An Operator Calculus For Computer Programs

T. G. Lewis

Computer Science Department
Oregon State University
Corvallis, Oregon 97331

78-1-5

An Operator Calculus For Computer Programs

T. G. Lewis

Computer Science Department
Oregon State University
Corvallis, Oregon 97331

78-1-5

An Operator Calculus For Computer Programs

Part 1

T. G. Lewis
Computer Science Department
Oregon State University
Corvallis, Oregon 97331
(503) 754-3273

Report
78-1-5

Abstract

An operator calculus for transforming assignment statements, loops, and choice constructs into equation-of-state expressions is shown to be powerful enough to synthetically execute a given computer program and its test data. A demonstration of the program is equivalent to applying an operator formalism to the equation-of-state in order to reduce the equation to a simple, single-assignment sequence of assignment statements. The program is said to be demonstrated correctly with respect to the given test data if the resulting single-assignment sequence produces the desired "output" result.

The method is applied to a variety of programs of increasing complexity. A published program shown to be a correct implementation of the binary search algorithm is demonstrated to be incorrect.

Contents

0. INTRODUCTION

Program correctness is an ideal sought after by every programmer creating new programs. It appears much easier to produce program text than it is to convince an observer of the text's correctness, however.

After many years developing theories and useful theorems for the skilled mathematician to use in proving programs, it has recently been suggested that pragmatic testing be used as a means of discovering errors in programs. Such tests can be automated, it is argued, and used by non-mathematically inclined programmers.

Testing is certainly a step forward for programmers concerned with software quality. Indeed, a broader perspective is needed in order to solve many of the problems encountered by software implementors. McKeeman (1) suggests, " ...we must realize that assertions are too constraining to become the only acceptable method of proving programs correct". It is the thesis of this paper that other "acceptable" methods do indeed exist, and furthermore, we propose such an alternative.

Geller (5) has applied a pragmatic approach to proving program correctness by combining earlier proof techniques with test data as an aid to proving programs. He states, "It is far easier to specify the relevant mathematical properties of the program ... by specifying behavior on test data than by other formal specification methods."

Geller's approach was motivated by the need for alternative methods. In fact, he suggests that one such alternative is to consider the program itself " ...as its own specification."

The operator calculus developed in this paper combines many ideas regarding program verification. In a sense, the formal operator technique merely serves to structure the orderly application of known techniques. It is tempting to designate the method a "structured proof" technique.

The symbolic execution method suggested by Handler and King (2) provides a framework for proving programs correct, also. Their approach suffers from an inability to effectively handle looping. Any alternative method must offer ease of application as a compensating merit to offset the discipline required to handle a formalism. We seek a method that works equally well with loops and assignment statements.

The suggestion that programs be designed to make proofs easier to obtain is also a concession difficult for many programmers to make. Deriving loop invariants is an example of such a philosophy. Loop invariants may acutally add to the programmer's problems if they are poor models of the process being implemented in a program. Therefore, an operator calculus must be able to transform loops into simple, single-assignment sequences without undo labor.

The goal of this research is to devise a formal (non-rigorous, but useable) operator calculus for modeling structured programs. This calculus is designed to be blindly applied by moderately skilled programmers. The proposed calculus is analogous with the formal application of operator algebras in the solution of differential equations. Most engineers are able to use Laplace Transforms without having to perform contour integration (a method of deriving the transforms).

The operator calculus is applied to program text to produce an "equivalent equation". The equation is "solved" by applying test data as a driving function of the equation, then the equation is reduced to a solution. The "reductions" are actually transformations that retain computational equivalence with the original program.

1. AN OPERATOR CALCULUS FOR ASSIGNMENT STATEMENTS

We employ a formal mathematical notation to interpretation of programming language objects. For example, a concatenated sequence of programming language statements is written as a string of statements U_i separated by the delimiting operator ";" , and defined by the equivalence operator " $\equiv$ " .

$$S_k \equiv U_1; U_2; U_3 \ldots U_k;$$

The equivalence operator states that the computation performed on either side of the equivalence symbol results in the same outcome. For example, we can establish a computational equivalence for differing statement sequences as shown in the following simple, single-assignment statement sequences.

$$y{:=}5;\ x{:=}0; \equiv x{:=}0;\ y{:=}5;$$

This illustrates how a sequence of computationally independent statements produce identical computational results due to their data independence. We say a sequence of simple, single-assignment statements is data independent if the set of variables formed by the lhs (left-hand-side) of the statements is disjoint from the set of variables formed by the rhs (right-hand-side) variables occuring in the sequence of statements. We will often demonstrate such independence by writing the sequence in concurrent form;

$$\begin{bmatrix} y{:=}5; \\ x{:=}0; \end{bmatrix} \equiv \begin{bmatrix} x{:=}0; \\ y{:=}5; \end{bmatrix}$$

A simple, single-assignment sequence is a sequence in which lhs and rhs variables are distinct. Such sequences of assignment statements are analogous with the constant function $f(x) = C_0$, because the statements can often be moved around within program segments without altering the computational equivalence of the program with its modified form. The operator calculus proposed here is used to reduce arbitrary programs to a single-assignment sequence. Hence, we seek techniques for transforming more complex sequences into an equivalent simple, single-assignment sequence.

A coupled, single-assignment sequence is a sequence in which each lhs occurs once, only, but the set of rhs variables intersect with the lhs set. For instance,

```
x:=0; y:=x+5; z:=y-2;
```

This example of a coupled, single-assignment sequence illustrates why the coupled (data-dependent) calculations cannot be executed concurrently. Hence, we must be careful with the order of their execution. This particular sequence is not computationally equivalent to the permutation below:

```
x:=0; z:=y-2; y:=x+5;
```

A coupled, single-assignment sequence can be reduced from k coupled statements to k simple statements by forward substitution of $j \leq k$ statements of the sequence. We say a coupled, single assignment sequence is c-reduced (coupled reduced) when forward substitution is used to eliminate coupling from the sequence. The example above may be c-reduced as follows.

$$[x{:=}0;\ y{:=}x+5;\ z{:=}y-2] \equiv [x{:=}0;\ y{:=}0+5;\ z{:=}y-2]$$

$$\equiv [x{:=}0;\ y{:=}5;\ z{:=}5-2]$$

$$\equiv [x{:=}0;\ y{:=}5;\ z{:=}3]$$

$$\equiv \begin{bmatrix} x{:=}0 \\ y{:=}5 \\ z{:=}3 \end{bmatrix}$$

Hence, coupled, single-assignment statements are reducible to simple, single-assignment statements. A c-reduction of a sequence of assignment statements in a manner illustrated above is an example of a synthetic execution of a program segment.

We employ synthetic execution in the programs to follow for the purpose of demonstrating their correctness relative to fixed test data. In the process of synthetic execution, we may discover sequences imbedded within programming language operators that are more complex than the assignment statement. For example, the operator IFTHENELSE imposes greater complexity on a sequence of statements than illustrated thus far. How can we formalize such complex operators and devise the corresponding reductions that allow synthetic execution? Before discussing this problem, we need methods for handling yet more sophisticated sequences of assignment statements.

A simple-recursive, single-assignment sequence is a coupled, single-assignment sequence in which one or more statements is recursively defined.

```
y:=y+1; i:=i/2;
```

The simple-recursive, single-assignment statements above are of the form,

```
variable  :=F(variable)
```

where F is an arbitrary expression function.

A forward substitution may be used to reduce the simple-recursive assignment statement, however, recursion is often used as part of a program loop. When the simple-recursive, single-assignment sequence is part of a loop, we must employ more sophisticated formal operators to the sequence in order to reduce the sequence (and the loop) to a simple, single-assignment sequence. The operator calculus for loop reduction is discussed in a later section.

A chained-recursive, simple-assignment sequence is a recursive, simple-assignment sequence in which a lhs variable appears on the rhs of other statements in the sequence.

```
a:=1-b; b:=-a;
```

A forward substitution of "a" into the right-most statement above produces a simple-recursive, single-assignment sequence from the chained-recursive single-assignment sequence.

```
[a:=1-b; b:=-a;] ≡ [a:=1-b; b:=b-1;]
```

In many cases, it may be possible to reduce a chained-recursive single-assignment sequence to a simple-recursive, single-assignment sequence by forward substitution. However, the chained-recursive statements may be cyclic in nature;

```
y:=y-x; x:=x-y;
```

in which case the substitution is unclear (indeterminate). Such a sequence is an example of a cyclic-recursive, single-assignment statement.

In many instances, a simple-recursive, single-assignment sequence may be generalized by rewriting it in non-recursive form. The simpler, non-recursive sequence makes it easier to apply synthetic execution, but may pose a problem of transforming the recursive statement into a non-recursive statement.

A transformation of a sequence S , is a formal operator induced change in S that produces the same computation as S . In other words, a transformation on S leaves S invariant with respect to results of the calculation. Consider the following program for the calculation of factorial.

```
recursive procedure RFACT(n:integer):integer;
       if n<1 then return(1);
               else return(n*RFACT(n-1));
       fi
end FACT;
```

This recursive procedure can be transformed by operator induced changes into the following computationally equivalent program.

```
procedure IFACT(n:integer):integer;
   var m:integer;
   IFACT:=1;
   if  n>1  then begin;
                  m:=n;
                  while(m>1) do
                      IFACT:=IFACT*m;
                      m:=m-1;
                  end;
            end;
   fi
   return(IFACT);
end IFACT
```

While the transformations (changes) are rather extensive for the factorial example, above, they are merely operator induced changes. Instead of using IFTHENELSE and recursion to compute RFACT, we employed IFTHENELSE and WHILE to obtain IFACT. The result obtained with IFACT is computationally equivalent to the result obtained with RFACT even though the number and form of the calculations involved is different in each procedure.

The point of all this is to demonstrate the importance of an operator formalism for expressing programs. Given a formalism for expressing either IFACT or RFACT, and formalism for reducing the expressions, we can reduce the expression to a simple, single-assignment sequence. Hence, we conjecture that every expressible program is reducible to a simple, single-assignment sequence.

<u>Conjecture</u> Every program expressible in a formal operator system D(C,R) is reducible to a simple, single-assignment sequence. System D is a set of control operators C , and a set of reduction rules R .

The remainder of this paper deals with the application of this conjecture. The conjecture can be proven true by a circular argument. Thus, only programs expressible in D will be studied, and only operators that are reducible in C by rules R will be discussed. Hence, every program expressible by the formal operators studied in this paper are reducible because only reducible operators are studied. This restriction is logically absurd, but will be shown to be a powerful idea, nonetheless.

2. AN OPERATOR CALCULUS FOR LOOPS AND CHOICE

A program is a collection of partially ordered statements whose complexity is measured by the number of predicates employed in the control structures. If the program is structured, then every control structure forms a single-entry, single-exit sequence of (nested) control structures. Hence, the most significant attribute of a structured program is its set of predicates.

Program predicates suggest a "handle" for devising an operator calculus for computer programs. The operator calculus suggested in this section is analogous with the operator calculus used by electronic engineers to analyze circuits. The circuit diagram is drawn, often from experience and intuition (the program), and then analyzed by writing an equation similar in form to the one below.

$$\frac{d^2V}{dt^2} + \frac{C}{I}\frac{dV}{dt} + K = f(t)$$

Once the differential equation is known, a set of formal operator calculus reductions are applied to derive the system performance. For instance, to solve the differential equation we supply a set of test data inputs, $f(t)$, and derive an expression for $V(t)$.

A computer program is also represented by an equation. The test data for a program is some (simple) function that generates values for the variables in the program. We "solve" the "equation" of a program by reducing the program to a simple, single-assignment sequence. As with the differential equation, the "solution" to a computer program "equation" is a function of the driving function (input data).

We use the $|$ operator to designate an IFTHENELSE predicate and the "integration" operator to designate the WHILE loop. A horizontal line is used to alternate program statement sequences. Thus, formal operators over predicate p,q, and statements u,v,w, are;

a) concatenation: u; v; w;

b) alternation: $\frac{u}{v}$;

c) decide from p : $\underset{p}{|}$

d) loop over u with step Δ :

$$\int_{q} u;\Delta$$

e) increment/decrement by v : $\overset{v}{\Delta}$ u;

Suppose we "integrate" a loop-body over some indefinite predicate. The "integral" is a function of the loop body, initial and terminal conditions, and the step size. Here are a few examples of loop integration where the initial and terminal conditions have been undeclared. (an indefinite integral).

Example 1

$\int \Delta x;\Delta i \equiv x:=x+i$

This operator is equivalent to the repeat clause of an indefinite loop.

```
repeat
  x:=x+1;
  i:=i+1
until (false);
```

Example 2

$\int \Delta^2 x;\Delta i \equiv$ x:=x+2*i;

These examples correspond to indefinite integration in sophomore calculus. The value of the integral depends on the initial conditions and the integrand's interval. The next two examples give generalized results for simple loop integration.

Example 3

$\int \Delta^k x;\Delta i \equiv$ x:=x+k*i;

Example 4

$\int \Delta^{-k} x;\Delta i \equiv$ x:=x-k*i;

Suppose we apply loop integration to the following simple summation program.

```
SUM(x[1..n]) ≡  procedure SUM(x:array[1..n]):real;
                var s:real;
                s:=0;
                for i:=1 to  n  do;
                    s:=s+x[i];
                end;
                return(s);
                end SUM
```

This simple program is transformed into the formal expression shown below.

$$\text{SUM}(x[1..n]) \equiv s:=0;\int_{1 \le i \le n} \Delta^{x[i]} s;$$

We can "solve" this equation for s given a driving function x[1..n]. Suppose we study the behavior of this equation when x[i] = c (constant).

Test #1: SUM(c..c);

$$\equiv s:=0; \int_{1\leq i\leq n} \Delta^{c} s;$$

$$\equiv s:=0; s:=n*c+s;$$

$$\equiv s:=n*c;$$

The test demonstrates the behavior of SUM when it is synthetically executed. The formal loop operator is reduced to a simple-recursive, single-assignment sequence (s:=n*c+s;) by loop integration. Next, the recursive form is reduced by forward substitution and redundant statement elimination, to a simple, single-assignment sequence. Hence, we have succeeded in demonstrating the behavior of this simple program through formal methods of operator transformation. The result is not a proof of correctness for the program; however, it is a demonstration of its synthetic execution relative to its simple inputs. Instead of giving an absolute proof, we have demonstrated the program's behavior relative to constant inputs.

A demonstration of a program is an operator calculus proof of its correctness relative to the synthetic (test data) inputs. Many demonstrations may be required before gaining confidence in the program. Suppose we apply a more difficult input to SUM.

Test #3: SUM(1..i..n);

$$\equiv s:=0; \int_{1\leq i\leq n} \Delta^{i} s;$$

$\equiv$ s:=0; s:= $\sum_{i=1}^{n}$ i+s;

$\equiv$ s:=0; s:=(n*(n-1)/2)+s;

$\equiv$ s:=(n*(n-1)/2);

The delta operator (Δ) is a shorthand form for simple-recursion in a single-statement sequence that offers a limited amount of algebraic convenience sometimes useful in reducing recursive sequences.

a) Show that $\Delta^{a}x;\ \Delta^{a}y \equiv \Delta^{a}(x;y)$

$\Delta^{a}x;\ \Delta^{a}y \equiv$ x:=x+a; y:=y+a;

$\equiv$ y:=y+a; x:=x+a;

$\equiv \Delta^{a}(x;y)$

$\equiv \Delta^{a}(y;x)$

b) Show that $\Delta^{b\pm a}y; \equiv \Delta^{b}y;\Delta^{\pm a}y;$

$\Delta^{b\pm a}y; \equiv$ y:=y+(b$\pm$a);

$\equiv$ y:=y+b; y:=y$\pm$a;

$\equiv \Delta^{b}y;\Delta^{\pm a}y;$

Let's turn now to use of the formal operator | . Suppose we derive an equation for the bubble sort algorithm. Bubble sort guarantees an output list in ascending order. That is, every element $x[i] \leq x[j]$ for $i \leq j$. We express this condition in succinct predicate form as follows.

x[i] $\leq$ ALL(x[j];) where $j \geq i$ ..

Furthermore, this condition must exist over all values of i . Hence we can integrate the predicate:

$$\int_{1\le i\le n} \Big|_{x[i]\le\ x[j]} \quad ;\Delta i;$$

The value of subscript j is undefined in the intergrand, but we want to run over all values greater than i . This assures that every element of x is greater than every other element of x whenever $j \ge i$. A second integration provides the needed values of j .

bubble(x[1..n]) ≡

$$\int_{1\le i\le n} \int_{i<j\le n} \Big|_{x[i]\le x[j]} \quad ;\Delta j;\Delta i;$$

The bubble sort algorithm employs an exhange or "ripple" to exchange x[i] with x[j] whenever the predicate fails to be true. We include this possibility in the equation using the alternation operator. The upper path is selected when the predicate in | is true, and the lower path selected, otherwise.

bubble(x[1..n]) ≡

$$\int_{1\le i\le n} \int_{i<j\le n} \Big|_{x[i]\le x[j]} \quad \frac{\qquad\qquad}{x[j] \leftrightarrow x[i]};\Delta j;\Delta i;$$

The equation above may be used to derive a program segment for bubble sort as shown below.

```
for  i:=1  to  n; do;
              begin; for j:=i+1 to n do
                             if x[i]≤x[j] then;
                               else x[i] ↔  x[j];
                             fi;
                  end;
end;
```

To demonstrate that the bubble program works as expected we reduce its equation. First, we assume a constant input; next an ordered input; and finally we study a descending sequence of inputs.

<u>Test #1: bubble(c..c);</u>

$$\equiv \int_{1\le i\le n} \int_{i<j\le n} \Big|_{c\le c} \overline{\quad c \leftrightarrow c \quad};\Delta j;\Delta i;$$

since $\Big|_{c\le c}$ is always true, we can simplify:

$$\equiv \int_{1\le i\le n} \int_{i<j\le n} \Delta j;\Delta i;$$

$$\equiv \int_{1\le i\le n} j{:=}n+1;\Delta i;$$

bubble(c..c) $\equiv$ j:=n+1; i:=n+1;

The definite integration is performed in each case by evaluating both upper and lower limits of the loop integration. For example,

$$\int_{1\le i\le n} \Delta i \equiv i{:=}1;\ i{:=}i+(n);$$

$$\equiv i{:=}1+n;$$

<u>Test #2: bubble (1..i..n);</u>

$$\equiv \int_{1\le i\le n} \int_{i<j\le n} \Big|_{i\le j} \overline{\qquad i \leftrightarrow j \qquad};\Delta j;\Delta i;$$

Again, since the input is ordered, $i\le j$ is always true over $i<j\le n$ (the loop predicate). Hence, the alternation path is reduced to the null case.

$$\equiv \int_{1\le i\le n} \int_{i<j\le n} \Delta j;\Delta i; \;\equiv\; j:=n+1;\; i:=n+1$$

Finally, suppose we demonstrate that bubble works correctly with a descending list. Let $x[i] \equiv (-i)$.

<u>Test #3 bubble(-1..-n);</u>

$$\equiv \int_{1\le i\le n} \int_{i<j\le n} \Big|_{-i\le -j} \frac{\qquad\qquad}{x[i] \leftrightarrow x[j]};\Delta j;\Delta i;$$

$$\equiv \int_{1\le i\le n} \int_{i<j\le n} \Big|_{i>j} \frac{\qquad\qquad}{x[i] \leftrightarrow x[j]};\Delta j;\Delta i;$$

This time, when the $|$ predicate is false, we reduce the alternation path to an exchange. As it turns out, the $|$ predicate is always false, leading to a circular shift of the array elements.

$$\equiv \int_{1\le i\le n} \int_{i<j\le n} x[i] \leftrightarrow x[j];\Delta j;\Delta i;$$

The resulting integration of the inner loop shows how the final value of x is "rotated" to the start of the sublist beginning at position i.

$$\equiv \int_{1\le i\le n} x[i]:=x[n-i+1];\; j:=n+1;\Delta i;$$

$$\equiv\; j:=n+1;\; i:=n+1; \begin{bmatrix} x[1]:= x[n]; \\ x[2]:= x[n-1]; \\ x[3]:= x[n-2]; \\ \vdots \\ x[n-1] := x[2] \end{bmatrix}$$

Note, in the final exchange, the last element of the list is not exchanged with itself because $(i<j\le n)$ would become false. Therefore, the list $(-1,-2,-3,\ldots -n)$ is reversed to give $(-n,-(n-1), \ldots,-1)$.

3. DIFFICULT TRANSFORMATIONS

The examples studied thus far are simple programs easily analyzed by other methods. The advantage of alternate program verification and/or derivation techniques must lie in their ability to analyze intellectually difficult programs. Furthermore, the formal operator technique provides a way to systematize the analysis of programs. Such systemization suggests possible automatic means of applying the operator transformations. The automation of program demonstrations has not been pursued todate, however.

The greatest-common-denominator program is often used as an example of an intellectually difficult program to prove (2). The source of difficulty is the coupled-recursive sequence contained within the loop body of GCD.

```
GCD(a,b) ≡ procedure GCD(a,b:real):real;
           while (a≠b) do
                if a>b then a:=a-b;
                       else b:=b-a;
                fi
           end;
           return (a);
           end GCD
```

Handler and King (2) explore the GCD algorithm using a symbolic execution tree. The tree helps to prove the program correct, but application of the proof technique takes the programmer far from the task of verification. How well does the formal operator calculus perform when applied to the GCD algorithm?

$$\mathrm{GCD}(a,b) \equiv \int_{a\neq b} \Big|_{a>b} \frac{a:=a-b}{b:=b-a};\Delta$$

$$\equiv \int_{a\neq b} \Big|_{a>b} \frac{\Delta^{-b}a}{\Delta^{-a}b}$$

How can we test (demonstrate) the GCD equation? Since the alternation paths are chained-recursive, we are at a loss to invent an input that exercises the coupled body of the loop.

The GCD program is usually proven correct employing mathematical properties of the GCD function. This approach is taken here. We note that $\mathrm{GCD}(a,a)=a$ if $a>0$, $\mathrm{GCD}(a,b) \equiv \mathrm{GCD}(b,a)$, and most importantly that $\mathrm{GCD}(a,b) \equiv \mathrm{GCD}(a+b,b)$. If these three relations are shown to hold, then we not only have demonstrated the GCD program, but we have proven it correct as surely as if traditional proof techniques were employed.

<u>Test #1: $\mathrm{GCD}(a,a) \equiv a$ if $a>0$.</u>

$$\equiv \int_{a\neq a} \Big|_{a>a} \frac{\Delta^{-a}a}{\Delta^{-a}a}$$

$$\equiv \int_{a\neq a} \Delta^{-a} a$$

$$\equiv$$

The result of this demonstration is a null sequence containing no simple, single-assignment statements. The reason is that the loop is never integrated, hence the output of GCD(a,a) is a .

Test #2: $GCD(a,b) \equiv GCD(b,a)$

$$\int_{a\neq b} \Big|_{a>b} \frac{\Delta^{-b}a}{\Delta^{-a}b} \equiv \int_{b\neq a} \Big|_{b>a} \frac{\Delta^{-a}b}{\Delta^{-b}a}$$

Since $\Big|_{b>a} \dfrac{\Delta^{-a}b}{\Delta^{-b}a} \equiv \Big|_{b\leq a} \dfrac{\Delta^{-b}a}{\Delta^{-a}b}$

is an identity relation, and since the single value $(b-a)$ cannot exist within the loop body;

$$\int_{b\neq a} \Big|_{b\leq a} \equiv \int_{b\neq a} \Big|_{b<a}$$

We derive the equivalence by direct substitution of the identity.

Test #3: $GCD(a,b) \equiv GCD(a+b,b)$

$$\begin{aligned}\int_{(a+b)\neq b} \Big|_{(a+b)>b} \frac{\Delta^{-b}(a+b)}{\Delta^{-(a+b)}b} &\equiv a:=a+b-b;\ GCD(a,b)\\ &\equiv a:=a;\ GCD(a,b)\\ &\equiv GCD(a,b)\end{aligned}$$

The basis of this demonstration is quite simple. The first cycle through the loop produces:

$(a+b)\neq b \equiv TRUE$

$(a+b)>b \equiv TRUE$

thus, $\Delta^{-b}(a+b) \equiv a:=a+b-b$

The next (and subsequent) execution of the loop uses the result a:=a to produce an execution identical to GCD(a,b). In short, the first pass through the loop replaces every occurence of (a+b) with the value (a) .

We could also demonstrate that GCD(a,a+b) ≡ GCD(a,b) by a similar argument.

The GCD program is intellectually difficult because of the chain-recursive coupling of variables. We were able to sidestep this difficulty by relying on a mathematical relationship defined for GCD over different test data. This is not always possible, and indeed the problem of demonstrating a program may be as difficult as proving it correct.

Perhaps one reason to demonstrate a difficult program even when the demonstration is difficult to do, is that the demonstration provides another handle for the same problem. We illustrate this idea with a program that was proven correct by Dershowitz and Manna (3), but according to the following demonstration, is incorrect. Indeed, the program fails to produce correct results for a very simple test data input.

Dershowitz and Manna use a loop-invariant version of the binary search algorithm. The loop-invariant provides a way to prove the program correct, but alas the program is incorrect! While useful, the loop-invariant is not needed for the following demonstration.

$$\text{binary}(a[1..n],b) \equiv z{:=}n; \int_{-n\le y<-1} y{:=}y/2; \Big|_{b\le a[z+y]} \overline{\overset{y}{\Delta}\, z} ;\Delta$$

This equation was derived from the program given in reference (3). It illustrates the use of a convergent loop invariant y:=y/2; z:=z+y; which converges to the location of element b in array a[1..n] .

The binary search algorithm is intellectually difficult because of the simple-recursive, single-assignment sequence imbedded within the loop integrand. We overcome this difficulty by removing the recursive statements and replacing them with non-recursive equivalents. In otherwords, we must first transform the equation into a computationally equivalent equation using formal operator changes.

The indefinite integrals for y and z are obtained by transformation of recursion, as follows.

$$\int y:=y/2;\Delta \equiv y:=y_o; \int (y/2);\Delta y$$
$$\equiv \int y:=y_o(\tfrac{1}{2})^k;\Delta k$$

This relation was obtained by solving the difference equation:

$$y_k = (y_{k-1})/2 \quad \text{and} \quad y_o=(-n)$$

Furthermore, we can derive a non-recursive version of z by solving its difference equation. Thus, the transformed equation for binary search is:

binary(a[1..n],b) $\equiv$

$$\int_{-n\le y<-1} y:=-n/2^{**}k; \quad \Big|_{b\le a[z+y]} \frac{z:=n/2^{**}k}{};\Delta$$

Now, every statement in the integrand is invariant with respect to loop integration. The only variable affected by the loop is k , which was introduced to generalize the expressions for y and z . This is the concept underlying loop invariance proposed by Dijkstra, but used in a somewhat reverse fashion. The operator calculus removes induction from the proof and replaces it with integration. The next step in solving this equation is to integrate the simple, single-assignment statements by computing a value of k that terminates the loop. The reader can clearly see the loop invariant in action by carefully studying the following demonstrations.

<u>Test #1: binary(1..n,1)</u>

Suppose we demonstrate this program with test data a[i] = i, and b=1.

$$\equiv \int_{-n \leq y < -1} y{:=}{-n/2{*}{*}k}; \quad \Big|_{1 \leq a[z+y]} \quad \underline{z{:=}n/2{*}{*}k};\Delta$$

Since the loop hopefully terminates with b=1=a(1), the predicate for $|$ is always true. We reduce the alternation paths to the evaluation of z . This means that every pass through the loop produces the following subscript for a[z+y].

$$\cdot \quad a[z+y] \equiv a[n/2{*}{*}(k-1)-n/2{*}{*}(k)] \equiv a[n/2{*}{*}k]$$

When $y \geq -1$ (loop termination) we get $-1 \leq -n/2^k$, which produces a value for k ;

$$k = \lceil \log_2 n \rceil$$

Thus, k is the ceiling of the logarithm of n . When n is <u>not</u> a power of two, the value of k is selected such that $2^k > n$. The loop terminates when we have integrated over $k=0,1,2...[\log_2 n]$.

$$\text{binary}(1..n,1) \equiv$$

$$y \geq -1;\ k = \lceil \log_2 n \rceil;\ 1 \leq a[n/2{*}{*}k];\ z{:=}n/2{*}{*}k;$$

$$\equiv y \geq -1;\ 1 \leq a[n/2{*}{*}\lceil \log_2 n \rceil];\ z{:=}n/2{*}{*}\lceil \log_2 n \rceil$$

<u>Case A</u>: $\lceil \log_2 n \rceil \equiv \log_2 n$

Then,

$$2{*}{*}\lceil \log_2 n \rceil \equiv n$$

And,

$$\begin{aligned} \text{binary}(1..n,1) &\equiv 1 \le a[1];\ z:=1 \\ &\equiv 1 \le 1;\ z:=1 \end{aligned}$$

<u>Case B</u>: $\lceil \log_2 n \rceil \equiv \log_2 n \pm 1$

Then,

$$2{*}{*}\lceil \log_2 n \rceil \equiv 2^{\pm 1} {*} n$$

thus,

$$\text{binary}(1..n,1) \equiv 1 \le a[2^{\mp 1}];\ z:=2^{\mp 1};$$

The binary program succeeds in case A, above, but fails due to array indexing errors in case B. Hence, when the length of array "a" is a power of two, this version of binary is correct, however, the program "blows up" when n is not a power of two. The reader is encouraged to demonstrate another problem area for the Dershowitz-Manna version of binary search; try to demonstrate binary(1..n,n).

The traditional binary search algorithm is implemented without loop invariance. The following equation mirrors such a traditional program. Notice the divide-and-conquer steps that take place by moving an upper and lower pointer (j and i) to designate upper and lower sublists within array x.

binary(x(1..n),k) $\equiv$

$$\underset{x[1]\leq k\leq x[n]}{\Big|}\ \frac{i{:=}1;\ j{:=}n;\ m{:=}\frac{n+1}{2};\ \int_{k\neq x[m]}\ \underset{x[m]>k}{\Big|}\ \frac{i{:=}m-1}{i{:=}m+1};\ m{:=}\frac{i+j}{2};\Delta}{m{:=}0}$$

This version of binary returns m:=0 if the key k is not within the list. Furthermore, the loop is integrated over code that is executed only when the key is not found. We transform this equation into a simple, single-assignment sequence containing the value of the matching element's subscript in m , and predicates that convince us that the value is indeed correct.

<u>Test #1: binary(1..n,k)</u>

$$\equiv\ \underset{1\leq k\leq n}{\Big|}\ \frac{i{:=}1;\ j{:=}n;\ m{:=}\frac{n+1}{2};\ \int_{k\neq m}\ \underset{m>k}{\Big|}\ \frac{j{:=}m-1}{i{:=}m+1};\ m{:}\frac{i+j}{2};\Delta}{m{:=}0;}$$

$$\equiv\ \frac{1\leq k\leq n;\ \begin{bmatrix} i{:=}1; \\ j{:=}n; \\ m{:=}\frac{n+1}{2} \end{bmatrix}\ \int_{k\neq m}\ \underset{m>k}{\Big|}\ \frac{j{:=}m-1}{i{:=}m+1};\ m{:=}\frac{i+j}{2};\Delta}{1\ngtr k>n;\ m{:=}0;}$$

We obtained the alternating paths above by splitting the predicate of the outer | operator. If we do this again for the | inside the intergrand, we get the same equation, but with a new integrand, as follows.

$$\int_{k\neq m} \frac{m>k;\ j:=m-1;}{m\leq k;\ i:=m+1;}\ m:=\frac{i+j}{2};\Delta$$

These paths are reduced by forward substitution and noting the integer-valued results.

$$m>k;\ j:=m-1 \equiv j>k-1 \equiv j:=k$$

$$m\leq k;\ i:=m+1 \equiv i\leq k+1 \equiv i:=k$$

We substitute these reductions into the integrand and get the following.

binary(1..n,k) $\equiv$

$$\frac{1\leq k\leq n;\begin{bmatrix} i:=1; \\ j:=n; \\ m:=\frac{n+1}{2}; \end{bmatrix}\int_{k\neq m} \frac{j:=k}{i:=k};m:=\frac{i+j}{2};\Delta}{1>k>n;\ m:=0}$$

Technically, we should have used the reduced forms

$$m>k;\ j:=k$$

$$m\leq k;\ j:=k$$

in the integrand, but in either case the results merge due to their equivalence.

Forward substitution gives a result for i,j, and m . The result produces

$$m:=\frac{k+k}{2} \equiv m:=k$$

which causes loop integration and reduction.

$$\equiv \frac{1\le k\le n;\ \begin{bmatrix} i:=1; \\ j:=n; \\ m:=\frac{n+1}{2}\ ; \end{bmatrix} \underset{k\neq m}{\int} \quad m:=k;\ j:=k;\ i:=k;\Delta}{1>k>n;\ m:=0;}$$

$$\equiv \frac{1\le k\le n;\ i:=k;\ j:=k;\ m:=k}{1>k>n;\ m:=0;}$$

This demonstrates the correctness of this version of the binary search with respect to an ascending list of integers.

The binary search is easily derived from the equation just used.

```
binary(x[1..n],k) ≡

     procedure BINARY(x:array[1..n], k:integer, m:integer);
     var    i,j:integer;
     if     x[1]<k<x[n] then begin
            i:=1; j:=n; m:=(n+1)/2;
            while (k≠x[m]) do
                 if x[m]>k then j:=m-1;
                           else i:=m+1;
                 fi;
                 m:=(i+j)/2;
                 end;
          else m:=0;
     fi;
     return;
end BINARY
```

The demonstration technique is powerful enough to compete with more formal proof techniques. There is a danger in using the method, however.

Success with synthetic execution may lull the user into believing the results too strongly. The demonstration does not guarantee correctness any more than formal techniques as used incorrectly in reference (3).. Indeed, it could be argued that demonstration is a much weaker method than proof by inductive assertion. The following recursive definition of Quicksort illustrates how one can be misled by initial success.

```
QUICK(x[1..n]) ≡  j:=n; i:=1; T:=x[n/2];

  ∫  |  QUICK(x[1..i]); QUICK(x[j+1..n]);Δ^∞ ;Δ
                        x[i] ↔ x[j]; Δi
        |          |
     x[i]>T     x[j]<T              Δ^-1 j
                ______________________________
                   |
                x[j]<T             Δi
                          ______________________

                               -1
                              Δ   j; Δi;
```

The Δ^{∞} term is a "breakloop" statement that terminates the loop integration. Hence, the loop is terminated after double recursion on the top path of the alternations rather than by a loop predicate.

Demonstrating QUICK(c..c) and QUICK(1..n) produces success, but careful scrutiny of the equation reveals that the algorithm is indeed incorrect. Selecting n=3, and simple values for x[i] immediately causes array overflow or underflow in the algorithm. Thus, we are cautioned not to use the formal techniques without careful analysis of the test data.

4. RESEARCH TOPICS

The foregoing operational calculus is merely an initial step toward a complete theory of programs, program equations, and transformations leading to program reductions.

The problems that still remain to be examined are:

1) extend the transformations to coupled sequences of greater generality and sophistication.

2) formulate a theory of test data selection that increases the chances that all possibilities have been considered. At first glance it appears that testing boundary conditions on the data is most fruitful, but the criterion are more complex than this. Test data selection techniques employed by traditional "data domain" techniques seem to apply equally to this method (4,5).

3) investigate automated equation-solving systems that read a program, produce the transformed equation, and demonstrate the equation by synthetic execution.

Perhaps the most significant contribution of a formal operator calculus for expressing computer programs is the insight obtained by viewing a program as a physical system. Physical systems are modeled by linear and non-linear differential equations, most of which are easily solved using non-rigorous mathematical manipulation. The same can be achieved with computer programs once we develop a mathematical understanding of operator notation and its use in programming.

REFERENCES

(1) McKeeman, W.M., On Preventing Programming Languages From Interfering With Programming, IEEE Trans. Soft. Engr., SE-1, 1, (March 1975), 19-25.

(2) Handler, S.L. and King, J.C., An Introduction To Proving The Correctness of Programs, Comp. Survey, 8,3, (Sept. 1976), 331-353.

(3) Dershowitz, and Manna, F., IEEE Trans. Soft. Engr., SE-3, 6, (Nov.1977).

(4) Goodenough, J.B., and Gerhart, S.L., Toward a Theory of Test Data Selection, Proc. Int. Conf. on Reliable Software, 1975, pp. 493-510.

(5) Geller, M., Test Data As An Aid In Proving Program Correctness, Comm. ACM, 21, 5, (May 1978), pp. 368-375

An Operator Calculus For Concurrent Computer Programs

Part II

T. G. Lewis
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

(503) 754-3273

Report

78-1-5

Contents

Abstract

The formal operational calculus is extended to include concurrent programs typically used to construct operating systems. An interleave principle is used to define a process-state matrix. The matrix is used to construct execution paths representing concurrent (asynchronous) execution of multiple processes. The "correctness" of concurrent programs is demonstrated by studying the paths given by the process-state matrix.

The technique is applied to Dekker's algorithm, several faulty semaphore's, and a device monitor suggested by Wirth's Modula programming language. The method performs moderately well as shown by success in demonstrating small programs. Directions for future research indicate fruitful areas of further investigation.

Motivation

The research reported in Part I [1] describes a formal operator calculus for demonstrating programs. The term "demonstration" is used to distinguish the methodology from proof of correctness techniques. A demonstration is a symbolic execution of a computer program carried out by submitting test data to the program. The value of such an approach depends heavily on the appropriateness of test data; often a program appears to be correct with one set of test data, and incorrect with another set of (unused) data. Hence, a demonstration provides additional evidence to support the conjecture that a program is correct.

The symbolic execution of a program and its test data is carried out in a structured way. A formal operator calculus is used to transform the source code of a program into an equation. This so-called equation-of-state for the program is completely analogous to a differential equation used to describe the behavior of a physical system. Given a driving function (test data), a differential equation can be solved using a set of formal operators to reduce the differentiation terms to functions defining the solution through time. The solution is accepted as a model of the physical system's behavior.

In deriving a solution to a computer program equation, we apply formal operators that reduce the program to a sequence of simple, single-assignment statements. Hence, the solution to every computer program equation is a sequence of single-assignment statements.

Part I [1] described the formal operators as follows.

A. The equivalence operator, $\equiv$

B. The step operator, Δ

$\Delta^x y \equiv$ y:=y+x;

C. The forward substitution, operation:

y:=a;x:=x+y $\equiv$ y:=a;x:=x+a;

D. The alternation operator, $|$

$$\Big|_{p} \frac{s_{true}}{s_{false}}$$

where p is a predicate, and s_{true} is used when p is true, s_{false} is used otherwise.

E. The loop integration operator, $\int$

$\int_p \Delta$;

where p is a loop predicate; the loop is executed forever, or until p becomes false.

F. The parallel operator, $\Big[\quad \Big]$

$\begin{bmatrix} y:=a \\ x:=b \end{bmatrix}$; both statements can be done in parallel.

The examples of Part I illustrated the power and versatility of this operator calculus. In short, the equations derived by "solving" programs range from simple execution of code without conclusive proof of its correctness, to correctness proofs as valid as proof by assertion, and proofs using loop invariants with induction.

The purpose of this report is to extend the formal operator notation and range of usefulness to include concurrent program demonstrations. The idea is very straight forward. We want a theory that can be used by non-mathematically inclined programmers to demonstrate that concurrent programs are correct when executed by one or more concurrent (asynchronous) processes running on one or more computers.

A program is said to be concurrently executable if it is re-entrant and overcomes two problems associated with multiprocess program execution: 1) critical sections can be shown to solve the semaphore problem for shared data, and 2) processes are deadlock-free while executing the program.

The Interleave Principle

The operator calculus is applied to a "concurrent" program by assigning a process "tag" to the program, and then deriving the equation-of-state for the tagged program. The equation-of-state for the program is derived using the formal operators given, above, and illustrated in Part 1 [1].

A demonstration of a concurrent program Q with statements $q_0\ q_1\ q_2 \ldots q_k$ is achieved by tagging the program with a process label, say P1, and then interleaving the processes with every statement of the program.

$$P1{:}Q \equiv P1{:}q_0;P1{:}q_1;P1{:}q_2 \ldots P1{:}q_k$$

We interleave a second process also executing program Q as follows.

$$P1{:}q_0;P2{:}Q;P1{:}q_1;P2{:}Q;P1{:}q_2;\ldots P2{:}Q;P1{:}q_k;$$

The "solution" to the equation above produces a sequence of single-assignment statements that represent all execution paths of a concurrently executable program. If the solution fails to reduce to a sequence of single-assignment statements, then we say the program is incapable of supporting concurrency.

An illustration of the Interleave Principle is shown below using a program for mutual exclusion (page 45, problem 5, [2]). This program attempts to implement mutual exclusion by busy waiting. We assume each statement is indivisible during execution because of memory interlock. We cannot, however, assume that two processes, P1 and P2 enter and execute this program in any prescribed order.

```
MUTEXBEGIN:
          need [me]:=TRUE;
          DO WHILE  (need [other]) ;END;
          CR: /* critical region */
MUTEXEND:  need [me]:=FALSE
```

This code is re-written in formal operator notation:

$$Q \equiv \text{need[me]:=T} \; \int_{\text{need[other]}} \Delta \; ; \; \text{CR}; \; \text{need[me]:=F}$$

We begin a process by initializing the shared variables need [1], need [2], to F (false). Furthermore, process P1 owns me=1, and process P2 owns me=2. In either case, other=2 or 1, respectively. Hence, execution of Q by P1 and P2 give program activations as follows (as viewed by process P1):

$$\text{P1:Q} \equiv \text{need[me]:=T}; \int_{\text{need[other]}} \Delta \; ; \; \text{CR}; \; \text{need[me]:=F}$$

$$\text{P2:Q} \equiv \text{need[other]:=F}; \int_{\text{need[me]}} \Delta \; ; \; \text{CR}; \; \text{need[other]:=F}$$

Initially need[1]=need[2]=F, and we assume P1 is executed first. The Interleave Principle is used next to solve this set of equations (simultaneous linear equations?).

$\equiv$ P1:need[me]:=T;P2:Q; $\int_{\text{need[other]}} \Delta$;P2:Q;CR;P2:Q;need[me]:=F;

We need only expand the first P2:Q statement, above to demonstrate that this program fails as a concurrently executable program.

$\equiv$ P1:need[me]:=T;P2:need[other]:=T; $\int_{\text{need[me]}} \Delta$;CR;need[other]:=F;P1: $\int_{\text{need[other]}} \Delta$;

This equation is reducible to the first loop integration, which never terminates; the computation in P2 is blocked.

$\equiv$ P1:need[me]:=T;P2:need[other]:=T; $\int_{\text{need[me]}} \Delta$; ...

The remaining part of the equation also terminates due to blocking.

... P1: $\int_{\text{need[other]}} \Delta$; P2:Q ...

Hence, we rapidly conclude that this program leads to deadlocked processes. If we start P2 first, and Interleave P1 after every statement of P2, the processes are blocked in reverse order, and lead to deadlock, also.

The Interleave Principle revealed a fault in the mutual exclusion example above because it produces an equation for every possible execution path through the two processes. Indeed, the symbolic execution of every path becomes a tangle of expressions when using the operator notation, above. A visual aid to symbolic execution is more useful.

The Process-State Matrix

The operator calculus notation can be used along with a matrix of process states to reveal the structure of concurrent program execution. A process-state matrix is a two-dimensional matrix A[n,n] such that the square at a[i,j] represents the state of process P1 after execution of statement i; and the state of process P2 after execution of statement j of the concurrent program in question.

The process-state matrix is constructed by drawing a grid. One horizontal line is drawn for each statement executed by P1; one vertical line for each statement executed by P2.

The upper left-hand square (the "home" square, or starting state) represents simultaneous entry into the concurrent program by two processes. Which process actually gains first entry is determined by initial values of shared and local variables.

As an illustration, we take an improved version of the mutual exclusion program shown to be deadlock-prone (see page 45, problem 5, [2]).

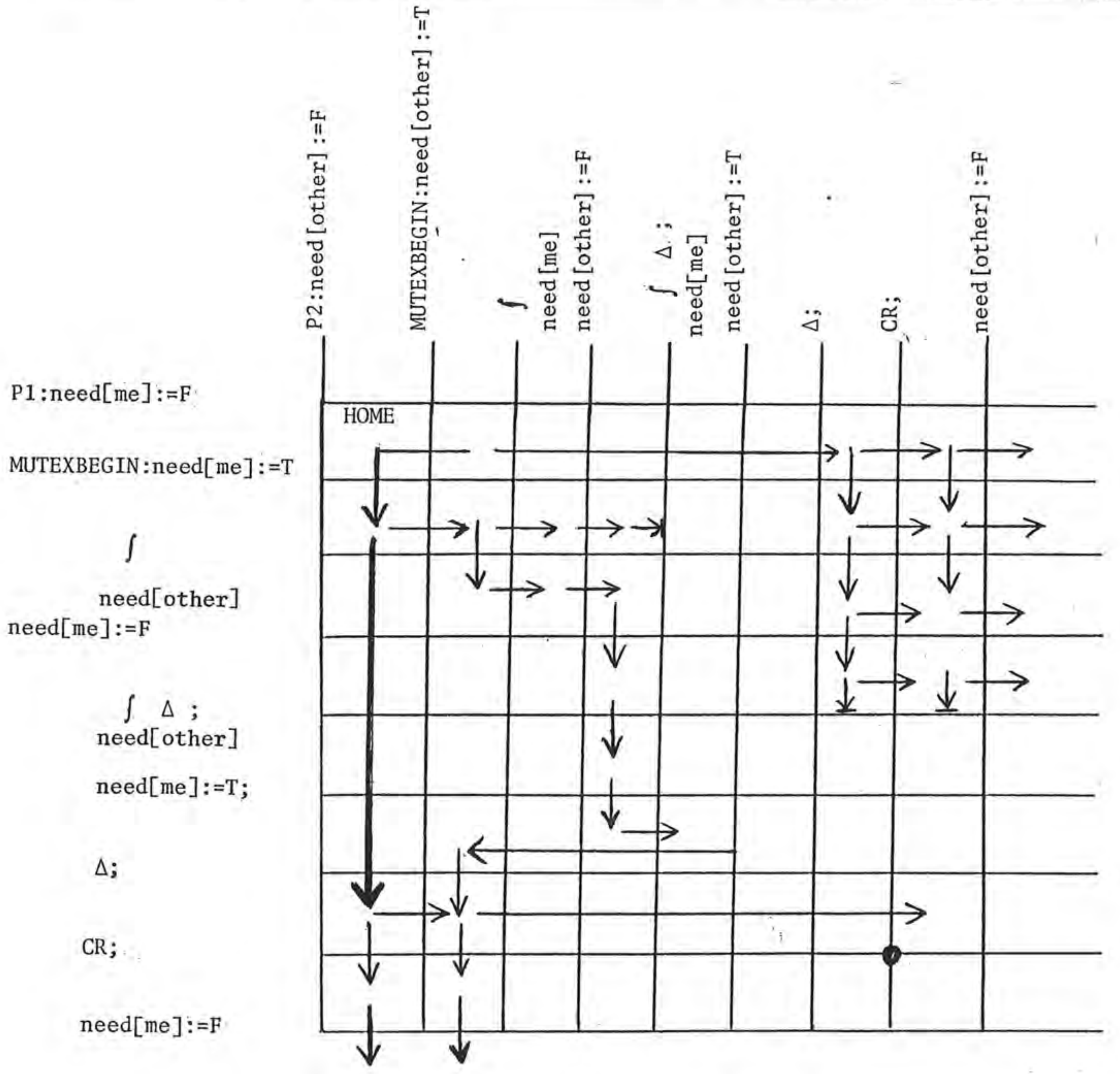
```
MUTEXBEGIN:
          need[me]:=TRUE;
          DO WHILE (need[other]);
              need[me]:=FALSE;
              DO WHILE (need[other]);
                  END;
              need[me]:=TRUE;
          END;
          CR;
          need[me]:=FALSE;
```

This program also begins execution with local variables "me" and "other" (equal to 1, 2 in process P1, and 2, 1 in process P2), and need [1], need [2] equal to FALSE (F).

The operator calculus version of this program is:

$$\text{P1:MUTEX} \equiv \text{need[me]:=T};\ \int_{\text{need[other]}} \text{need[me]:=F};\ \int_{\text{need[other]}} \Delta\ ;\text{need[me]:=T};\Delta;\text{CR};\text{need[me]:=F}$$

The process-state matrix for MUTEX is given below (also showing the home square).



We have drawn only a small portion of the total number of paths possible, above. The path algorithm is as follows: 1) begin in the home square, 2) draw an arrow connecting one or more squares with the home square, (the arrow represents a one-step concurrent execution of either P1 or P2), 3) continue to draw arrows representing state transitions for either P1 or P2, and 4) when a block is encountered indicate the block with a bar in the square being blocked.

The demonstration of a concurrent program is based on three fundamental characteristics of "correct" programs.

1) (Mutual exclusion): Only one process can be in the critical region at a time. Hence, the process-state matrix must contain no path passing through the squares adjacent to (CR,CR).

2) (No permanent blocking): If no other process is in CR, then any other process can enter the CR. Hence, there exists at least one path from the home square that passes through a CR square (crosses the CR line). Furthermore, there is a path for every process passing through CR, e.g. P1:CR and P2:CR are both possible.

3) (No starvation): No set of timings can keep a process waiting, indefinitely. Hence, there does not exist a path through one process's critical region that bypasses another process's critical region.

Let's examine each of the three fundamental characteristics, above with respect to the process-state matrix. The demonstration of the last example of MUTEX reveals a violation of mutual exclusion.

1) (Mutual exclusion): A path does exist leading to the (CR,CR) intersection. Hence, the program fails to prevent indeterminism during concurrent execution. The path is shown in the process-state matrix for the following symbolic execution.

$\equiv$ P1:need[me]:=T ; P2:need[other]:=T ;

P1: $\int_{\text{need[other]}}$ P2: $\int_{\text{need[me]}}$ need[other]:=F ...

P1:need[me]:=F ; $\int_{\text{need[other]}} \Delta$; P2: $\int_{\text{need[me]}} \Delta$;need[other]:=T

P1:need[me]:=T ; Δ ; P2: $\int_{\text{need[me]}} \Delta$; CR

P1: CR; ...

This path leads to concurrent entry into CR by P2 first, then P1 (or vice-versa) as shown by the last two terms in the equation, above.

2) (No permanent blocking): A fully drawn set of paths through the process-state matrix will reveal that no permanent blocking exists in this version of MUTEX. The squares containing a block also contain an alternate arrow that is not blocked. Hence, this characteristic is satisfied.

3) (No starvation). Starvation is possible. For P1 there exists a path through CR that does not also pass through P2:CR. Rigorously, the path should lie to the right of the P2:need[other]:=T line since starvation of P2 is meaningless

unless P2 requests access. The following perpetual loop is possible.

```
P1:need[me]:=T; P2:need[other]:=F;
P1: CR;         P2:need[other]:=T
P1:need[me]:=F;  MUTEXBEGIN:need[me]:=T
                P2:  ∫ ...
                    need[me]
```

It is possible, as shown above, for process P2 to quickly request a second, third, etc. access before P2 has a chance to break from its wait loop. Thus, this particular set of timings leads to starvation of P2.

We have shown two examples of mutual exclusion algorithms that fail to provide the necessary characteristics of concurrent programs. The formal operator calculus in its two-dimensional format has demonstrated faults in the routines. Hence, we have greater confidence in its ability to test concurrent programs. However, the question remains, "what does this method offer for certifying correct algorithms?"

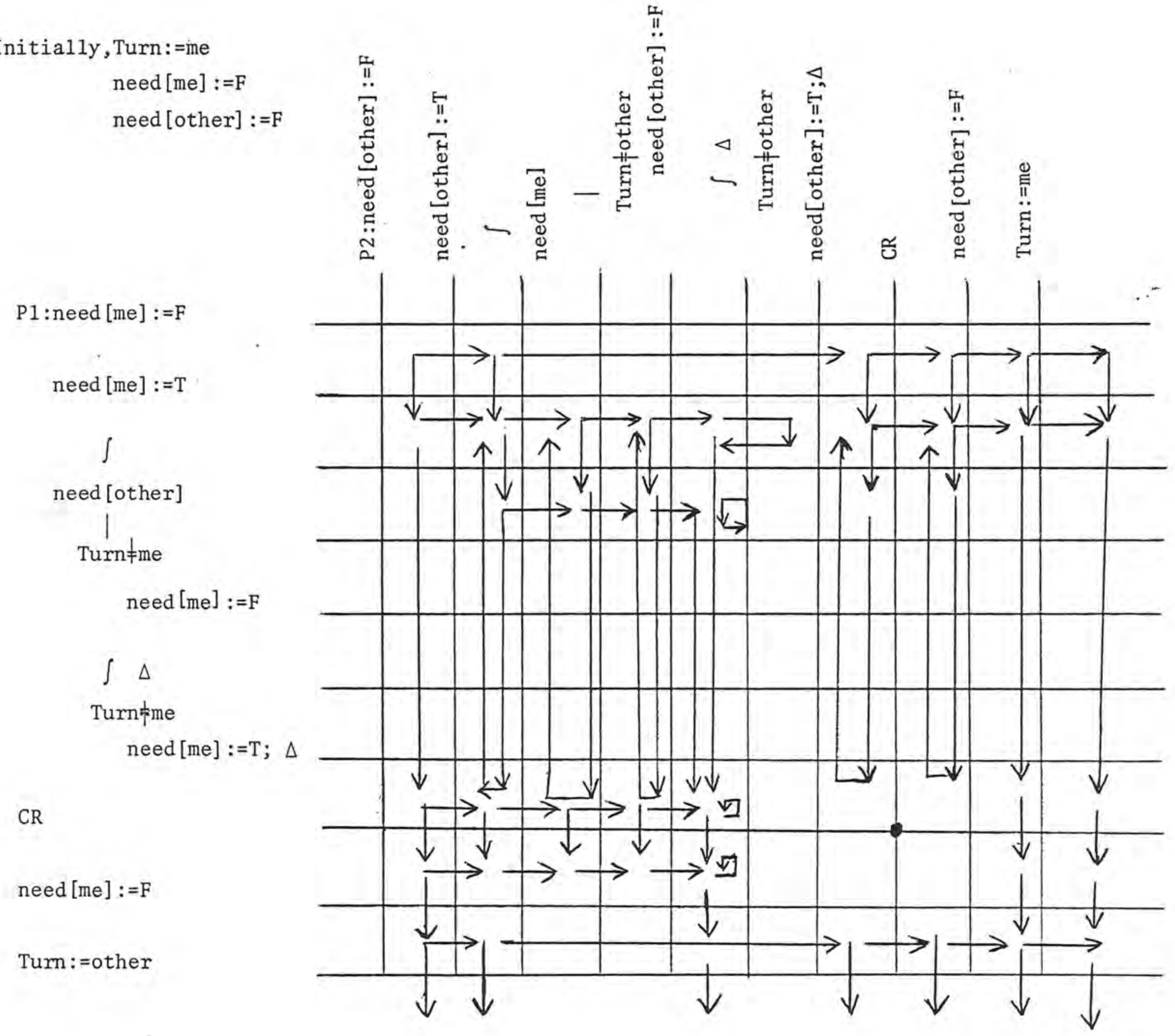
Dekker's algorithm [2] is a busy-waiting algorithm commonly accepted as a "good" mutual exclusion algorithm when memory interlock (but no "test-and-set") is the only mechanism for preventing indeterminism. The algorithm is given in succinct form in reference [1], page 25.

```
MUTEXBEGIN:
    need[me]:=TRUE;
    DO WHILE (need[other]);
        IF  turn ¬ = me  THEN
             DO; need[me]:=FALSE:
                 DO WHILE (turn ¬ = me);
                 END;
             need[me]:=TRUE;
             END;
END;
CR;
need[me]:=FALSE
turn:=other;
```

Dekker's algorithm uses shared variables need [1], need [2], and turn, to indicate pending requests from P1 or P2 and whose turn it is to have access. Their initial values are FALSE,FALSE, and turn=me.

The following process-state matrix for Dekker's algorithm reveals a possibility of process starvation, but guarantees mutual exclusion.

```
Initially, Turn:=me
          need[me]:=F
          need[other]:=F
```



Checking the three characteristics of a good concurrent program:

1) The intersection (CR,CR) cannot be reached, hence Dekker's algorithm guarantees mutual exclusion.

2) There exists at least one path to P1:CR and one path to P2:CR.

3) There exists a path from home to P2:CR and P1:CR, hence no starvation is likely. However, there also exists a path from home to P1:CR that does not pass through P2:CR, hence starvation is possible.

P and V Operators

A synchronizing semaphore can be implemented on machines with indivisible "test-and-set" instructions. Such machines employ the Testandset instruction instead of a busy-wait loop to prevent concurrent (indeterminate) access to the critical region encased in P-V operators.

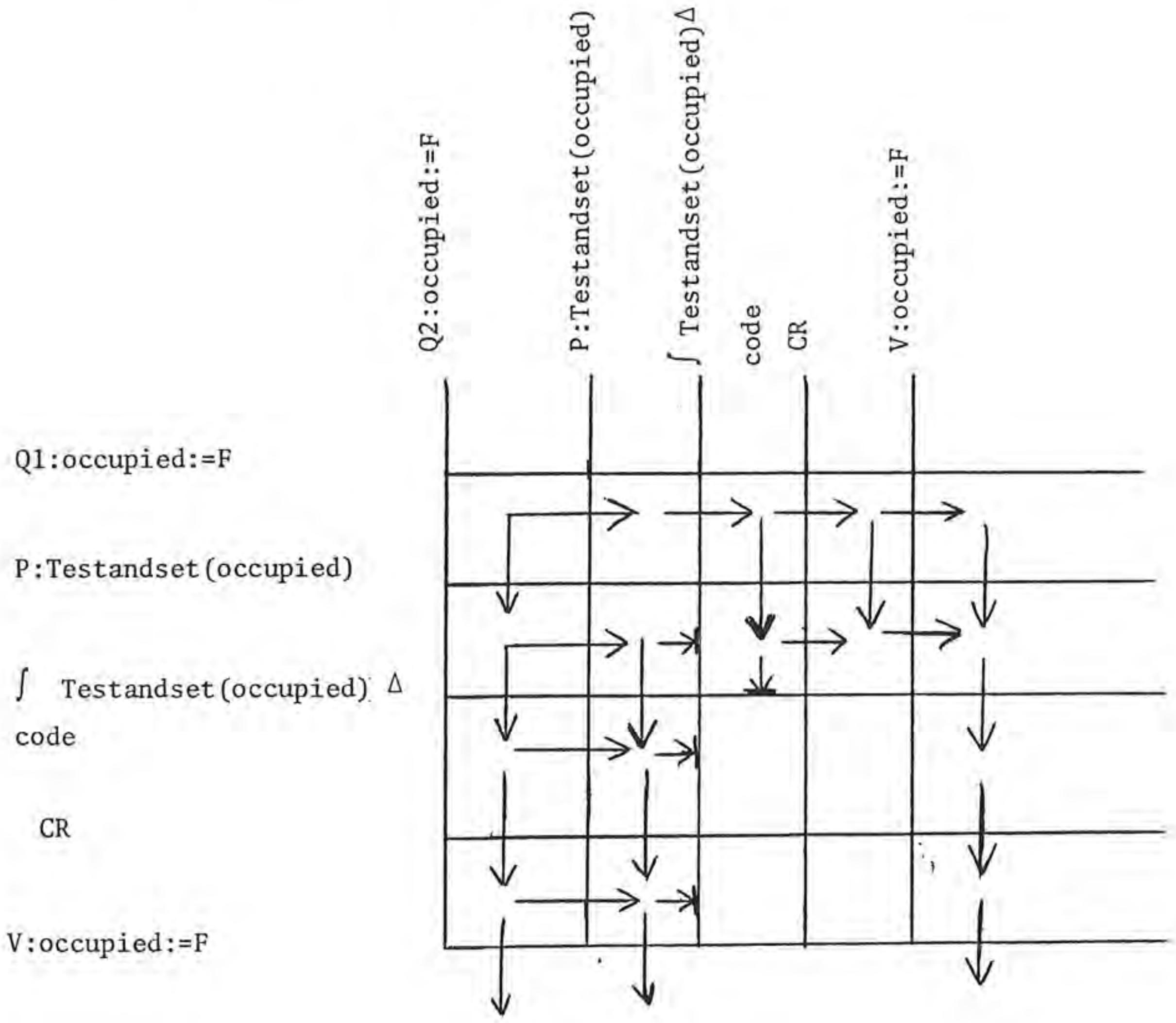
The testandset instruction "simultaneously" (indivisibly) tests a bit and sets a condition code, e.g. CODE. If the CODE is false, the Testandset changes the flag to true, and sets CODE to true if the flag was previously true; to false if the flag was previously false. In otherwords, CODE indicates the condition resulting from a test, while the flag is set to "1" or true.

A decrement instruction simulates Testandset by decrementing a word containing zero (thus setting it to (-1) and the condition code to Negative).

The semaphore for mutual exclusion given in reference [1], page 26, uses Testandset, a flag "occupied", and condition code CODE.

```
P:  Testandset [occupied]
    DO WHILE (CODE);
         Testandset [occupied],
    END;
    CR;
V:  occupied:=FALSE;
```

The process-state diagram for this concurrent program is given below. Note that every path leading to a potential indeterminacy is blocked. Starvation is possible because there exists paths through P1:CR that do not pass through P2:CR.



Device Monitors

Recent developments in structured programming languages have led to structuring concepts in operating systems. A form of "structured concurrency" is possible if a device monitor is employed.

A monitor is an abstract data type consisting of permanent data (the critical section data), access procedures (the routines needed by a cluster to define the data to a routine outside the monitor), and initialization code that "starts" the device monitor running.

The concept of a monitor is quite straight-forward: it is intended to encapsulate, or "virtualize" the critical resources of a computer system. The resources are "critical" due to sharing by more than one process running concurrently in the computer.

The monitor described below is derived from Wirth's Modula [3]. Its purpose is to guarantee mutual exclusion, blocking and waking-up of processes. We use the explanation of monitor given in reference [2]:

> "They are fences enclosing critical data. The variables declared in the monitor are the critical data and are permanent (retain their values between executions in the monitor). A monitor consists of a monitor header, declarations of local variables, initialization code, zero or more procedures, and zero or more entries."

The following monitor from Modula is used in a PDP-11 operating system to manage the console keyboard. The kb process runs from an interrupt vector at location 60B. The "readch" routine runs from outside invocations. The permanent data includes a buffer of 64 characters, and signals for synchronizing the two active routines in the device monitor.

```
device module keyboard

     define  readch;

     const   n = 64;

     var     in,out,n#:integer;

             nonempty,nonfull:signal;
             buf:array 1:n of char

   procedure readch (var ch: char);

     begin

        if  n#=o then wait (nonempty) end

        ch:= buf [out]; out:=(out mod n) +1;

        dec (n#): send (nonfull)

     end     readch;

     process kb[60B];

     begin

        loop

           if  n#=n then wait (nonfull) end

           buf [in]:=doio; in:=(in mod n) +1;

           inc (n#): send (nonempty);

        end;

     end   kb;

initial:begin

           in :=1; out:=1; n#:=o;

           kb:    (*start process *)

        end;

        end  keyboard
```

Characteristics of a "good" "keyboard" monitor:

1) No buffer overflow, underflow:

$$1 \leq in,\ out \leq n$$

2) No buffer overrun:

$$1 \leq out \leq in \leq n$$

Process kb is a perpetually running device reader while procedure "readch" is a callable routine. Since "readch" appears in a "define" statement, it is exported to other routines outside the keyboard module (monitor). Other processes may use "readch" to get data, but may <u>not</u> use process kb, or access permanent variables "in", "out", "n#", etc. directly.

Characteristics of a "good" monitor "keyboard" are demonstrated, as follows:

1) No buffer overflow, underflow:

$$1 \leq in,\ out \leq n$$

Since in:=1; out:=1; and modulo n calculations are performed in every increment of each variable, this characteristic holds, immediately:

$$1 \leq in{:=}(in\ \underline{mod}\ n)\ +1 \leq n$$

$$1 \leq (in\ \underline{mod}\ n)\ +1 \leq n$$

$$0 \leq (in\ \underline{mod}\ n) \leq n-1$$

and,

$$1 \leq out{:=}(out\ \underline{mod}\ n)\ +1 \leq n$$

thus,

$$0 \leq (out\ \underline{mod}\ n) \leq n-1$$

2) No buffer overrun:

$$(0 \leq n\# \leq n) \text{ implies } (1 \leq out \leq in \leq n)$$

This is proven by solving the equations for "kb" and "readch" as follows. First, we will shorten the resulting expressions by eliminating the executed code in "kb" and "readch" except for the code that deals with the values of interest, e.g. n;n#,in, and out. Also, the values of these variables are computed corresponding to their use by buf [x], where x=in, or x=out. Hence, we are concerned only with references to the critical data.

Secondly, the solution to this problem is given by reducing the processing to "kb" or "readch" activations. Whenever a "wait" is encountered we substitute an execution in its place. The substitutions are noted by the prefix "kb:" or "readch:", as done earlier for concurrent processes.

Given $0 \leq n\# \leq n$ show $1 \leq out \leq in \leq n$. Start with the execution of process kb.

$$\equiv \text{kb:}0 \leq n\# \leq n; \quad \frac{n\#=n \text{ ; wait (nonfull)}}{0 \leq n\# < n;\text{in}=(n\#+\text{out})\text{mod } n+1;\text{ send(nonempty)}}$$

The next step in substitution is to replace the "wait" and "send" with "readch".

$$\equiv \text{kb:}0 \leq n\# \leq n; \quad \frac{n\#=n;\text{readch:out}=1}{0 \leq n\# < n;\text{in}=(n\#+\text{out})\text{mod } n+1;\text{readch: } \dfrac{n\#=0;\text{wait(nonempty)}}{0 < n\# < n;\text{out}=(\text{in}-n\#)}}$$

Notice above how the expressions for "in" and "out" have been replaced by expressions representing the value obtained through n#. The n# counter is the only variable common to both processes, hence we use it to break the chained-recursive expressions for in and out. The n# counter counts up in "kb" and counts down in "readch". Therefore, "in" is equal to the number of inputs as recorded by n# <u>plus</u> the number of outputs as recorded by "out".

When n#=n, the buffer is full (in and n# have counted up to their maximum) and out=1. Conversely, when the buffer is empty, n#=0 (decremented to zero) and "in" begins over, again.

Finally, note that "out" is equal to the number of inputs, "in", minus the number of outputs as recorded in variable "n#".

The next step is to substitute "readch" for the remaining "wait" statement, and reduce the expression to a sequence of simple, single-assignment statements.

$$\equiv \mathrm{kb}{:}\,0 \le n\# \le n;\ \frac{n\#=n;\mathrm{readch}{:}\mathrm{out}=1;}{0 \le n\# < n;\mathrm{in}=(n\#+\mathrm{out})\bmod n+1;\mathrm{readch}{:}\ \dfrac{n\#=0;\mathrm{kb}{:}\mathrm{in}=1;}{0 < n\# < n;\mathrm{out}=(\mathrm{in}-n\#)}}$$

We began by trying to demonstrate the implication:

$$(0 \le n\# \le n) \text{ implies } (1 \le \mathrm{out} \le \mathrm{in} \le n).$$

The next step in the demonstration is to reduce each expression containing the variables in question to predicates. We can do this by combining the predicates as follows.

$$(0 \le n\# < n;\mathrm{in}=(n\#+\mathrm{out})\bmod n+1 \text{ implies } (\mathrm{out} \le \mathrm{in} \le n)$$

Forward substitution of "in" into the expression for out gives the final reduction:

$$\mathrm{out}=(n\#+\mathrm{out}-n\#) \text{ implies } \mathrm{out}=\mathrm{out}.$$

$$\mathrm{kb}{:}\,0 \le n\# \le n;\ \frac{n\#=n;\mathrm{readch}{:}\mathrm{out}=1}{0 \le n\# < n;\mathrm{out} \le \mathrm{in} \le n;\mathrm{readch}{:}\ \dfrac{n\#=0;\mathrm{kb}{:}\mathrm{in}=1}{0 < n\# < n;\mathrm{out}=\mathrm{out}}}$$

This final expression (all simple single-assignment statements and predicates) demonstrates the point. Hence, we have shown the keyboard monitor of Wirth's paper to be correct relative to the two characteristics stated, above.

CONCLUSIONS

The formal operator technique is useful for straight forward application to concurrent programs of unusual complexity. Few other formal methods exist for study of these small, but convoluted procedures. Thus, the methods of this paper hold promise for formal testing procedures especially applicable to operating system modules. Many problems exist, however.

1) The process-state matrix is detailed and error-prone. An automatic process-state matrix generator should be developed to analyze programs of reasonable complexity.

2) The method is basically a hand method. The transformations and their operators need to be formalized even further to allow automatic application. This may require cataloging the various transformations used in demonstrating a class of programs. Obvious differences exist between sequential and concurrent programs as shown in this two-part report.

3) A formal theory of test data selection still does not exist for the operators given here.

References

[1] Lewis, T. G.,
"An Operator Calculus For Computer Programs: Part I",
Oregon State University, Computer Science Department,
Tech. Report 78-1-5, 1978.

[2] Holt, R. C., Graham, G. S., Lazowska, E. D., Scott, M. A.,
Structured Concurrent Programming With Operating Systems
Applications, Addison-Wesley, 1978.

[3] Wirth, N.,
"Modula: A Language For Modular Multiprogramming",
Software Practices and Experience, 7, 1 (1977), pp. 3-35.