RESUMPTION MODELS :

A DENOTATIONAL SEMANTICS FOR CONCURRENCY AND SYNCHRONIZATION

by

Robert L. Treece
B.S., University of Kansas, 1983

Submitted to the Department of
Computer Science and the Faculty
of the Graduate School of the
University of Kansas in partial
fulfillment of the requirements for
the degree of Master of Science.

Redacted Signature

Professor in Charge

Redacted Signature

Redacted Signature

Committee Members

Redacted Signature

For the Department

Date thesis Accepted

ABSTRACT

This thesis describes a technique for the study of the semantics of concurrent programming languages. A denotational semantics is developed using resumption models, an adaptation of Milner's processes. In such a model, the meaning of a language construct is a resumption, a function from initial state to a set of all possible paths of execution. Thus the semantics, though denotational, has an operational flavor as computation history is included in the meaning of a program.

A general language allowing parallel execution of processes is given using shared memory and critical region mutual exclustion. The semantics of this language is provided as a framework for studying the semantics of other multiprogramming languages. Hoare's Communicating Sequential Processes is discussed as an example of a distributed language utilizing interprocess communication. The semantics of CSP could be adapted to other distributed languages.

ACKNOWLEDGEMENTS

I must certainly express my gratitude to my adviser, Professor Adrian Tang. He has provided much information and invaluable inspiration.

Thanks also to Drs. Appie van de Liefvoort and Tsutomu Kamimura for their guidance and for serving on my thesis committee. And to Dr. William Bulgren for his support during my graduate work.

TABLE OF CONTENTS

I. Introduction

Introduction

This thesis describes a technique for the study of the denotational semantics of concurrent programming languages.

In a denotational semantics, programming language constructs are modelled by implementation independent entities. The meaning of a construct is given by a function which maps syntactic constructs into semantic domains. This is in contrast to an operational semantics in which the behavior is studied via an abstract machine.

A program is a group of statements and therefore a program's meaning is derived from the statements comprising it. Each constituent statement is viewed as a function which transforms the machine state. The meaning of the program is then the composition of the constituent functions, providing again a function from state to state.

Many constructs have recursively defined meanings, the meaning of the whole relying on the semantic definitions of its constituents. For instance, the assignment A:=A+6 is described in terms of the denotations for the expression A+6. The value of A+6 depends on the current state of the machine; in particular, on the interpretation of the value in the memory location refered to as A.

When describing sequential programs using denotational methods, it is sufficient to consider the statements of a program, and the entire program itself, as indivisible. The execution of a sequential program is continuous; the state of the program will not be altered by another process. Therefore, programs (A:=0; A:=A+6) and (A:=6) will have the

same meaning. Even if nondeterminancy is allowed by a restricted pseudo-concurrency in which no program interrupts another, a program's meaning can be given in terms of the sequential behaviors of its parts. The meaning of ((A:=0; A:=A+6)//(A:=6)) in such a case would be the union of the meanings of the two possibilities ((A:=0; A:=A+6); (A:=6)) and ((A:=6); (A:=0; A:=A+6)). In each case, (A=6) is true after execution.

When actual concurrency or pseudo-concurrency with interruptions is allowed, a command which began in an initial state may not complete execution before being interrupted. It may be suspended in an intermediate state with some computation to finish when resumed. While suspended, another process may alter the machine state. The command may also complete execution without interruption, terminating in some final state. The meaning of a statement must contain all such possibilities. Making such considerations, the meaning of ((A:=0; A:=A+6)//(A:=6)) now yields as postcondition (A=6 V A=12).

The meaning of a nondeterministic program can be viewed as a tree structure. Each node represents a control point or interruption point. Branches from a node are the possible next atomic actions. The leaves are the possible final states. Of course, the actual execution of such a program will follow only one path from root to leaf though the chosen path may differ for different executions.

The typical denotational model must be altered to include the notion of interuption. The meaning of a command is no longer a mapping from initial machine state to final machine state, but to a set of possible final states. By dividing a program's processes into in-

divisible parts, the possible statement interleavings may be seen. Each interleaving may produce a different final state.

To distinguish easily between the various possible executions, we will consider labelled statements. Each atomic action will be given a name. Then the execution of a program, given by the order of execution of its atomic components, can be seen as a string of labels. Therefore, at any point in a computation, the execution path which brought the program to that point can be seen as well as the possible executions to come.

To accomodate the many possible executions of a given concurrent program, we will use resumption models which are variations of Milner's Processes [Mil73]. The meaning of the segment of a program remaining to be executed is called the resumption of the program. Of course initially the meaning of the entire program is itself a resumption. Because a resumption includes all interuption points, the various possible paths of execution are contained in the meaning of a program. This gives the semantics an operational flavor.

Other approaches to the semantics of concurrency can be found in [Sou84] and [DeB82]. In the former, the paths of execution are not included in the meaning of a program. The set of final results is the main concern. Therefore, where the current state determines the meaning of a construct in our semantics, the state is instead not consulted. In this respect, our semantics is perhaps more operational. The semantics of CSP found in [DeB82] is more closely based on Milner's Processes than the resumptions used here. The domains used are more complex than ours.

Thesis Outline

The notion of a resumption is ellaborated in section II and provides a mathematical basis for the rest of the paper. Notations are given for resumption combinators necessary to the later discussions. Some domain theory is also discussed.

A general Parallel Programming Language is given in section III and the semantics using resumptions is derived. Some simple examples are given, illustrating the use of the semantic definitions. In presenting the semantics for a general parallel language, we provide the necessary framework for using resumptions to describe the semantics of a group of languages.

In section IV, Hoare's Communicating Sequential Processes (CSP) [Hoa78] is introduced, then given a semantics applicable to other distributed languages utilizing interprocess communication. Meanings are also derived for example CSP programs.

Section V concludes the paper by briefly suggesting implementation possibilities.

II. Resumptions

In this thesis, we will refer to the meaning of a program as a resumption. We will also call the meaning of a program segment a resumption and describe ways to combine the resumptions of program parts to obtain the meaning of the entire program.

As discussed earlier, a sequential program can be viewed as a function which transforms the machine state. A statement takes an initial state and produces a final state. Thus the meaning of a statement is a function f:S-->S where S is the domain of machine states.

Since there are no other processes manipulating the state of a sequential program, the meaning of a group of statements is the composition of the meaning functions of the statements comprising the group. The final state of one statement becomes the initial state for the following statement. Therefore, the meaning of a program is also a function g:S-->S.

Things are not so simple once concurrency is introduced. In parallel programs, nondeterminancy exists since there are often several actions which could be completed next at a given point in a program. In this case, the initial state of a statement in one process may not be the same as the final state produced by the previous statement. Another process may have altered the state in between.

Since a parallel program is, like a sequential program, a group of statements, we would like to be able to describe the semantics of the whole in terms of that of its parts. A parallel program is made up of indivisible, atomic actions seperated by possible interruption points. Some statements are themselves atomic, and their meanings may be viewed

in parallel programs in the same way they are in sequential programs: as functions from state to state. For instance, the skip statement could be atomic and is then viewed as the identity function in sequential and concurrent semantics. Other types of statements have possible interruptions. This type of statement is itself a sequence of atomic actions. An example of a statement of this type is the if-then-else-fi statement in which an interruption point follows evaluation of the boolean condition. Then the meaning is a pair consisting of the intermediate interruption state and the resumption of the unexecuted remainder of the command. As you can see, resumption is used to refer to the meaning of a portion of a single statement as well as a portion of a program.

Because a given concurrent program may have several possible executions, we must consider not a single result, but a set of possible results. Thus we will model concurrent behaviors by the domain R (for resumptions) satisfying the recursive domain equation :

$$R = S_{\perp} \dashrightarrow_{\perp} P[S_{\perp} + (S_{\perp} \otimes R_{\perp})] \qquad (*)$$

in which the meaning of a program, a resumption, is a function which maps initial states to sets. The sets contain elements which are either final states or state-resumption pairs. This recursive domain equation is guaranteed a solution by Plotkin's SFP objects [Plo76].

Domain Operators

We will not discuss domains in detail here. An introduction can be found in [Gor79] with a much more extensive treatment in [Sto81]. The key notion is that a domain is a group of objects with an ordering. These objects are used to give meaning to syntactic constructs. And

each domain contains a least element $\perp$ refered to as "bottom" or "undefined."

The operators used in the above domain equation: $\rightarrow_{\perp}$, P, +, and $\otimes$ represent, respectively, strict functions, powerdomain, coalesced sum, and strict product. Stoy's text [Sto81] also provides rigorous definitions of these constructions excluding P which is treated in [Plo76].

The notion of a function space D-->E is common. When using domains as models for programming language semantics, a restricted space [D-->E] of only those functions which preserve the structure of D are considered. These are the continuous functions.

The strict function space $[D \rightarrow_{\perp} E]$ is a further restriction of [D-->E] and contains only strict functions. The strict functions are those functions in [D-->E] which map $\perp_D$ to $\perp_E$. The motivation for this restriction involves functions with call-by-value arguments. If the argument e of a function f cannot be evaluated, or the evaluation does not terminate, then f(e) cannot be evaluated.

The notion of the powerdomain of a domain is analogous to that of a powerset of a set. Each element of the powerdomain is a set of elements from the domain. For our purposes, a member of the powerdomain represents the set of all possible executions of a particular statement.

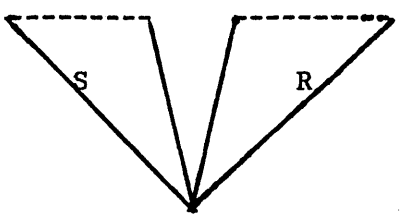
Members of the coalesced sum of two domains A + B are members either of A or of B. When an element c is in both A and B, it is possible to distinguish between c in A and c in B, except $\perp_A$ and $\perp_B$ are now the same object.

The cartesian product of two domains A X B is the set of pairs (a,b) such that a is in A and b is in B. The strict product $A \otimes B$

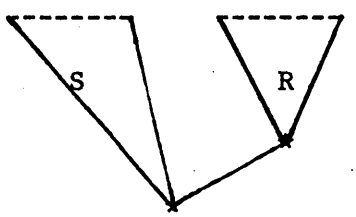
eliminates all pairs of the form $(a,\perp_B)$ or $(\perp_A,b)$ except $(\perp_A,\perp_B)$. The reasoning again is derived from call-by-value functions. If an n-ary function f has arguments (e1,e2,...,eN) and one of the eI's cannot be evaluated, then neither can f(e1,e2,...,eN) and the result would be the same as if each eI were undefined.

The domain $D_\perp$ is refered to as the lift of D. $D_\perp$ contains all the elements of D and includes a new least element. The function up:D-->$D_\perp$ maps elements of D to their corresponding element in the lift of D and is essentially an identity mapping. For any function f:D-->E, lift(f): $D_\perp$-->E is the function such that for each d in D, lift(f)(up(d))=f(d).

The lift of a domain is used to establish an accurate model of computation. In general, a lifted domain is used when combining domains using $\otimes$ or +. By adding a new least element to each domain, we avoid amalgamation of the bottom elements when combining domains (see diagram below.) In the domain S $\otimes$ R, you may not have a pair of the form $(s,\perp_R)$ unless $s=\perp_S$. S here is a flat domain with a least element; R also has a least element. By lifting R, in the product S $\otimes$ $R_\perp$ you may now have pairs $(s,\perp_R)$ in which s is not $\perp_S$. Such pairs would be used when a program may make some finite progress but has an undefined resumption. The intermediate state information should still be given.



(S $\otimes$ R)



(S $\otimes$ $R_\perp$)

Resumption Combinators

In order to describe the semantics of the following sections certain combinators must be defined for use with resumptions. In these definitions assume s is some element in $S_\perp$ and r1 and r2 are in R.

For composition of resumptions we define $*:R^2 \rightarrow R$ such that:

r1*r2(s) = cases r1(s)

if s'.{s' ⊗ up(r2)}

if (s',r1').{s' ⊗ up(r1'*r2)}

The cases notation is read, if r1(s) is a single state s', then the result is {s' ⊗ up(r2)}, otherwise if r1(s) is an interruption state-resumption pair (s',r1'), then the result is {s' ⊗ up(r1'*r2)}. This composition is very intuitive. If the meaning of the first statement, r1, is a single state, then r1*r2 is simply a pair consisting of the final state of r1(s) and the resumption r2. The up(r2) must be used because r2 is in R and the earlier domain equation (*) specifies a pair to include a member of $R_\perp$. If r1(s) is a pair, then the composition is defined recursively.

The parallelism operator $||:R^2 \rightarrow R$ is defined:

r1||r2(s) = cases r1(s)

if s'.{s' ⊗ up(r2)}

if (s',r1').{s' ⊗ up(r1'||r2)}

U cases r2(s)

if s'.{s' ⊗ up(r1)}

if (s',r2').{s' ⊗ up(r1||r2')}

Note that this operator is for parallelism in which r1||r2 terminates when both r1 and r2 have terminated. Here we have the union of pos-

sibilities. The first atomic part of r1 or of r2 could be done first.

Nondeterminancy between resumptions is accomodated by the operator ?:R^2-->R such that:

```
r1?r2(s) = cases r1(s)
             if s'.{s'}
             if (s',r1').{s' ⊗ up(r1')}
           U cases r2(s)
             if s'.{s'}
             if (s',r2').{s' ⊗ up(r2')}
```

Only one of the pair of resumptions is executed and the meaning of such a combination includes all possibilities from execution of either resumption but includes no interleavings.

When using these combinators to derive the semantics of a program, several of each combinator may appear in a given expression, therefore the precedence of evaluation is defined: *, ?, ||. Of course parenthesis may alter the order of evaluation.

Different resumption models will be used in the following semantic definitions, though the underlying notions remain consistent.

III. Parallel Programming Language and Semantics

An excellent overview of concurrent programming language constructs can be found in [And83].

Parallel Programming Language

To begin with, we will describe a simple Parallel Programming Language (PPL) allowing concurrency based on a common store. Mutual exclusion in the language is provided by a critical region construct.

The programming language consists of the following statements: skip, assignment, alternation, iteration, compound, parallel execution and critical region. We will be using a labelled langauge in which atomic actions are given names. The atomic constructs in PPL are boolean expressions, skip, assignment and critical region. Thus a partial grammar would be :

C ::= L:skip | L:I := E | if L:B then C else C fi |
while L:B do C od | C;C |
cobegin C{//C}* coend | L:< C' >

where the descriptions for I (identifiers), E (expressions), L (labels) and B (boolean expressions) are as commonly described and it is assumed that all variables are globally defined. A restricted PPL with grammar start symbol C' is described shortly.

The interesting construct in this language is the parallel execution of processes. When processes are executing concurrently and sharing memory, the possibility for one process to interfere with or change the execution of one of its coprocesses exists. For instance, consider the following program segment containing a pair of coprocesses:

cobegin x := y + y // y := x + x coend

and assume that x = 1 and y = 3 is true as a precondition of the statement. Also assume for the time being, that no other processes exist in the same environment. If no interruptions are allowed in the execution of either parallel statement, i.e. each assignment is atomic, then the resulting value for (x,y) will be (6,12) or (4,2), depending on which of the two statements is executed first. If interruption is allowed between evaluation of the expressions and assignment, then the possible values will be (6,12), (4,2) and (6,2). More interruption points could exist, depending upon the implementation of the language. If interruption could occur within evaluation of the expression, then possible values would be (6,12), (4,2), (6,2), (5,2) and (6,7). If there were other processes running concurrently with this one and sharing the same memory, then the set of possible values for (x,y) would be infinite as the effects of those processes on x and y are unknown.

When processes run concurrently they must often compete for shared resources. As was illustrated above, a memory location could be a shared resource; others could be hardware devices or library routines.

In general, access to one of these shared resources by more than one process at one time can provide undesired results. Consider two processes using a line printer simultaneously in which the output from the two processes is interleaved. Use of such a resource must be restricted so that only one process may use it at a time; the resource must be dedicated to the process until the process has finished using it.

The code in which a process uses a shared resource is called a critical region. Thus, of a pair of processes having critical regions referencing the same shared resource, only one may be executing its critical region at one time. Execution of critical regions should be mutually exclusive in time. This mutual exclusion can be supplied by the critical region construct which serves to make a sequence of otherwise interruptable statements atomic.

Sequential Programming Language and Semantics

The meaning of a statement within a critical region is the same as it would be in a non-concurrent program. So, we will first define the sequential semantics of those statements of PPL which may appear within a critical region. A grammar for this Sequential Programming Language (SPL) is:

C' ::= L:skip | L:V := E | if L:B then C' else C' fi |
while B do C' od | C';C'

The semantics is given by the mapping $\mathcal{S}$:SPL-->S-->S where SPL is the domain of all valid SPL programs and S the domain of machine states. Here, the domain of states consists of the partial functions from identifiers to values: S = [I-->V]. V will contain integers and truth values and a bottom element. Though the commands are labelled, the labels will be ignored for the time being.

The sementic mappings are defined:

$\mathcal{S}$[L: skip] = λs.s

The meaning of skip is the identity function.

$$\mathcal{S}[\text{L:v:=e}] = \lambda s.s[\mathcal{E}(e)(s)/v]$$

An assignment replaces the current value of the variable on the left hand side with the evaluation of the expression on the right hand side. The function $\mathcal{E}$:E-->[S-->V] evaluates expressions. The state s[x/y] is the state derived from s by placing x in location y.

$$\mathcal{S}[\ \underline{\text{if}}\ \text{L:b}\ \underline{\text{then}}\ \text{C1}\ \underline{\text{else}}\ \text{C2}] = \lambda s.\text{if}\ \mathcal{B}(b)(s)$$
$$\text{then}\ \mathcal{S}[\text{C1}](s)$$
$$\text{else}\ \mathcal{S}[\text{C2}](s)$$

The meaning of an if statement depends on the evaluation of the boolean condition in the current state. $\mathcal{B}$:B-->[S-->T] is a semantic function used to evaluate boolean expressions. T is the domain of truth values.

$$\mathcal{S}[\ \underline{\text{while}}\ \text{L:b}\ \underline{\text{do}}\ \text{C}\ \underline{\text{od}}\ ] = \lambda s.\text{if}\ \mathcal{B}(b)(s)$$
$$\text{then}\ \mathcal{S}[\text{C};\ \underline{\text{while}}\ \text{L:b}\ \underline{\text{do}}\ \text{C}\ \underline{\text{od}}\ ](s)$$
$$\text{else s}$$

The meaning of a while loop also depends on the boolean expression.

$$\mathcal{S}[\text{C1;C2}] = \lambda s.\mathcal{S}[\text{C2}](\mathcal{S}[\text{C1}](s))$$

Statement sequencing is modelled by functional composition. The meaning of C2 is applied to the state resulting from applying C1 to the initial state.

We may now define the semantics of the PPL.

Semantics of PPL

PPL is the domain of all valid programs in the PPL. We will define a semantic mapping $\mathcal{M}$:PPL-->R which maps a program to its meaning. The domain of resumptions which we will use will be similar to that seen in equation (*). Because we are using labelled atomic commands, we would

like to include these labels in the program's resumption. Thus we will use a modification of (*):

$$R = S_{\perp} \rightarrow_{\perp} P[(L_{\perp} \otimes S_{\perp}) + (L_{\perp} \otimes S_{\perp} \otimes R_{\perp})] \qquad (**)$$

Each element in the set of possible results of statement execution now includes a label. This label identifies the atomic action just performed. The notion of a state here is the same as that of the SPL: S = [I-->V]. In this shared memory language, all processes use a global state. By lifting elements of R to $R_{\perp}$ on the right-hand-side of the domain equation, it is possible to represent more information. A member of the powerdomain which is a triple can have an undefined resumption while still containing label and state information. Without lifting, an undefined resumption would cause the label and state to be ignored.

The seven statements in the PPL, being skip, assignment, alternation, iteration, compound, parallelism and critical region imply that our semantics will contain seven mappings, one for each type of statement. In the definitions of these functions, we will use the mappings seen earlier, namely $\mathcal{E}$:E-->[S-->V] and $\mathcal{B}$:E-->[S-->T] where E is the domain of expressions, V the domain of values and T the domain of truth values. $\mathcal{E}$ is an evaluation mapping providing the value associated with an expression. $\mathcal{B}$ evaluates boolean expressions. Each of the domains contains an undefined element so that undefined expressions will have denotations and non-boolean expressions will still have meaning in T. These two mappings are necessary for the formalism and are rather self explanitory though they are explained well in [Gor81].

The skip statement does nothing and does not change the machine state. Therefore:

$$\mathcal{M}[\text{L: } \underline{\text{skip}} \text{ }] = \lambda_{\perp} s.\{L \otimes s\}$$

The meaning of skip is a function which takes a machine state and produces the same state. It doesn't change the state though it does introduce another interruption point. The label l identifies the statement.

The assignment L:v := e evaluates e in the current state and places the value in the location associated with v.

$$\mathcal{M}[\text{L:v:=e}] = \lambda_{\perp} s.\{L \otimes s[\mathcal{E}(e)(s)/v]\}$$

Notice that this assignment is indivisible; there are no possible interruptions between the initial state and the state in which v is updated. Also note that the expression e must be evaluated in the current state using $\mathcal{E}$ before assignment is made. Assignment is considered atomic and thus has a label which is included in the meanging.

In alternation, a choice is made between two statements based on a boolean condition.

$$\mathcal{M}[\text{ } \underline{\text{if}} \text{ L:b } \underline{\text{then}} \text{ C1 } \underline{\text{else}} \text{ C2 } \underline{\text{fi}} \text{ }] = \lambda_{\perp} s.\text{if } \mathcal{B}(b)(s)$$
$$\text{then } \{L \otimes s \otimes \text{up}(\mathcal{M}[C1])\}$$
$$\text{else } \{L \otimes s \otimes \text{up}\mathcal{M}[C2])\}$$

No interruption is allowed during evaluation of the boolean condition. The first interruption point of the statement is prior to execution of the chosen statement. And though an interruption may occur there, the choice between C1 and C2 has been made based on the condition. The evaluation of b does not change the machine state. Because evalation of the condition is atomic, its label is used to mark the result. The resumption in each triple here must be lifted from R to $R_{\perp}$ using the function up. This satisfies domain equation (**).

The syntax presented for PPL does not allow an if statement without an else clause, though this may be simulated by having skip in place of C2. The semantics of the if b then C else skip fi statement would differ from that of if b then C fi by having an extra atomic action (skip) and interruption point when the condition is false.

Iteration is execution of a statement as long as a condition is true.

$$\mathcal{M}[\ \underline{\text{while}}\ \text{L:b}\ \underline{\text{do}}\ \text{C}\ \underline{\text{od}}\ ] = \lambda_{\perp} s.\ \text{if}\ \mathcal{B}(b)(s)$$
$$\text{then}\ \{L \otimes s \otimes \text{up}(\mathcal{M}[C;\ \underline{\text{while}}\ b\ \underline{\text{do}}\ C\ \underline{\text{od}}\ ])\}$$
$$\text{else}\ \{L \otimes s\}$$

As in the alternate statement, no interruption is allowed when evaluating the boolean expression. Note that though the condition is evaluated to false, there is still a possible interruption point before execution of the statement is complete and control passes to the next statement or the program terminates. The function up is again used to lift the resumption into $R_{\perp}$.

The mappings for compound statements or parallel statements are easily defined due to the combinators which were given in section II:

$$\mathcal{M}[C1;C2] = \mathcal{M}[C1] * \mathcal{M}[C2]$$

$$\mathcal{M}[\ \underline{\text{cobegin}}\ C1//C2\ \underline{\text{coend}}\ ] = \mathcal{M}[C1] \,|\!|\, \mathcal{M}[C2]$$

Both mappings rely on those combinators for their descriptions.

The critical region construct makes a group of statements atomic by eliminating all interruption points from the enclosed statements. This transforms the meaning of a given statement to its meaning as used in a sequential language.

$$\mathcal{M}[\text{L:< C >}] = \lambda_{\perp}s.\{\text{L} \otimes \mathcal{S}[\text{C}](s)\}$$

Since this is an atomic action, its result is a label-state pair, the state given by the sequential meaning of the enclosed statements using the semantics of SPL.

The use of this semantics will be shown in the following examples:

PPL Example One

Program1 :: cobegin L1: A := 6
// L2: A := 0; L3: A := A + 6
coend

$$\mathcal{M}[\text{Program1}] = \mathcal{M}[\text{L1:A:=6}] \,||\, \mathcal{M}[\text{L2:A:=0;L3:A:=A+6}]$$

Using the rule for compound statements:

$$= \mathcal{M}[\text{L1:A:=6}] \,||\, \mathcal{M}[\text{L2:A:=0}] * \mathcal{M}[\text{L3:A:=A+6}]$$

Expanding the last two statements:

$$= \mathcal{M}[\text{L1:A:=6}] \,||\, (\lambda_{\perp}s.\{(\text{L2}, s[\mathcal{E}(0)(s)/\text{A}])\}) * (\lambda_{\perp}s.\{(\text{L3}, s[\mathcal{E}(\text{A+6})(s)/\text{A}])\})$$

Evaluating * before || as defined by the precedence of combinators:

$$= \mathcal{M}[\text{L1:A:=6}] \,||\, (\lambda_{\perp}s.\{(\text{L2}, s[\mathcal{E}(0)(s)/\text{A}], (\lambda_{\perp}s.\{(\text{L3}, s[\mathcal{E}(\text{A+6})(s)/\text{A}])\}))\})$$

The result using * was obtained by seeing that $\mathcal{M}$[L2:A:=0] applied to a state yielded a label-state pair rather than a label-state-resumption triple. Continuing:

$$= \lambda_{\perp}s.\{(\text{L1}, s[\mathcal{E}(6)(s)/\text{A}])\} \,||\, (\lambda_{\perp}s.\{(\text{L2}, s[\mathcal{E}(0)(s)/\text{A}], (\lambda_{\perp}s.\{(\text{L3}, s[\mathcal{E}(\text{A+6})(s)/\text{A}])\}))\})$$

Now, using the || combinator:

$= \lambda_{\perp} s.\{(L1, s[\mathcal{E}(6)(s)/A],$

$(\lambda_{\perp} s.\{(L2, s[\mathcal{E}(0)(s)/A],$

$(\lambda_{\perp} s.\{(L3, s[\mathcal{E}(A+6)(s)/A])\})\})\})\}$

$\cup \{(\lambda_{\perp} s.\{(L2, s[\mathcal{E}(0)(s)/A],$

$((\lambda_{\perp} s.\{(L1, s[\mathcal{E}(6)(s)/A])\}) \,||\,$

$(\lambda_{\perp} s.\{(L3, s[\mathcal{E}(A+6)(s)/A])\})\})\}$

And finally $||$ once more yields the final result:

$= \lambda_{\perp} s.\{(L1, s[\mathcal{E}(6)(s)/A],$

$(\lambda_{\perp} s.\{L2, s[\mathcal{E}(0)(s)/A],$

$(\lambda_{\perp} s.\{(L3, s[\mathcal{E}(A+6)(s)/A])\})\})\})\}$

$\cup \{(\lambda_{\perp} s.\{(L2, s[\mathcal{E}(0)(s)/A],$

$(\lambda_{\perp} s.\{(L1, s[\mathcal{E}(6)(s)/A], \lambda_{\perp} s.\{(L3, s[\mathcal{E}(A+6)(s)/A])\})\}$

$\cup \{(L3, s[\mathcal{E}(A+6)(s)/A], \lambda_{\perp} s.\{(L1, s[\mathcal{E}(6)(s)/A])\})\})\})\}$

A tree structure exhibiting the program's possible behaviors can be found in the Appendix.

PPL Example Two

```
Program2 ::  cobegin L1: x := 0;
                     L2: < if B1:y=0 then L3: x := 1
                                     else L4:  skip
                          fi>
                  // L5: y :=0;
                          if B2:x=0 then L6: y := 1
                                    else L7:  skip
                          fi
             coend
```

For this PPL program, I will derive the set of possible first branches. The remainder of the derivation will be left to the reader.

$\mathcal{M}$[Program2] = $\mathcal{M}$[L1: x:=0;

L2: < if B1:y=0 then L3: x := 1

else L4: skip

fi >]

||$\mathcal{M}$[L5: y := 0;

if B2:x=0 then L6: y := 1

else L7: skip

fi]

= $\mathcal{M}$[L1: x :=0]

*$\mathcal{M}$[L2:< if B1:y=0 then L3: x := 1

else L4: skip

fi >]

||$\mathcal{M}$[L5: y := 0]

*$\mathcal{M}$[if B2:x=0 then L6: y := 1

else L7: skip

fi]

= ($\lambda_{\perp}$s.{(L1,s[$\mathcal{E}$(0)(s)/x])})

*$\mathcal{M}$[L2]

||($\lambda_{\perp}$s.{(L5,s[$\mathcal{E}$(0)(s)/y])})

* ($\lambda_{\perp}$s.if$\mathcal{B}$(x=0)(s)then {(B2,s,($\lambda_{\perp}$s.{(L6,s[$\mathcal{E}$(1)(s)/y])}))}

else {(B2,s,($\lambda_{\perp}$s.{(L7,s)}))})

$$= (\lambda_{\perp} s.\{(L1, s[\mathcal{E}(0)(s)/x], \mathcal{M}[L2])\})$$

$$\| (\lambda_{\perp} s.\{(L5, s[\mathcal{E}(0)(s)/y], (\mathcal{M}[\text{if}...])\})\}$$

$$= \lambda_{\perp} s.\{(L1, s[\mathcal{E}(0)(s)/x], \mathcal{M}[L2] \| \lambda_{\perp} s.\{(L5, s[\mathcal{E}(0)(s)/y], \mathcal{M}[\text{if}...])\}\}$$

$$\cup \{(L5, s[\mathcal{E}(0)(s)/y], (\mathcal{M}[\text{if}...] \| (\lambda_{\perp} s.\{(L1, s[\mathcal{E}(0)(s)/x], \mathcal{M}[L2])\}))\}$$

This gives the first two branches of the execution tree of Program2. The possible actions from the initial state are L1 and L5 with interruption states $s[\mathcal{E}(0)(s)/x]$ and $s[\mathcal{E}(0)(s)/y]$ respectively. The subtrees are then given by the resumption in each initial triple. The complete derivation, expressed as a tree, is given in the Appendix.

IV. Communicating Sequential Processes and Semantics

It is certainly true that processes using a common data item communicate. One process may make changes to a shared memory location which another process detects. The changes made to a shared object reflect the state of the process making the changes. Unfortunately, this description of process state may not be valid by the time another process inspects the shared object and receives the communication. The communication is asynchronous as the message is not sent and received simultaneously.

A semaphore [Dij68] can facilitate synchronous communication when a process, waiting for a resource after performing P, is granted the resource by a fellow process executing the corresponding V. The semaphore could transmit data as well as perform synchronization and thus provide synchronous communication.

Synchronous communication can also be accomplished by message passing in which the sending process specifies the destination process for output and the receiver specifies the source process of input. This provides synchronization as well as communication because the data transfer only occurs when a matched pair of input and output statements are reached at the same time. Obviously, a message may not be received until it has been sent, but also the sending process may not proceed until its message has been received.

Hoare's Communicating Sequential Processes (CSP) [Hoa78] provides our framework for the study of synchronous communication.

Communicating Sequential Processes

A Communicating Sequential Processes program consists of a group of parallel processes. The execution of each process begins when the execution of the program starts; execution of the program is completed when every process has terminated. Each of the processes has its own store assuming that no nesting of processes is allowed.

Because no memory is shared, interprocess communication occurs only through input and output commands of the form :

```
[ {IN} ProcessA :: ...  ProcessB!message ...
//
{IN} ProcessB :: ... ProcessA?target ... ]
```

in which each such command staticly specifies the process with which it intends to communicate. The ! primitive used is similar to a send(ProcessB,message) and ? to receive(ProcessA,target). The communication is carried out when a process's input statement matches the output statement of another process. A pair of input and output statements match when the source process of the input statement contains an output command naming the first process as destination, and the message type matches the target type. An I/O command is said to fail when the named process has terminated. Obviously, a message may not be received until it is sent. Also, a sending process must wait for its output command to be matched before proceeding. Therefore message passing occurs synchronously.

When a process desires to perform communication, some information must be available about the state of its potential communication partner [Kie79]. When a process is running but is performing non-I/O state-

ments, it is said to be active. A process may also be ready when waiting for a matching I/O command. Whether a given I/O statement is performed or not depends on its partner's readiness and also with whom the partner intends to communicate and with what type of message. A process may also be in a terminated state which would cause failure of an I/O statement naming the terminated process as partner. We will return to the topic of process I/O states when discussing the semantics of the I/O statements.

Other language constructs in CSP are repetition and alternation based on Dijkstra's guarded command [Dij75]. An alternate command is formed by a list of guarded commands. The guards are evaluated and a selection is made from those guards which are ready. A guard may contain boolean expressions and variable declarations. I/O commands may also appear in guards and are ready when a match occurs. Allowing output commands in guards is a reasonable extension of Hoare's original CSP [Buc83]. Message exchange in an I/O guard occurs only when that guard is ready and selected. A guard is ready when the boolean condition is true and the I/O command, if present, is ready. A guard fails when the condition is false or the communication partner has terminated. Otherwise the guard is neither ready nor failed, but will not be selected if another guard is ready. In this case, a guard will not be chosen unless it becomes ready before other such guards. Repetition is also performed on a group of guarded commands in which a choice is made of ready guards repeatedly until all guards fail at which time the looping terminates.

CSP also includes skip and assignment statements and parallel execution similar to cobegin.

A simple CSP program to compute a given term in the fibonacci sequence would consist of two concurrent process: a User process and process Fib which calculates the term. Process Fib is as follows :

```
Fib :: {declarations}
       current , lastone ,
       onebeforethat , term,
       desiredterm : integer;
       {initializations}
       I1:current := 0;
       I2:lastone := 1;
       I3:onebeforethat := -1;
       I4:term := 0;
    {main loop}
    *[C1:User?desiredterm -->
         {calculate term}
         *[B2:term <= desiredterm -->
             A1:current := lastone + onebeforethat;
             A2:onebeforethat := lastone;
             A3:lastone := current;
             A4:term := term + 1
          ]
    #B1:term > desiredterm -->
         {report result and reinitialize}
         C2:User!current;
         A5:current := 0;
         A6:lastone := 1;
         A7:onebeforethat := -1;
         A8:term := 0
    ]
    E1:end
```

The repetition command forms the bulk of the process. On the first execution of the main loop, the second guard fails and the command will wait for input from User. Once received, the inner loop is performed, calculating the appropriate term. When this terminates, the choice again is at the outside loop. The second guard would be chosen here and the result returned to User. Fib would then wait for more input, terminating only when User had also terminated.

Another interesting example using this language is a semaphore in which synchronization is performed via a shared process rather than

shared data. The semaphore, accessed by an array User of processes is done :

```
Sem :: val : integer;
       {init} I1:val := NumResources;
       *[(i:1..UserLimit)C1:User(i)?V()--> A1:val:=val+1
       #(i:1..UserLimit)B1:val>0;C2:User(i)?P()--> A2:val:=val-1
       ]
       E1:end
```

Note that P() and V() are value-less structured messages and the process Sem stops when all user processes have terminated. A process requesting access by performing Sem!P() is automatically suspended because its execution will not proceed until the output command is matched.

Semantics of CSP

We will define a mapping $\mathcal{R}$:CSP-->R which provides the resumption of a CSP construct. CSP here is a domain of all CSP programs. R will be the domain of resumptions seen in equation (**) of section III:

$$R = S_{\perp} \rightarrow_{\perp} P[(L_{\perp} \otimes S_{\perp}) + (L_{\perp} \otimes S_{\perp} \otimes R_{\perp})] \qquad (**)$$

The notion of a state S will be different here though:

S = M X G X TE X TI where

M = [I-->V] X ... X [I-->V],

G = [P-->N],

TE = [P-->N],

TI = [P X N -->{true,false} X N].

P is the domain of process names, I the domain of identifiers, V the domain of values and T the domain of truth values. I, V and T are as seen previously. The state consists of four parts. M is the domain of memories and is the only component of S which may be altered directly by

a process. M can be viewed as a vector whose projections are memories [I-->V] for individual processes. We assume a finite number of processes. G, TE and TI facilitate maintenance of global information.

G provides an integer for each process. This integer represents the I/O commands for which a process is waiting. This may be a list of commands, one or no commands. To obtain an integer from an I/O command, we will use an encoding function K: CSP + {NIL}-->N. The necessary information when encoding is the process name, type of communication desired and the parameter type. The encoding of NIL represents an empty list. Mapping W: P X N-->[P-->N]-->T is used to inspect part G of the state. Given a process name, an I/O command coded as an integer and a global state sG, W indicates whether or not the given process is waiting for the given I/O command in the given state. To add or delete commands from the list, we will use A: N X N-->N and M: P X N-->N. A takes an encoded list and the code for a new command to be added to the list and produces the new encoding. M takes a process name and an encoded list and deletes from the list all I/O commands which refer to the given process, providing a new encoded list.

TE contains information about which processes have terminated. This information is necessary when evaluating I/O commands for failure. Mapping D: P-->[P-->T]-->T inspects the TE portion of a state and indicates whether or not a given process has terminated. We assume that each computation begins in a state in which each process's entry in sTE is false.

TI contains information about the number of times a particular I/O command in a process has been tried. The use of this domain will be

seen in the semantics of I/O commands. Function SUCC: N-->N will be used to increment an entry in this portion of the state.

For simplicity, we will assume that process names are of the form P# where # is an integer. This will facilitate ease in notation but doesn't restrict the semantics. In the semantic mappings which follow, it will often be necessary to refer to only a portion of a state s. sN will be the own memory of process Pn, sG will be the G component of s, sTE the TE portion and sTI the TI part. Other notations used: s[x/Ny] will place the value x in location y of the memory of process Pn; s[x/Gn] will place x in the location for process Pn in sG; s[x/TEn] places x in Pn's location in sTE; s[x/TIn(y)] places x in the yth position of Pn's location in TI.

As in PPL, we will be using labels for atomic actions. This is not usually a part of CSP though the language is easily adapted. The atomic actions are boolean evaluation, assignment, skip, input, output and termination. Termination is indicated by an end statement. This construct also is not normally a part of CSP.

In the following definitions we will assume the given statements are found in process Pn and will therefore often refer to the memory sN.

The mappings for skip and assignment are the same as those found in the semantics of PPL:

$$\mathcal{K}[\text{L: } \underline{\text{skip}} \text{ }] = \lambda_{\perp}s.\{(l,s)\}$$

$$\mathcal{K}[\text{L: v:=e}] = \lambda_{\perp}s.\{(L,s[\mathcal{E}(e)(sN)/Nv])\}$$

Notice that we are using the evaluation mapping $\mathcal{E}$ as seen in the previous section. The meaning of $\mathcal{E}$ is the same here. We will also be using an identical $\mathcal{B}$ mapping for evaluating boolean expressions.

We will assume that the end statement is the last construct in each process:

$$\mathcal{R}[\text{L: } \underline{\text{end}}\ ] = \lambda_{\perp}s.\{(L,s[\mathcal{B}(\text{true})(sN)/TEn][M(Pn,sG(P1))/G1]\ldots [M(Pn,sG(Pz))/Gz])\}$$

The termination of a process must be reflected in the global state sTE so that I/O commands referencing the terminated process may fail. A process may already be waiting to communicate with the terminated process and therefore each such command must be removed from the other process's waiting lists. This is done using M. We assume here that there are Z process in the program.

For an input command found in process Pn:

$$\begin{aligned}\mathcal{R}[\text{L: Pj?v}] = \lambda_{\perp}s.&\text{if } D(Pj)(sTE)\\ &\text{then } \{(L,s[\mathcal{B}(\text{true})(sN)/TEn])\}\\ &\text{else if } W(Pj,K(Pn!v))(sG) \text{ and}\\ &\qquad \text{not } W(Pn,K(Pj?v))(sG)\\ &\qquad \text{then } \{(L,s[\mathcal{E}(e)(sJ)/Nv][K(Pj?v)/Gn]\\ &\qquad\qquad [\mathcal{E}(0)(sN)/TIn(K(Pj?v))])\}\\ &\qquad \text{else if } W(Pj,K(Pn!v))(sG)\\ &\qquad\qquad \text{then } \{(L,s[K(NIL)/Gn][K(NIL)/Gj])\}\\ &\qquad\qquad \text{else } \{(L,s[K(Pj?v)/Gn]\\ &\qquad\qquad\qquad [SUCC(sTI(Pn,K(Pj?v)))/TIn(K(Pj?v))],\\ &\qquad\qquad\qquad up(\mathcal{R}[\text{L: Pj?v}]))\}\end{aligned}$$

There are four possible results here. The first is given when the communication partner has terminated. If the named process has not terminated, then the other results may be reached. The second result occurs when the potential partner is already waiting for the current I/O

command but the current command has not yet been tried; the third when the partner is ready with respect to the current command and this command has been tried once already. Finally, the fourth choice occurs when the statement is neither ready nor fails.

When a process reaches an I/O statement which names a terminated process, the current process also terminates as it can make no further progress. In this case, sTE is inspected using the mapping D which yields true when the named process has terminated.

W is used to report whether the potential partner is waiting to communicate with this I/O command. In the second result, the communication is performed, altering the own memory of the receiving process. Pn's entry in sG is altered to indicate its readiness to communicate with Pj. sTI is also used here, reinitialized to zero once the communication has been performed. sTI is used to count the number of times an I/O command is tried. This will provide information about potential deadlocks. The limit of incrementing an element of sTI using SUCC would indicate that a deadlock has occurred.

When the partner process is ready for communication with this command and this command has been tried at least once before, the transfer has already taken place, performed when the other process became ready. In states where this is the case, the waiting lists for Pn and Pj in sG are reinitialized to empty using K(NIL).

In the final case, when the guard is not ready and does not fail, it must be tried again later. The corresponding entry in sTI is incremented to indicate the current try, and Pn's waiting list is set to indicate its readiness to communicate.

Output is similar to input:

$\mathcal{R}$[L: Pj!e] = λ_1s.if D(Pj)(sTE)

then {(L,s[$\mathcal{B}$(true)(sN)/TEn])}

else if W(Pj,K(Pn?e))(sG) and

not W(Pn,K(Pj!e))(sG)

then {(L,s[$\mathcal{E}$(e)(sN)/Jv][K(Pj!e)/Gn]

[$\mathcal{E}$(0)(sN)/TIn(K(Pj!e))])}

else if W(Pj,K(Pn?e))(sG)

then {(L,s[K(NIL)/Gn][K(NIL)/Gj])}

else {(L,s[K(Pj!e)/Gn]

[SUCC(sTI(Pn,K(Pj!e)))/Tin(K(Pj!e))],

up$\mathcal{R}$[L:Pj!e]))}

The remaining two maps, for alternate and repetition commands will use this notation:

Let Ga = [Ba:ba;La:Pia.ka-->Ca] where a is an integer and let:

G1..GZ = [G1#G2#...#GZ].

In the I/O command listed in Ga, . could be ! or ?, while .' is the compliment. Therefore:

$\mathcal{R}$[G1..GZ] = λ_1s.if ($\mathcal{B}$(not b1)(sN) or D(Pi1)(sTE)) and...and

($\mathcal{B}$(not bZ)(sN) or D(PiZ)(sTE)))

then {($,s[$\mathcal{B}$(true)(sN)/TEn])}

else U for j:1..Z of

if$\mathcal{B}$(bj)(sN) and W(Pij,K(Pn.'kj))(sG)

then {(Bj,s[K(NIL)/Gij][K(Pij.kj)/Gn],

up([Lj:Pij.kj; Cj]))}

else if $\mathcal{B}$(bj)(sN) and not D(Pij)(sTE)

then {(Bj,s[A(sG(Pn),K(Pij.kj))/Gn]

[SUCC(sTI(Pn,K(Pij.kj)))

/TIn(K(Pij.kj))])}

else ϕ

First of all, if the alternate command fails because all of its guards fail, then the process terminates. A reserved label $ is used to mark the evaluation of the failed guards. Otherwise the command has a set of possible results, one each for each guarded command comprising the alternate command. For each guard which is ready, i.e. the condition is true and the I/O command ready, the I/O command and subsequent command Ca may be performed. For each guard which is not ready but does not fail, there is a choice of trying the I/O command. This result must also include all the other possible choices which could occur here since the I/O command may fail at a later time if the named partner terminates. These other choices are given by including the meaning of the entire alternate command as the resumption in the triple. sTI must also be altered to indicate the new try of the I/O command. Finally, for those guards which fail, there is no corresponding choice.

This semantics for alternate commands will be used in the mapping for the repetition command:

$\mathcal{R}$[*[G1..GZ]] = λ_1s.if (($\mathcal{B}$(not b1)(sN) or D(Pi1)(sTE)) and...and

(($\mathcal{B}$(not bZ)(sN) or D(PiZ)(sTE)))

then {($,s)}

else $\mathcal{R}$[[G1..GZ];*[G1..GZ]](s)

In the repetition command, if all guards fail then the looping stops rather than the process terminating as in the alternate. Otherwise the statement is executed just as an alternate command followed by other possible iterations of the repetition statement. This definition is similar to that for while found in the semantics of PPL. The reserved label $ is used to mark the evaluation of the failed guards, as in the alternate.

Compound statements and parallel processes are defined as in the previous section using the combinators of section II:

$$\mathcal{R}[C1;C2] = \mathcal{R}[C1] * \mathcal{R}[C2]$$

$$\mathcal{R}[C1 // C2] = \mathcal{R}[C1] \,||\, \mathcal{R}[C2]$$

The following example shows sample derivations using the semantics:

CSP Example

In this example, we will ignore changes made to sTI.

```
CSPexample :: [ P1:: L1:P2!3; E1:end
              //P2:: L2:P1?w; E2:end]
```

$$\mathcal{R}[\text{CSPexample}] = \mathcal{R}[P1] \,||\, \mathcal{R}[P2]$$

$$= \mathcal{R}[L1] * \mathcal{R}[E1] \,||\, \mathcal{R}[L2] * \mathcal{R}[E2]$$

$$= (\lambda_\perp s.\{(L1, s[K(P2!3)/G1], \mathcal{R}[L1] * (\lambda_\perp s.\{(E1, s[\mathcal{B}(true)(s1)/TE1][M(P1,sG(P2))/G2])\})\}) \,||\, (\lambda_\perp s.\{(L2, s[K(P1?w)/G2], \mathcal{R}[L2] * (\lambda_\perp s.\{(E2, s[\mathcal{B}(true)(s2)/TE2][M(P2,sG(P1))/G1])\})\})$$

$$= \lambda_\perp s.\{(L1, s[K(P2!3)/G1], \mathcal{R}[L2]*\mathcal{R}[E2] \,||\, \mathcal{R}[L1]*\mathcal{R}[E1])\} \cup \{(L2, s[K(P1?w)/G2], \mathcal{R}[L1]*\mathcal{R}[E1] \,||\, \mathcal{R}[L2]*\mathcal{R}[E2])\}$$

$$= \ldots$$

```
=λ⊥s.{(L1,s[K(P2!3)/G1],
      λ⊥s.{(L1,s[K(P2!3)/G1],ℛ[L2]*ℛ[E2]¦¦ℛ[L1]*ℛ[E1])}
       U {(L2,s[ℰ(3)(s1)/2w][K(P1?w)/G2],
          λ⊥s.{(E2,s[ℬ(true)(s2)/TE2],
               λ⊥s.{(L1,s[K(NIL)/G1][K(NIL)/G2],
                    λ⊥s.{(E1,s[ℬ(true)(s1)/TE1])})})}
            U {(L1,s[K(NIL)/G1][K(NIL)/G2],
               λ⊥s.{(E2,s[ℬ(true)(s2)/TE2],
                    λ⊥s.{(E1,s[ℬ(true)(s2)/TE1])})}
             U {(E1,s[ℬ(true)(s1)/TE1],
                λ⊥s.{(E2,s[ℬ(true)(s2)/TE2])})})})}
  U {(L2,s[K(P1?w)/G2],
     λ⊥s.{(L2,s[K(P1?w)/G2],ℛ[L1]*ℛ[E1]¦¦ℛ[L2]*ℛ[E2])}
      U {(L1,s[ℰ(3)(s1)/2w][K(P1?w)/G2],
         λ⊥s.{(E1,s[ℬ(true)(s1)/TE1],
              λ⊥s.{(L2,s[K(NIL)/G2][K(NIL)/G1],
                   λ⊥s.{(E2,s[ℬ(true)(s2)/TE2])})})}
           U {(L2,s[K(NIL)/G2][K(NIL)/G1],
              λ⊥s.{(E1,s[ℬ(true)(s1)/TE1],
                   λ⊥s.{(E2,s[ℬ(true)(s2)/TE2])})}
            U {(E2,s[ℬ(true)(s2)/TE2],
               λ⊥s.{(E1,s[ℬ(true)(s1)/TE1])})})})}
= {etc...}
```

The notion of trying an I/O command is shown here. It would be possible for an implementation to repeatedly try an I/O command unsuccessfully, making no practical progress. The tree illustrating the execution of

this program is located in the Appendix.

V. Conclusion

In the preceeding pages, I have presented a general technique for evaluating the meaning of a concurrent program. With the two sets of definitions provided - that for the Parallel Programming Language and that for Communicating Sequential Processes - one could study the meanings of multiprogramming languages or of distributed languages.

Given the semantic functions for the program constructs and the resumption combinators discussed, deriving the meaning of a program is very algorithmic. It is readily discovered though, upon attempting to derive the meaning of even a relatively small program, that though straightforward, the method is rather tedious. Since this method is deterministic, it seems natural to consider possibilities for implementation.

The first stage of such an implementation would be a parser. This could be made part of a compiler and could therefore use the parsing generated there. In fact, it would be easier to consider generating meanings for only syntacticly correct programs.

The next part of the meaning generator would be to fill some suitable data structure representing the structure of the program. This would be similar to evaluating an expression, making use of the semantic mappings and resumption combinators. This data structure could be some kind of tree in which each node is an interruption state, having possible actions as subtrees. From this structure, the final semantic function derived for the program could be shown. A tree structure similar to those found in the Appendix could also be given.

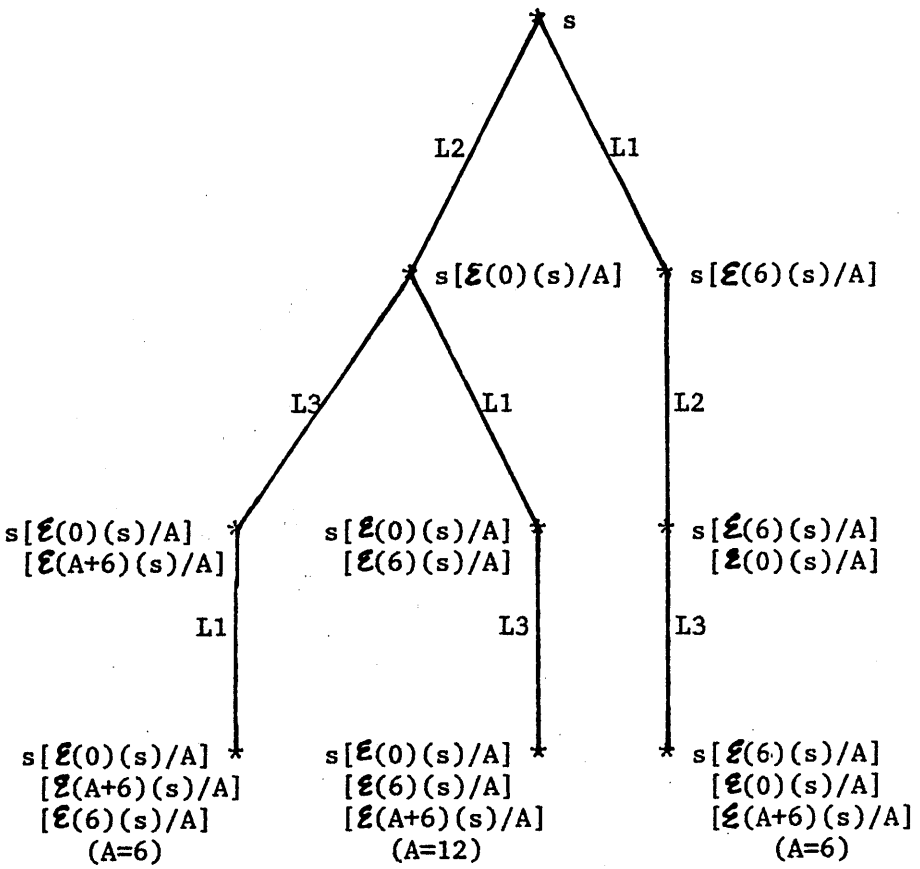
Though I haven't looked into it much, I think it would be possible to determine when cetain concurrent programs were in some senses equivalent. One could compare the set of final configurations for two programs and also check various inter-execution states. Or isolate certain pertinent variables and compare the effects of programs on these variables.

A program generating an execution description tree structure could certainly be useful as a debugging tool for semantic errors. Since errors produced by fairly complicated concurrent programs are not always easily reproduced, it would be helpful to be able to see the chain of atomic actions made in reaching a certain state. For this purpose, including such a semantic generator in a compiler seems logical.

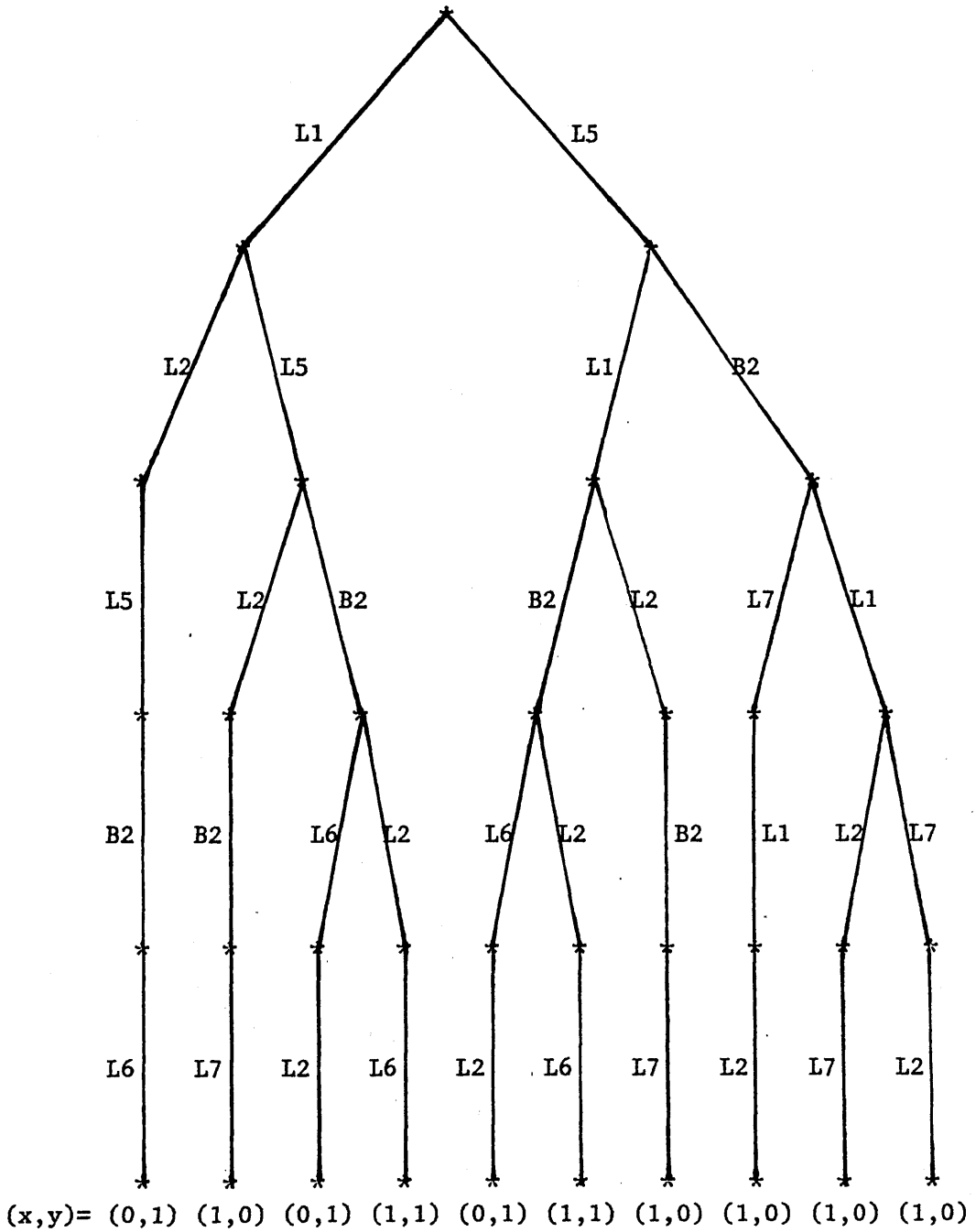
Complexity issues must certainly be considered before embarking on such an implementation.

APPENDIX

The following figures represent the possible executions of a given concurrent program. An interruption point is labelled by its intermediate state, a leaf by its corresponding final state. Branches, representing actions, are labeled by the corresponding atomic action. Values of pertinent variables are listed below each final state.

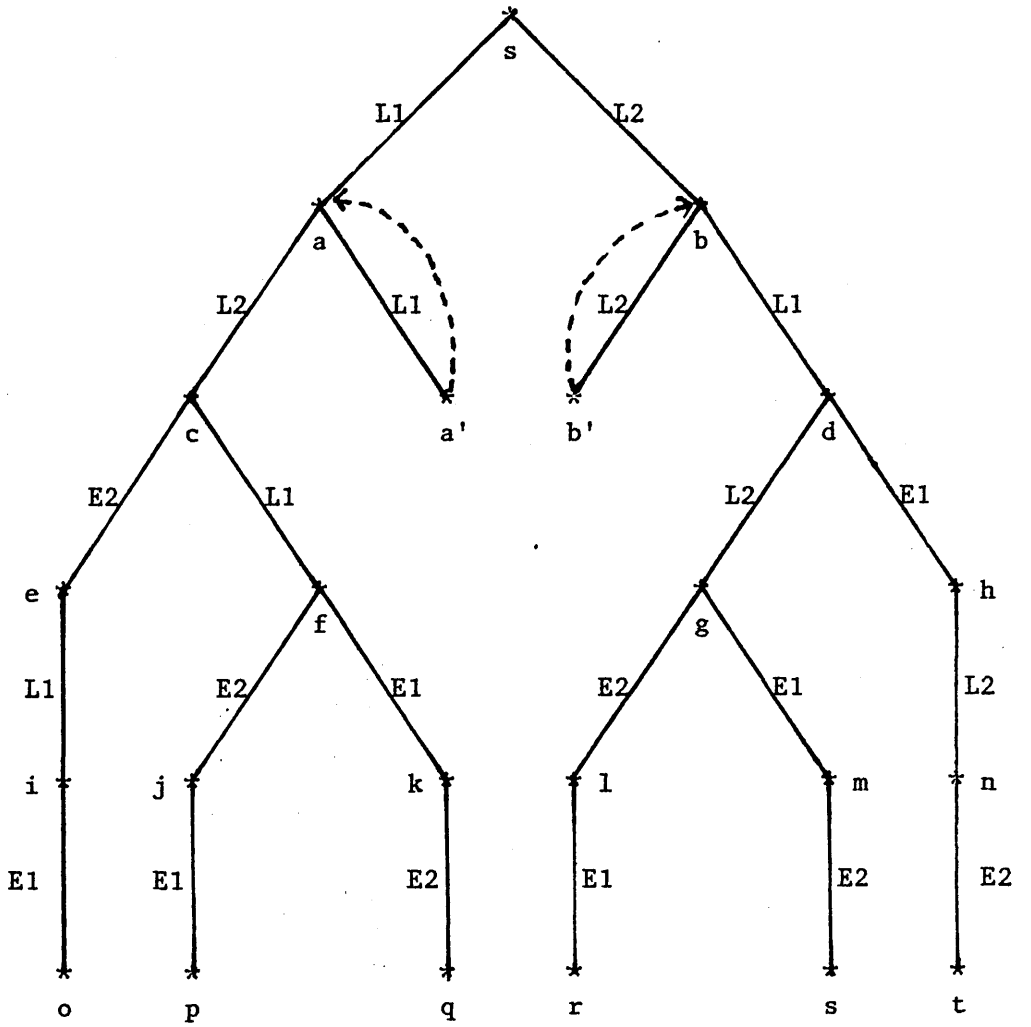


PPL EXAMPLE ONE



PPL EXAMPLE TWO

The dashed arrows used here indicate that the subtrees for paths L1;L1 and L2;L2 are the same as for L1 and L2 respectively. In each final state of this example, (w=3) in s2.



where : a = s[K(P1?w)/G2] = a' b = s[K(P2!3)/G1] = b'
c = s[K(P1?w)/G2][$\mathcal{E}$(3)(s1)/2w][K(P2!3)/G1]
d = s[K(P2!3)/G1][$\mathcal{E}$(3)(s1)/2w][K(P1?w)/G2] (note: c=d)
e = c[$\mathcal{B}$(true)(s2)/TE2] h = d[$\mathcal{B}$(true)(s1)/TE1]
f = c[K(NIL)/G1][K(NIL)/G2] g = d[K(NIL)/G2][K(NIL)/G1]
i = e[K(NIL)/G1][K(NIL)/G2] n = h[K(NIL)/G2][K(NIL)/G1]
o = s[K(P1?w)/G2][$\mathcal{E}$(3)(s1)/2w][K(P2!3)/G1]
[$\mathcal{B}$(true)(s2)/TE2][K(NIL)/G1][K(NIL)/G2][$\mathcal{B}$(true)(s1)/TE1]
p = o = q = r = s = t

CSP EXAMPLE

BIBLIOGRAPHY

[And83] Andrews, G.R. and F.B. Schneider. Concepts and Notations for Concurrent Programming. ACM Computing Surveys 15:1, pp. 3-43, 1983.

[Bro85] Brookes, S.D. On The Axiomatic Treatment Of Concurrency. To appear: Proceedings 1984 NSF-SERC Seminar On Concurrency Springer LNCS, 1985.

[Buc83] Buckley, G.N. and A. Silberschatz. An Effective Implementation for the Generalized Input-Output Construct of CSP. ACM Transactions on Programming Languages and Systems 5:2, pp. 223-235, 1983.

[DeB82] De Bakker, J.W. and J.I. Zucker. Processes and the Denotational Semantics of Concurrency. Information and Control 54, pp. 70-120, 1982.

[Dij68] Dijkstra, E.W. Cooperating Sequential Processes. In F. Genuys (Ed.) Programming Languages Academic Press, pp. 43-112, 1968.

[Dij75] Dijkstra, E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. CACM 18:8, pp. 453-457, 1975.

[Gor79] Gordon, M.J.C. The Denotational Description of Programming Languages. Springer-Verlag, 1979.

[Hoa72] Hoare, C.A.R. Towards a Theory of Parallel Programming. In C.A.R. Hoare and R.H. Perrott (Eds.) Operating Systems Techniques, pp. 61-71, Academic Press, 1972.

[Hoa78] Hoare, C.A.R. Communicating Sequential Processes. CACM 21:8, pp. 666-677, 1978.

[Kie79] Kieburtz, R.B. and A. Silberschatz. Comments on "Communicating Sequential Processes". ACM Transactions on Programming Languages and Systems 1:2, pp. 218-225, 1979.

[Mil73] Milner, R. Processes: A Mathematical Model of Computing Agents. In Rose and Shepherdson (Eds.), Proceedngs, Logic Colloquim 1973, pp. 157-173. North-Holland, 1973.

[Owi80] Owicki, S.S. Axiomatic Proof Techniques for Parallel Programs. Garland Publishing, 1980.

[Plo76] Plotkin, G.D. A Powerdomain Construction. SIAM Journal of Computing 5:3, pp. 452-487, 1976.

[Sil79] Silberschatz, A. Communication and Synchronization in Distributed Systems. IEEE Transactions on Software Engineering, 5:6, pp. 542-546, 1979.

[Sou84] Soundararajan, N. Denotational Semantics of CSP. Theoretical Computer Science 33, pp. 279-304, 1984.

[Sto81] Stoy, J.E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory MIT Press, 1981.