementary contribution from L_2, because left brackets occur in L_1 (right brackets occur in L_2) before (after) each 0 or 1. The position of the \$ and ϕ markers in ϕ guarantee that the first L_2 configuration matches the second L_1 and so forth. We thus have the fact that $\phi(x) \in q_1(w) \Delta g_2(v)$, and v is the shifted complement of w. But by our remarks above, and the fact that w starts with an initial configuration and ends with a final one, this means that w represents an accepting computation on x. Hence $x \in L \Longleftrightarrow \phi(x) \in L_1 \Delta L_2$, proving theorem 3.

4. CONCLUSION

The results above represent a small venture into possible methods for expressing concurreht behavior. By appropriately restricting allowable synchronization patterns, it seems that many types of synchronized concurrent behavior are definable. Various forms of Petri-net languages can be described by introducing symbols representing the operations of adding 1 and subtracting 1 from a given place, and insisting that in any string the number of "add 1" symbols up to any point exceeds the number of "subtract 1" symbols.

We have not dealt with the class of expressions formed by adding shuffle and dagger to ordinary regular expressions. Adding just shuffle does not increase denotational power, although the generalized membership problem: "given e and x, is $x \in L(e)$?" becomes NP-hard. (This can be proved by a reduction from the exact cover problem.) Finally, we have no results on the possible complexity of the languages formed from regular languages by closing under the dagger operation.

REFERENCES

[1] Bancilhon, F., "Data Structures: Specification and Realization", Ph.D. Thesis, University of Michigan, 1976.

[2] Campbell, R. A., and Habermann, N., "The Specifications of Process Synchronization by Path Expressions", Lecture Notes in Computer Science Vol. 16, Springer-Verlag, 1974.

[3] Codd, E. F., "A Relational Model for Large Shared Data Banks", CACM 13, #6 pp. 377-387, 1970.

[4] Ginsburg, S., The Mathematical Theory of Context-Free Languages, McGraw Hill, New York, 1966.

[5] Ginsburg, S., and Spanier, E. H., "Mappings of Languages by Two-Tape Devices", JACM, 12 pp. 423-434, 1965.

[6] Greibach, S., "The Hardest Context-Free Language", SIAM J. Compt. 2, pp 304-310, 1973.

[7] Horning, J. J., and Randell, B., "Process Structuring", Computing Surveys, 5 #1 pp. 5-30, 1975.

[8] Lipton, R. J., "Reduction: A Method of Proving Properties of Parallel Programs", CACM, 18, #12 pp. 717-721, 1975.

[9] Lipton, R. J., "The Reachability Problem Requires Exponential Space", Tech. Report, Department of Computer Science, Yale University, 1975, to appear in Theoretical Computer Science.

[10] Owicki, S. "A Consistent & Complete Deductive System for the Verification of Parallel Programs", Proceedings of 8th Annual ACM Symposium on Theory of Computing, May, 1976), pp. 73-86.

[11] Peterson, J., "Petri Nets", Computing Surveys 9, #3, pp 223-252, 1977.

[12] Pratt, V. R., "Semantical Consideration on Floyd-Hoare Logic", Proceedings of the 17th Symposium on Foundations of Computer Science, (October, 1976), pp. 109-121.

[13] Riddle, W. E., "Modelling and Analysis of Supervisory Systems", Ph.D. Thesis, Stanford University, 1972.

[14] Riddle, W. E., "Software System Modelling and Analysis", RSSM/25, Tech. Report, Department of Computer and Communication Sciences, University of Michigan, July 1976.

[15] Shoenfield, J. R., Mathematical Logic, Addison-Wesley, Reading, Mass., 1967.