An Experiment in Code Verification

Frank Knowles
Gould, Inc.
Computer Systems Division - Urbana
1101 East University Avenue
Urbana, ILL 61801
U. S. A.

GOULD
Electronics

Nov 15, 1985

AN EXPERIMENT IN CODE-LEVEL VERIFICATION

This talk describes a research effort to investigate theorem proving requirements for code level verification.

The progamming language used is the sequential subset of Toronto Concurrent Euclid, developed at the University of Toronto, Canada. The Verification Condition Generator was written by Phillip Matthews as a Master's Degree project at the University of Toronto. The run-time-specific proof rules are based on those in a Doctor's thesis by W. David Elliot, done also at Toronto. The theorm prover was written by Dan Putnam as part of Compion's (now Gould Computer Systems - Urbana) specification system, VERUS.

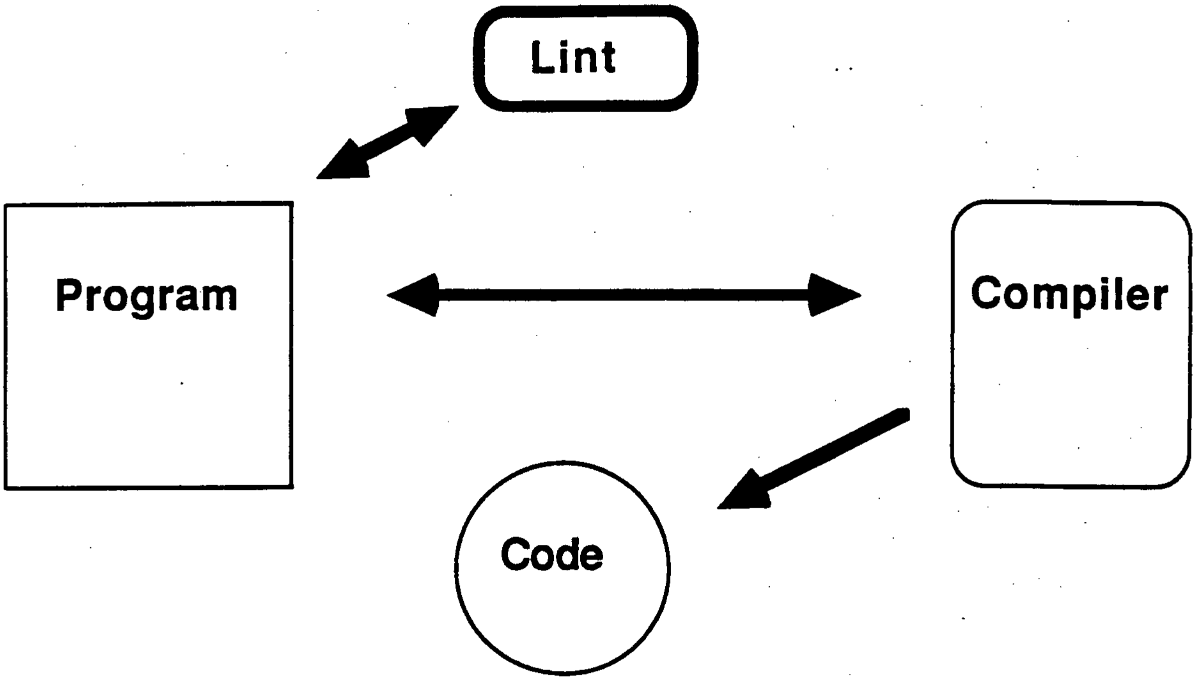
The immediate goal is to characterize the verification conditions that can be proven without significant interaction between the user and the theorem prover.

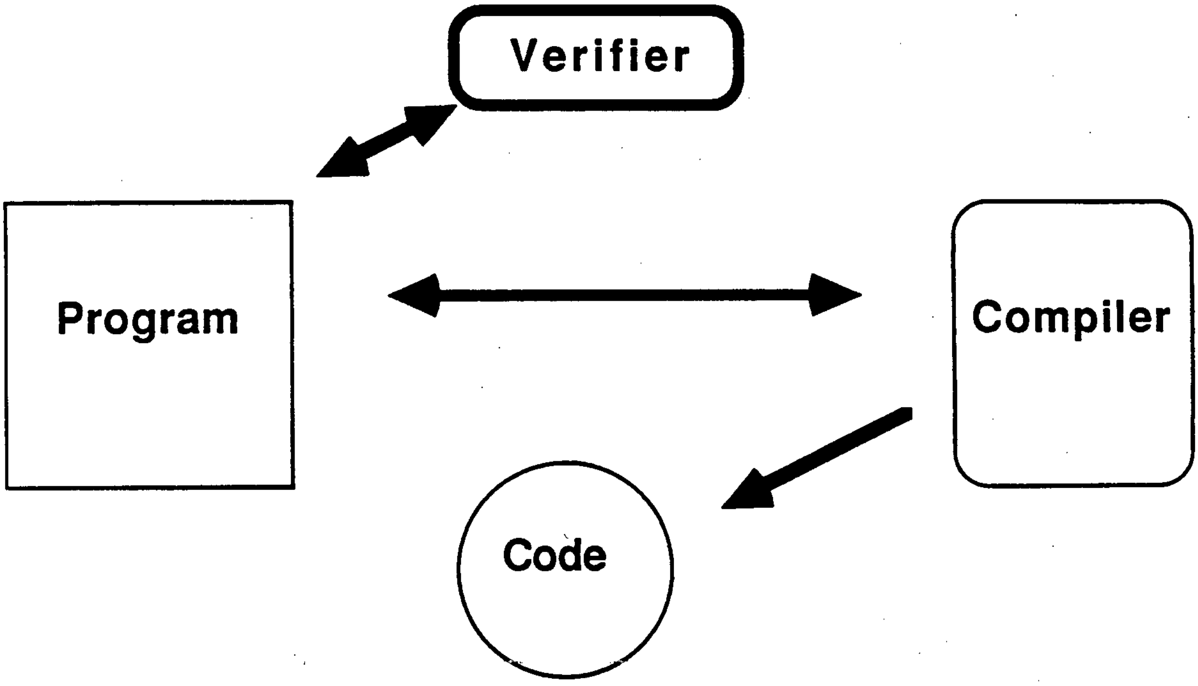
Two hypotheses are being investigated. The first is that verification conditions showing merely the absence of run-time errors fall into this category. The second is that Putnam's prover is especially good at shallow proofs of the type that will be encountered.

New proof rules will be presented that combine a modified form of the "forward" proof rules of the verification condition generator with Elliot's rules for detection of run-time errors.

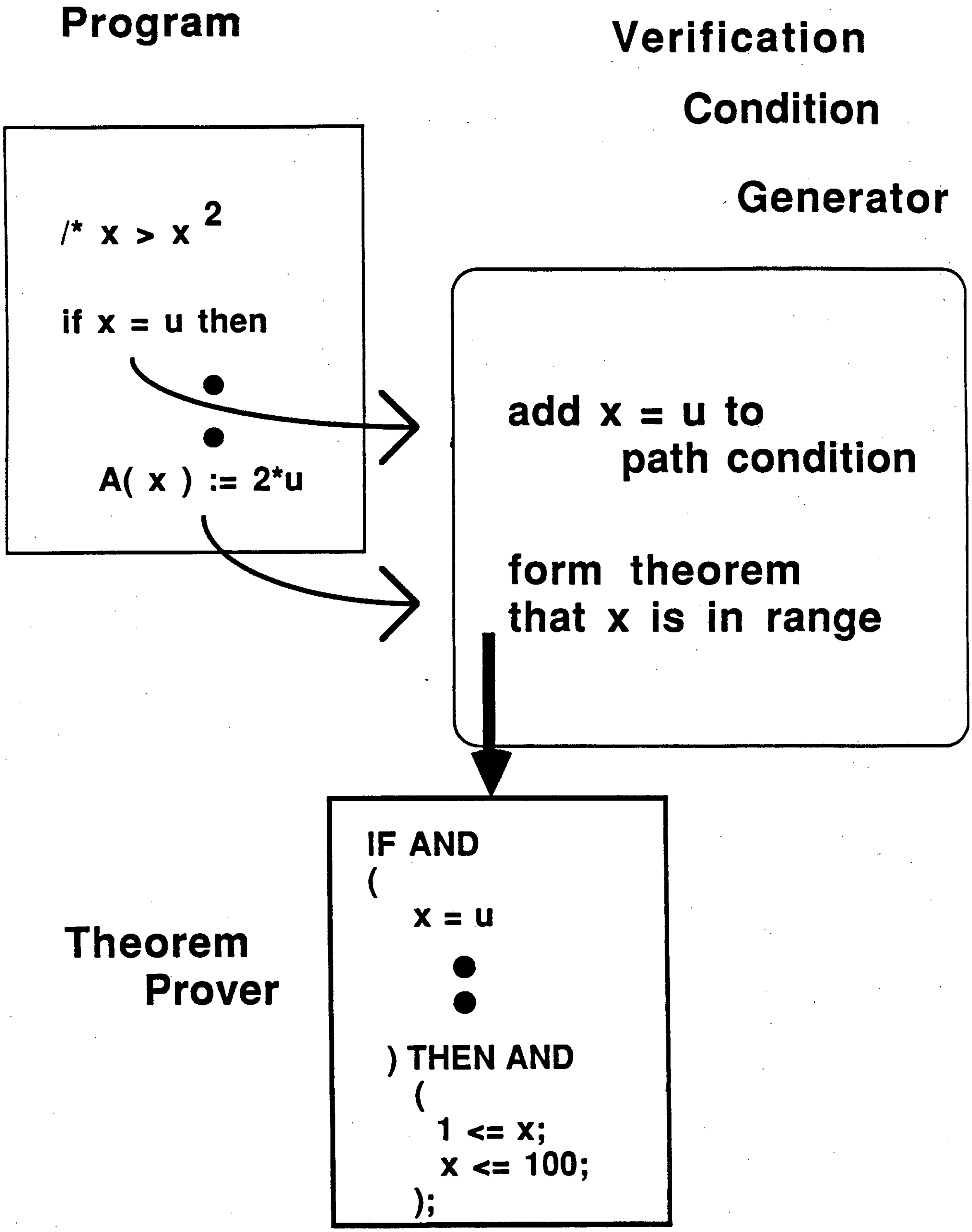
The verification condition generator has not been modified to work with the new rules, so verification conditions being investigated must be manually generated and given to the theorem prover in the form of a VERUS proof outline.

Examples of programs, verification conditions, proofs, and problems will be presented.


Lint
Program
Compiler
Code


Verifier
Program
Compiler
Code

```
/*   move i up till we find a value too large ...     */


          loop
              exit  when  A(  i  )  >=  r

              i  :=  i  +  1



/* FOR j IF AND ( 1 <= j; j <= i ) THEN A( J ) < r;  */


          loop
              exit  when  A(  i  )  >=  r

              i  :=  i  +  1
```



Program

/* $x > x^2$

if x = u then

●

●

A(x) := 2*u

Verification Condition Generator

add x = u to path condition

form theorem that x is in range

Theorem Prover

```
IF AND
(
   x = u
    ●
    ●
 ) THEN AND
   (
    1 <= x;
    x <= 100;
   );
```

```
 { swap.e       exchange values of two variables }

 procedure Swap ( var i : ShortInt, var j : ShortInt ) =
 { entry: i and j are defined and refer to distinct places in memory }
 { exit: i = INITIAL (j) and j = INITIAL (i) }

 begin

        var w : ShortInt;

        w := i
        i := j
        j := w

end Swap
```

One way to simulate the program:

```
  Path Condition                       Renaming Table
----------------------          ---------------------
                                 i        j       w
                                ---------------------
INITIAL( i ) = i_1               1
INITIAL( j ) = j_1                        1
w_1 = i_1                                         1
i_2 = j_1                        2
j_2 = w_1                                 2
```

Resulting theorem for the prover:

```
PROVE
  IF AND
  {
    INITIAL( i ) = i_1
    INITIAL( j ) = j_1
    w_1 = i_1
    i_2 = j_1
    j_2 = w_1
  }
  THEN AND
{
  i_2 = INITIAL( j );
  j_2 = INITIAL( i );
};
```

```
 { swap.e        exchange values of two variables }

 procedure Swap ( var i : ShortInt, var j : ShortInt ) =
 { entry: i and j are defined and refer to distinct places in memory }
 { exit: i = INITIAL (j) and j = INITIAL (i) }

 begin

         var w : ShortInt;

         w := i
         i := j
         j := w

end Swap

A better way to simulate the program:

    Path Condition              Symbol Table Stack
---------------------          --------------------

(VCG tracks defineness.)       i         j       w
                               --------------------
(VCG/compiler enforces         i_1       j_1
no aliasing.)                                    i_1
                               j_1
                                         i_1

Easier theorem for the prover:

PROVE AND
{
  i_1 = i_1;
  j_1 = j_1;
};
```

```
{ fast linear search }

procedure Search (      key: ShortInt,
                   var A : array 1..10 of ShortInt,
                   var i: ShortInt
                 ) =

post ( A( i ) = key )

begin
        A(10) := key    { don't care about the very last place }

        i := 1

        loop
                exit when A(i) = key

                i := i + 1
        end loop
end Search
```

```
{ fast linear search }

procedure Search (      key: ShortInt,
                    var A : array 1..10 of ShortInt,
                    var i: ShortInt
                  ) =
post ( A( i ) = key )

begin
        A(10) := key     { don't care about the very last place }

        i := 1

        loop
                invariant ( 1 <= i and i <= 10 )
                measure ( M := 10 - i )

                exit when A(i) = key

                i := i + 1
        end loop
end Search
```

Show that i in range after an iteration:

```
                                          Symbol Table

                                           i
                                         --------
                                          i_1
                                          i_1 + 1
```

```
PROVE
  IF AND
  {
    A( 10 ) = key;          $ array assignments in path condition!
    1 <= i_1;               $ old invariant
    i_1 <= 10;
    NOT A( i_1 ) = key;  $ false exit condition
  }
  THEN AND
  {
    1 <= i_1 + 1            $ new invariant
    i_1 + 1 <= 10;
  };
```

```
{ fast linear search }

procedure Search (      key: ShortInt,
                   var A : array 1..10 of ShortInt,
                   var i: ShortInt
                 ) =
post ( A( i ) = key )

begin
        A(10) := key    { don't care about the very last place }

        i := 1

        loop
                invariant ( 1 <= i and i <= 10 )
                measure ( M := 10 - i )

                exit when A(i) = key

                i := i + 1
        end loop
end Search

Show that i in range after an iteration:

                                              Symbol Table

                                               i
                                             --------
                                              i_1
                                              i_1 + 1

Show loop termination

PROVE
  IF AND
  {
    A( 10 ) = key;        $ array assignments in path condition!
    1 <= i_1;             $ old invariant
    i_1 <= 10;
    NOT A( i_1 ) = key;   $ false exit condition
    1 <= i_1 + 1          $ new invariant
    i_1 + 1 <= 10;
    NOT A( i_1 + 1 ) = key;       $ new exit condition false
  }
  THEN AND
  {
    10 - ( i_1 + 1 ) >= 0;        $ naw measure in range
    10 - ( i_1 + 1 ) < 10 - i_1;          $ new < old
  };
```

Proof of a Program: FIND

C. A. R. Hoare
Queen's University, Belfast, Ireland*

A proof is given of the correctness of the algorithm "Find." First, an informal description is given of the purpose of the program and the method used. A systematic technique is described for constucting the program proof during the process of coding it, in such a way as to prevent the intrusion of logical errors. The proof of termination is treated as a separate exercise. Finally, some conclusions relating to general programming methodology are drawn.

KEY WORDS AND PHRASES: proofs of programs, programming methodology, program documentation, program correctness, theory of programming
CR CATEGORIES: 4.0, 4.22, 5.21, 5.23, 5.24

1. Introduction

In a number of papers [1, 2, 3] the desirability of proving the correctness of programs has been suggested and this has been illustrated by proofs of simple example programs. In this paper the construction of the proof of a useful, efficient, and nontrivial program, using a method based on invariants, is shown. It is suggested that if a proof is constructed as part of the coding process for an algorithm, it is hardly more laborious than the traditional practice of program testing.

2. The Program "Find"

The purpose of the program Find [4] is to find that element of an array $A[1:N]$ whose value is fth in order of magnitude; and to rearrange the array in such a way that this element is placed in $A[f]$; and furthermore, all elements with subscripts lower than f have lesser values, and all elements with subscripts greater than f have greater values. Thus on completion of the program, the following relationship will hold:

$$A[1], A[2], \cdots, A[f-1] \leq A[f] \leq A[f+1], \cdots, A[N]$$

This relation is abbreviated as Found.

One method of achieving the desired effect would be to sort the whole array. If the array is small, this would be a good method; but if the array is large, the time taken to sort it will also be large. The Find program is designed to take advantage of the weaker requirements to save much of the time which would be involved in a full sort.

The usefulness of the Find program arises from its application to the problem of finding the median or other quantiles of a set of observations stored in a computer array. For example, if N is odd and f is set to $(N+1)/2$, the effect of the Find program will be to place an observation with value equal to the median in $A[f]$. Similarly the first quartile may be found by setting f to $(N+1)/4$, and so on.

The method used is based on the principle that the desired effect of Find is to move lower valued elements of the array to one end—the "left-hand" end—and higher valued elements of the array to the other end—the "right-hand" end. (See Table I(a)). This suggests that the array be scanned, starting at the left-hand end and moving rightward. Any element encountered which is small will remain where it is, but any element which is large should be moved up to the right-hand end of the array, in exchange for a small one. In order to find such a small element, a separate scan is made, starting at the right-hand end and moving leftward. In this scan, any large element encountered remains where it is; the first small element encountered is moved down to the left-hand end in exchange for the large element already encountered in the rightward scan. Then both scans can be resumed until the next exchange is necessary. The process is repeated until the scans meet somewhere in the middle of the array. It is then known that all elements to the left of this meeting point will be small, and all elements to the right will be large. When this condition holds, we will say that the array is *split at* the given point into two parts (see Table I(b)).

The reasoning of the previous paragraph assumes that there is some means of distinguishing small elements from large ones. Since we are interested only in their comparative values, it is sufficient to select the value of some arbitrary element before either of the scans starts; any element with lower value than the selected element is counted as small, and any element with higher value is counted as large. The fact that the discriminating value is arbitrary means that the place where the two scans will meet is also arbitrary; but it does not affect the fact that the array will be split at the meeting point, wherever that may be.

Now consider the question on which side of the split the fth element in order of value is to be found. If the split is to the right of $A[f]$, then the desired element must of necessity be to the left of the split, and all elements to the right of the split will be greater than it. In this case, all elements to the right of the split can be ignored in any future processing, since they are already in their proper

---

* Department of Computer Science

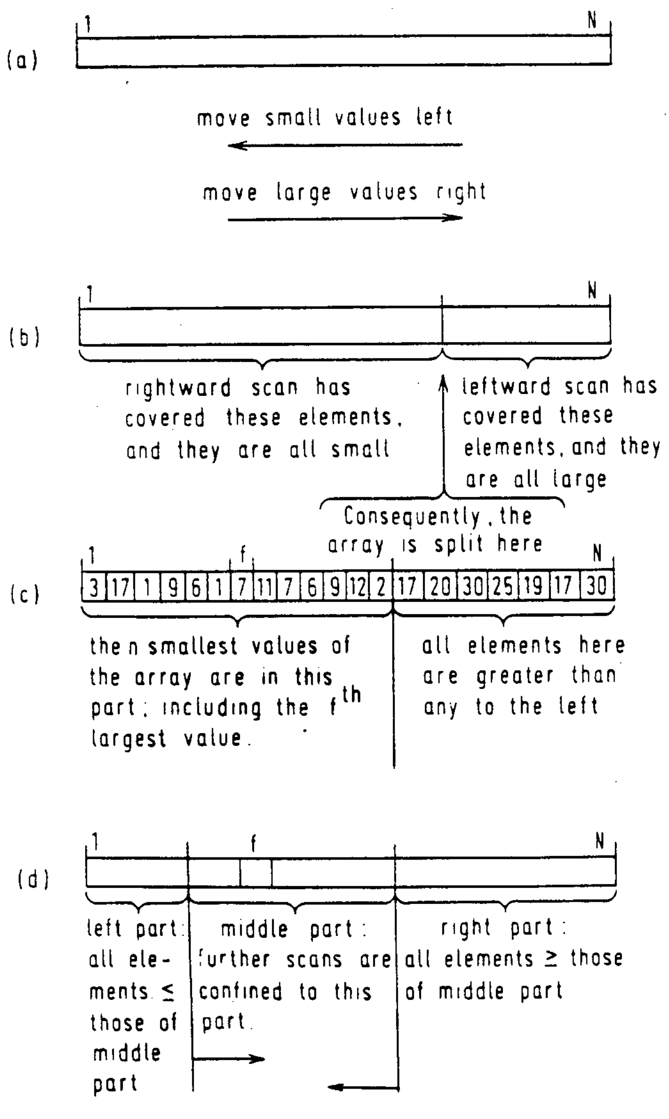
place, namely to the right of $A[f]$ (see Table I(c)). Similarly, if the split is to the left of $A[f]$, the element to be found must be to the right of the split, and all elements to the left of the split must be equal or less than it; furthermore, these elements can be ignored in future processing.

In either case, the program proceeds by repeating the rightward and leftward scans, but this time one of the scans will start at the split rather than at the beginning of the array. When the two scans meet again, it will be known that there is a second split in the array, this time perhaps on the other side of $A[f]$. Thus again, we may proceed with the rightward and leftward scans, but we start the rightward scan at the split on the left of $A[f]$ and the leftward scan at the split on the right, thus confining attention only to that part of the array that lies between the two splits; this will be known as the *middle part* of the array (see Table I(d)).

When the third scan is complete, the middle part of the array will be split again into two parts. We take the new middle part as that part which contains $A[f]$ and repeat the double scan on this new middle part. The process is repeated until the middle part consists of only one element, namely $A[f]$. This element will now be equal to or greater than all elements to the left and equal to or less than all elements to the right; and thus the desired result of Find will be accomplished

This has been an informal description of the method used by the program Find. Diagrams have been used to convey an understanding of how and why the method works, and they serve as an intuitive proof of its correctness. However, the method is described only in general terms, leaving many details undecided; and accordingly, the intuitive proof is far from watertight. In the next section, the details of the method will be filled in during the process of coding it in a formal programming language; and simultaneously, the details of the proof will be formalized in traditional logical notation. The end product of this activity will be a program suitable for computer execution, together with a proof of its correctness. The reader who checks the validity of the proof will thereby convince himself that the program requires no testing.

TABLE I



3. Coding and Proof Construction

The coding and proof construction may be split into several stages, each stage dealing with greater detail than the previous one. Furthermore, each stage may be systematically analyzed as a series of steps.

3.1. STAGE 1: PROBLEM DEFINITION

The first stage in coding and proof construction is to obtain a rigorous formulation of what is to be accomplished and what may be assumed to begin with. In this case we may assume

(a) The subscript bounds of A are 1 and N.

(b) $1 \leq f \leq N$.

The required result is:

$$\forall p, q(1 \leq p \leq f \leq q \leq N \supset A[p] \leq A[f] \leq A[q]) \qquad \text{[Found]}$$

3.2. STAGE 2: THE GENERAL METHOD

(1) The first step in each stage is to decide what variables will be required to hold intermediate results of the program. In the case of Find, it will be necessary to know at all times the extent of the middle part, which is currently being scanned. This indicates the introduction of variables m and n to point to the first element $A[m]$ and the last element $A[n]$ of the middle part.

(2) The second step is to attempt to describe more fo

FIND--WITHOUT COMMENTS

```
procedure Find ( f : ShortInt, var A: array 1..100 of UnsignedInt ) =

begin
        var i : ShortInt
        var j : ShortInt
        var m : ShortInt
        var n : ShortInt
        var r : UnsignedInt
        var w : UnsignedInt

        m := 1
        n := 100

        loop
                exit when ( m >= n )
                r := A(f)
                i := m
                j := n

                loop
                        exit when ( i > j )

                        loop
                                exit when ( A( i ) >= r )
                                i := i + 1
                        end loop

                        loop
                                exit when ( A( j ) <= r )
                                j := j - 1
                        end loop

                        if i <= j then
                                w := A( i )
                                A( i ) := A( j )
                                A( j ) := w
                                i := i + 1; j := j - 1
                        end if
                end loop

                if f <= j then n := j
                elseif f >= i then m := i
                else exit
                end if
        end loop
end Find
```

```
{ C. A. R. Hoare,               FIND            CACM Vol 14, No 7, (Jan 1971)   }

procedure Find ( f : ShortInt, var A: array 1..100 of UnsignedInt ) =
pre ( 1 <= f and f <= 100 ) { f, and all elements of A are defined             }
begin
        var i : ShortInt
        var j : ShortInt
        var m : ShortInt
        var n : ShortInt
        var r : UnsignedInt
        var w : UnsignedInt

        m := 1
        n := 100

        loop  { m_n     modifies: r, i, j, w, A, n, m                           }
{       invariant ( AND ( 1 <= m; m <= f; f <= n; n <= 100;) )                  }
        measure ( Meas1 := n - m )
                exit when ( m >= n )
                r := A(f)
                i := m
                j := n

                loop  { i_j     modifies: i, j, w, A                            }
{               invariant ( AND ( m <= i; j <= n; IF i <= j
                                                  THEN AND
                                                  (
                              OR ( AND ( i <= f; r <= A( f ); );
                                   AND ( i <= j + 1; r <= A( j + 1 ); ););
                              OR ( AND ( f <= j; r >= A( f ); );
                                   AND ( i - 1 <= j; r >= A( i - 1 ); ););
                                                  );
                                 ); )
                directive ( PROVE AND ( NEW( i ) > OLD( i ); NEW( j ) < OLD( j ); ) ) }
                measure ( Meas2 := j - i )
                        exit when ( i > j )

                        loop  { less_r  modifies: i                             }
{                       invariant ( OR ( AND ( i <= f; r <= A( f ); );
                                         AND ( i <= [i_j](j) + 1;
                                               r <= A( [i_j](j) + 1 ); ); ); )
                        directive ( PROVE NEW( i ) >= OLD( i ) ) }
                        measure ( Meas3 := n - i )
                                exit when ( A( i ) >= r )
                                i := i + 1
                        end loop  { less_r                                      }

                        loop  { greater_r       modifies: j                     }
{                       invariant ( OR ( AND ( f <= j; r >= A( f ); );
                                         AND ( [i_j](i) - 1 <= j;
                                               r >= A( [i_j](i) - 1 ); ); ); )
                        directive ( PROVE ( NEW( j ) <= OLD( j ) ) ) }
                        measure ( Meas4 := j - m )
                                exit when ( A( j ) <= r )
                                j := j - 1
                        end loop  { greater_r                                   }


                        if i <= j then
                                w := A( i ); A( i ) := A( j ); A( j ) := w

{                       directive ( PROVE AND ( A( i ) <= r; A( j ) >= r; )
                                     USING SWAP( A ) ) ) }

                                 i := i + 1; j := j - 1
                         end if                                                 }
                 end loop  { i_j

                 if f <= j then n := j
                 elseif f >= i then m := i
                 else exit
                 end if                                                         }
         end loop  { m_n
 end Find
```

SWAP LEMMA

```
$ A PROVE directive with USING SWAP( A ) will cause this array
$ lemma to be accepted as an axiom ONLY for the proof indicated.

$ The array lemma is the result of scanning the path condition
$ for modifications of A_3 and finding

$   A_4 = A_3[i_3|A_3(j_3)]
$   A_5 = A_4[j_3|A_3(i_3)]

VAR A_5_i : INTEGER;

DECLARE A_4( INTEGER ) : INTEGER;
DECLARE A_5( INTEGER ) : INTEGER;

DEFINE A_5_Array_Lemma : BOOLEAN
BY
 FOR A_5_i AND
  {
   A_5( j_3 ) = A_3( i_3 );
   A_5( i_3 ) = A_3( j_3 );
   IF AND
   {
    NOT A_5_i = j_3;
    NOT A_5_i = i_3;
   }
   THEN A_5( A_5_i ) = A_3( A_5_i );
  };
```

path	execution point	path condition stack
1	enter loop m_n	(1) f_Init
	top of m_n after iteration	(2) m_n invariant
2	enter loop i_j	(3) false m_n exit condition
	top of i_j after iteration	(4) i_j invariant
3	enter loop less_r	(5) false i_j exit condition
	top of less_r after iteration	(6) less_r invariant
4	execute loop less_r	(7) false less_r exit condition
5	enter loop greater_r	elim (7), (8) i_3 >= i_2, (9) less_r exit condition
	top of greater_r after iteration	(10) greater_r invariant
6	execute loop greater_r	(11) false greater_r exit condition
7	execute loop i_j: if branch	elim (11), (12) j_3 <= j_2, (13) greater_r exit condition, (14) i_3 <= j_3
8	execute loop i_j: else branch	elim (14), (15) NOT i_3 <= j_3
9	execute loop m_n: if branch	elim (5), elim (15), (16) i_j exit condition, (17) f_1 <= j_2
10	execute loop m_n: elseif branch	elim (17), (18) NOT f_1 <= j_2, (19) f_1 >= i_2
11	execute m_n loop: exit branch	*--------nothing to prove--------*
12	exit m_n loop	*--------nothing to prove--------*
13	exit Find	*--------nothing to prove--------*

FINDING A PATH CONDITION FOR A THEOREM

path	execution point	path condition stack
1	enter loop m_n	(1) f_Init
	top of m_n after iteration	(2) m_n invariant
2	enter loop i_j	(3) false m_n exit condition
	top of i_j after iteration	(4) i_j invariant
3	enter loop less_r	(5) false i_j exit condition
	top of less_r after iteration	(6) less_r invariant
4	execute loop less_r	(7) false less_r exit condition
5	enter loop greater_r	elim (7), (8) i_3 >= i_2, (9) less_r exit condition

Directions for finding the path condition for a theorem:

The numbers on the left (1 - 5 above) indicate execution points in the program where a theorem to validate an invariant and measure must be proven. There are two theorems for each loop. For instance, 3 and 4 are for loop less_r. The first shows that the invariant is true upon initial entry; the second shows that same after an arbitrary iteration.

For each number indicating an execution point, go to the right in the table and concatenate the conditions in that row and all rows higher up. If you see a statement like "elim(7)", that means that statement 7 is NOT to be included in the path condition. The path structure is simple enough in FIND that this technique will work.
up is not to be included.

		Symbol Table Stack							
path	execution point	m	n	f	A	r	i	j	w
1	enter m_n loop	1	100	f_1	A_1				
	top of m_n after an iteration	m_1	n_1		A_2	r_1	i_1	j_1	w_1
2	enter i_j loop					A_2(f_1)	m_1	n_1	
	top of i_j after an iteration				A_3		i_2	j_2	w_2
3	enter loop less_r								
	top of less_r after iteration						i_3		
4	execute loop less_r						i_3+1		
5	enter loop greater_r						i_3		
	top of greater_r after iteration							j_3	
6	execute loop greater_r							j_3-1	
7	execute i_j loop: if branch				A_3 A_4 = A_3[i_3 | A_3(j_3)] A_5 = A_4[j_3 | A_3(i_3)]		i_3 i_3+1	j_3 j_3-1	w_2 A_3(i_3)
8	execute loop i_j: else branch				A_3		i_3	j_3	w_2
9	execute loop m_n: if branch		n_1 j_2						
10	execute loop m_n: elseif branch	m_1 i_2	n_1						
11	execute loop m_n: exit branch				*----------nothing to prove------------*				
12	exit loop m_n				*----------nothing to prove------------*				
13	exit Find				*----------nothing to prove------------*				

FINDING THE CORRECT CONSTANT SUBSTITUTION FOR PATH EXPRESSION

		Symbol Table Stack							
path	execution point	m	n	f	A	r	i	j	w
1	enter m_n loop	1	100	f_1	A_1				
	top of m_n after an iteration	m_1	r_1		A_2	r_1	i_1	j_1	w_1
2	enter i_j loop					A_2(f_1)	m_1	n_1	
	top of i_j after an iteration				A_3		i_2	j_2	w_2
3	enter loop less_r								
	top of less_r after iteration						i_3		
4	execute loop less_r						i_3+1		
5	enter loop greater_r						i_3		

Directions for finding the correct constant substitution for a path expression:

The numbers on the left (1 - 5 above) indicate execution points in the program where a theorem to validate an invariant and measure must be proven. There are two theorems for each loop. For instance, 3 and 4 are for loop less_r. The first shows that the invariant is true upon initial entry; the second shows that same after an arbitrary iteration.

For each number indicating an execution point, the entry in the row under the variable column is the value of that variable at that point in the program. For example, after an arbitrary iteration of the i_j loop, the VCG will assume that the value of i is i_2, a value about which nothing is known. Previous facts concerning i_1 are not affected.

TO FORM A THEOREM to prove a particular invariant, first get the path condition from the path chart, then use this chart to substitute constant values for each subexpression of the path condition, as well as the conclusion of the theorem.

```
                         VC7 THEOREM--DECLARATIONS

$ find.e

$ PATH #7 :  execute i_j loop, if branch

$ initial assumptions regarding f

VAR f : INTEGER;
CONST f_1 : INTEGER;     $ initial value of f

DEFINE f_Init : BOOLEAN
BY
  AND
  {
    1 <= f_1;
    f_1 <= 100;
  };

AXIOM f_Init;

$ initial assumptions concerning A

DECLARE A( INTEGER ) : INTEGER;
DECLARE A_1( INTEGER ) : INTEGER;        $ initial value

$ declare local variables

VAR i, j, m, n, r, w : INTEGER;

$ values for variables modified in m_n loop

CONST m_1, n_1, r_1, i_1, j_1, w_1 : INTEGER;
DECLARE A_2( INTEGER ) : INTEGER;

$ values for variables modified in i_j loop

CONST i_2, j_2, w_2 : INTEGER;
DECLARE A_3( INTEGER ) : INTEGER;

$ values for variables modified in less_r loop

CONST i_3 : INTEGER;

$ new values for variables modified in greater_r loop

CONST j_3 : INTEGER;

$ A PROVE directive with USING SWAP( A ) will cause this array
$ lemma to be accepted as an axiom ONLY for the proof indicated.

$ The array lemma is the result of scanning the path condition
$ for modifications of A_3 and finding

$       A_4 = A_3[i_3|A_3(j_3)]
$       A_5 = A_4[j_3|A_3(i_3)]


VAR A_5_i : INTEGER;

DECLARE A_4( INTEGER ) : INTEGER;
DECLARE A_5( INTEGER ) : INTEGER;

DEFINE A_5_Array_Lemma : BOOLEAN
BY
  FOR A_5_i AND
    {
      A_5( j_3 ) = A_3( i_3 );
      A_5( i_3 ) = A_3( j_3 );
      IF AND
      {
        NOT A_5_i = j_3;
        NOT A_5_i = i_3;
      }
      THEN A_5( A_5_i ) = A_3( A_5_i );
    };
```

```
                        VC7 THEOREM--IF PART 1

PROVE                   $ prove new invariant true
  IF AND
  {
    f_Init;             $ initial path condition
    1 <= m_1;           $ m_n loop invariant
    m_1 <= f_1;
    f_1 <= n_1;
    n_1 <= 100;
    NOT m_1 >= n_1;     $ false m_n loop exit condition
    AND                 $ i_j loop invariant
    {
      m_1 <= i_2;
      j_2 <= n_1;
      IF i_2 <= j_2
      THEN AND
      {
        OR
        {
          AND
          {
            i_2 <= f_1;
            A_2( f_1 ) <= A_3( f_1 );
          };
          AND
          {
            i_2 <= j_2 + 1;
            A_2( f_1 ) <= A_3( j_2 + 1 );
          };
        };
        OR
        {
          AND
          {
            f_1 <= j_2;
            A_2( f_1 ) >= A_3( f_1 );
          };
          AND
          {
            i_2 - 1 <= j_2;
            A_2( f_1 ) >= A_3( i_2 - 1 );
          };
        };
      };
    };
    NOT i_2 > j_2;      $ false i_j loop exit condition
```

VC7 THEOREM--IF PART 2

```
                        $ less_r loop invariant
    OR
    {
      AND
      {
        i_3 <= f_1;
        A_2( f_1 ) <= A_3( f_1 );
      };
      AND
      {
        i_3 <= j_2 + 1;                 $ j_2 for [i_j](j)
        A_2( f_1 ) <= A_3( j_2 + 1 );   $ j_2 for [i_j](j)
      };
    };
    A_3( i_3 ) >= A_2( f_1 );    $ less_r loop exit condition
    i_3 >= i_2;                  $ NEW( i ) >= OLD( i )
    OR                           $ greater_r loop invariant
    {
      AND
        {
          f_1 <= j_3;
          A_2( f_1 ) >= A_3( f_1 );
        };
      AND
        {
          i_2 - 1 <= j_3;                $ i_2 for [i_j](i)
          A_2( f_1 ) >= A_3( i_2 - 1 ); $ i_2 for [i_j](i)
        };
    };
    A_3( j_3) <= A_2( f_1 );     $ greater_r loop exit condition
    j_3 <= j_2;                  $ NEW( j ) <= OLD( j )
    i_3 <= j_3;                  $ if branch condition

$ result of directive ( PROVE ( ..next two statements.. ) USING SWAP( A )

    A_5( i_3 ) <= A_2( f_1 );
    A_5( j_3 ) >= A_2( f_1 );
```

VC7 THEOREM--THEN PART

```
  THEN AND                      $ new i_j loop invariant
    {
      m_1 <= i_3 + 1;
      j_3 - 1 <= n_1;
      IF i_3 + 1 <= j_3 - 1
      THEN AND
      {
        OR
        {
          AND
          {
            i_3 + 1 <= f_1;
            A_2( f_1 ) <= A_5( f_1 );
          };
          AND
          {
            i_3 + 1 <= ( j_3 - 1 ) + 1;
            A_2( f_1 ) <= A_5( ( j_3 - 1 ) + 1 );
          };
        };
        OR
        {
          AND
          {
            f_1 <= j_3 - 1;
            A_2( f_1 ) >= A_5( f_1 );
          };
          AND
          {
            ( i_3 + 1 ) - 1  <= j_3 - 1;
            A_2( f_1 ) >= A_5( ( i_3 + 1 ) - 1 );
          };
        };
      };
    };
{};
```

References

1. James R. Cordy, Richard C. Holt, "Specification of Concurrent Euclid," Technical Report CSRG-133, Computer Systems Research Group, University of Toronto, Toronto, CANADA (Aug 1981)

2. W. David Elliott, "On Proving the Absence of Execution Errors," Technical Report CSRG-141, Computer Systems Research Group, University of Toronto, Toronto, Ontario, CANADA (Mar 1982)

3. Steven M. German, "Verifying the Absence of Common Runtime Errors in Computer Programs," Report No. STAN-CS-81-866, Department of Computer Science, Stanford University, Stanford, CA 94305, USA

4. Philip Arnold Matthews, "Concurrent Euclid: Proof Rules and a VCG," Master of Science Thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, CANADA (Jun 1982)

5. Philip A. Matthews, Richard C. Holt, "A Guide to the Concurrent Euclid XVCG: An Experimental Verification Condition Generator," Technical Note CSRG-27, Computer Systems Research Group, University of Toronto, Toronto, Ontario, CANADA (Jun 1982)

6. "VERUS Language Manual," Release 3.0, Gould Inc., Computer Systems Division, 1101 East University, Urbana, IL, 61801 USA (Jun 1985)

7. "VERUS State Machine Specification Checker Guide," Release 3.0, Gould Inc., Computer Systems Division, 1101 East University, Urbana, IL, 61801 USA (Jun 1985)

8. "VERUS State Machine Specification Guide," Release 3.0, Gould Inc., Computer Systems Division, 1101 East University, Urbana, IL, 61801 USA (Jun 1985)

9. "VERUS User's Guide," Release 3.0, Gould Inc., Computer Systems Division, 1101 East University, Urbana, IL, 61801 USA (Jun 1985)