An Experiment in Code Verification

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

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 programming 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 Elliott, done also at Toronto. The theorem prover was written by Dan Putnam as part of Compion's (now Gould Computer Systems - Urbana) specification system, VERUS.

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.
/* 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:

<table>
<thead>
<tr>
<th>Path Condition</th>
<th>Renaming Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL( i ) = i_1</td>
<td>i  1  w  1</td>
</tr>
<tr>
<td>INITIAL( j ) = j_1</td>
<td>j  1  w  1</td>
</tr>
<tr>
<td>w_1 = i_1</td>
<td>1  1  1</td>
</tr>
<tr>
<td>i_2 = j_1</td>
<td>2  1  1</td>
</tr>
<tr>
<td>j_2 = w_1</td>
<td>2  2  2</td>
</tr>
</tbody>
</table>

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( i ) ;
 j_2 = INITIAL( i ) ;
} ;
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
------------
(VCG tracks definiteness.)
(VCG/compiler enforces
nc aliasing.)

Symbol Table Stack
-------------------
i  j  w
i_1  j_1  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(1) = 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

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>i-1</td>
</tr>
<tr>
<td>i-1 + 1</td>
</tr>
</tbody>
</table>

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 ) =
begin
  A(10) := key  { don't care about the very last place }
  i := 1
  loop
    invariant ( i <= 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

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>i-1</td>
</tr>
<tr>
<td>i-1+1</td>
</tr>
</tbody>
</table>

Show loop termination

prove

if and

{ A(10) = key; $ array assignments in path condition!
  i <= i-1; $ old invariant
  i-1 <= 10;
  not A( i-1 ) = key; $ false exit condition
  l <= 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; $ new 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 constructing 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

PROOF OF A PROGRAM: FIND

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 \(f\)th 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 lower values, and all elements with subscripts greater than \(f\) have greater values. Thus on completion of the program, the following relationship will hold:

\[
\]

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 \(f\)th 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 order.
place, namely to the right of \(A[j]\) (see Table I(c)). Similarly, if the split is to the left of \(A[j]\), 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 at a different location. We then start the rightward scan at the split on the left of \(A[j]\) 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 \(A[j]\) and repeat the double scan on this new middle part. The process is repeated until the middle part consists of only one element, namely \(A[j]\). 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.

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:

\[ \forall p, q(1 \leq p \leq f \leq q \leq N \Rightarrow A[p] \leq A[f] \leq A[q]) \]

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 formally what must happen in the program. In the case of Find, it will be necessary to know the exact location of the split in the middle part. This indicates the introduction of a new variable \(s\) to denote the position of the split.
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
    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 Find
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 := 130
    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 <= [i..j](j) + 1;
            r <= A([i..j](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 AND (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 AND (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
The array lemma is the result of scanning the path condition $S$ 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;
DECLARE $A_5$ : 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 $A_5(A_5_i) = A_3(A_5_i)$; $A_5(A_5_i)$; \}
<table>
<thead>
<tr>
<th>path</th>
<th>execution point</th>
<th>path condition stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>enter loop $m \cdot n$</td>
<td>(1) $f_{\text{Init}}$</td>
</tr>
<tr>
<td></td>
<td>top of $m \cdot n$</td>
<td>(2) $m \cdot n$ invariant</td>
</tr>
<tr>
<td></td>
<td>after iteration</td>
<td>(3) false $m \cdot n$ exit condition</td>
</tr>
<tr>
<td>2</td>
<td>enter loop $i \cdot j$</td>
<td>(4) $i \cdot j$ invariant</td>
</tr>
<tr>
<td></td>
<td>top of $i \cdot j$</td>
<td>(5) false $i \cdot j$ exit condition</td>
</tr>
<tr>
<td></td>
<td>after iteration</td>
<td>(6) $i \cdot j$ invariant</td>
</tr>
<tr>
<td>3</td>
<td>enter loop $less \cdot r$</td>
<td>(7) false $less \cdot r$ exit condition</td>
</tr>
<tr>
<td></td>
<td>top of $less \cdot r$</td>
<td>(8) $less \cdot r$ invariant</td>
</tr>
<tr>
<td></td>
<td>after iteration</td>
<td>(9) false $less \cdot r$ exit condition</td>
</tr>
<tr>
<td>4</td>
<td>execute loop $less \cdot r$</td>
<td>(10) $less \cdot r$ invariant</td>
</tr>
<tr>
<td></td>
<td>(11) false $less \cdot r$ exit condition</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>enter loop $greater \cdot r$</td>
<td>(12) $greater \cdot r$ invariant</td>
</tr>
<tr>
<td></td>
<td>top of $greater \cdot r$</td>
<td>(13) false $greater \cdot r$ exit condition</td>
</tr>
<tr>
<td></td>
<td>after iteration</td>
<td>(14) $greater \cdot r$ invariant</td>
</tr>
<tr>
<td>6</td>
<td>execute loop $greater \cdot r$</td>
<td>(15) false $greater \cdot r$ exit condition</td>
</tr>
<tr>
<td>7</td>
<td>execute loop $i \cdot j$:</td>
<td>(16) $i \cdot j$ exit condition</td>
</tr>
<tr>
<td></td>
<td>if branch</td>
<td>(17) $f_{1} \leq j_{2}$</td>
</tr>
<tr>
<td>8</td>
<td>execute loop $i \cdot j$:</td>
<td>(18) $f_{1} \geq i_{2}$</td>
</tr>
<tr>
<td></td>
<td>else branch</td>
<td>(19) $f_{1} \geq i_{2}$</td>
</tr>
<tr>
<td>9</td>
<td>execute loop $m \cdot n$:</td>
<td>(20) $m \cdot n$ exit condition</td>
</tr>
<tr>
<td></td>
<td>if branch</td>
<td>(21) $f_{1} \leq j_{2}$</td>
</tr>
<tr>
<td>10</td>
<td>execute loop $m \cdot n$:</td>
<td>(22) $f_{1} \geq i_{2}$</td>
</tr>
<tr>
<td></td>
<td>else branch</td>
<td>(23) $f_{1} \geq i_{2}$</td>
</tr>
<tr>
<td>11</td>
<td>execute $m \cdot n$ loop:</td>
<td>(24) $m \cdot n$ loop</td>
</tr>
<tr>
<td></td>
<td>exit branch</td>
<td>(25) $m \cdot n$ loop</td>
</tr>
<tr>
<td>12</td>
<td>exit $m \cdot n$ loop</td>
<td>(26) $m \cdot n$ loop</td>
</tr>
<tr>
<td>13</td>
<td>exit Find</td>
<td>(27) $m \cdot n$ loop</td>
</tr>
</tbody>
</table>

*nothing to prove*
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 FINE that this technique will work up is not to be included.
| Path | Execution Point | m | n | f | g | 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 | A_3 | i_2 | j_2 | w_2 |
|      | top of i_j after an iteration | A_3 |
| 3    | enter loop less_r | A_3 |
|      | 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_3-1 |
| 6    | execute loop greater_r | A_3 |
|      | execute i_j loop: | A_3 |
|      | if branch | i_3 | j_3 | w_2 |
|      | A_6 = A_3[i_3 | A_3(j_3)] | A_5 = A_4[j_3 | A_3(i_3)] |
|      | else branch | A_3 |
| 8    | execute loop i_j: | A_3 |
|      | else branch | A_3 |
| 9    | execute loop m_n: | n_1 |
|      | if branch | j_2 |
| 10   | execute loop m_n: | m_1 | n_1 |
|      | elseif branch | i_2 |
| 11   | execute loop m_n: | exit branch | nothing to prove |
| 12   | exit loop m_n | nothing to prove |
| 13   | exit Find | nothing to prove |
## Symbol Table Stack

<table>
<thead>
<tr>
<th>path</th>
<th>execution point</th>
<th>m</th>
<th>r</th>
<th>f</th>
<th>i</th>
<th>j</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>enter m,n loop</td>
<td>1</td>
<td>100</td>
<td>f,1</td>
<td>A,1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>top of m,n after an iteration</td>
<td>m,1</td>
<td>r,1</td>
<td>A,2</td>
<td>r,1</td>
<td>i,1</td>
<td>j,1</td>
</tr>
<tr>
<td>2</td>
<td>enter i,j loop</td>
<td>m,1</td>
<td>r,1</td>
<td>A,2(f,1)</td>
<td>m,1</td>
<td>n,1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>top of i,j after an iteration</td>
<td>A,3</td>
<td>i,2</td>
<td>j,2</td>
<td>w,2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>enter loop less_r</td>
<td>A,3</td>
<td>i,3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>execute loop less_r</td>
<td>A,3+i,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>enter loop greater_r</td>
<td>A,3+i,3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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.
$ 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;  $ initial value
DECLARE A_1( INTEGER ) : INTEGER;

$ 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, r_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

$ \text{prove new invariant true}$

$ \text{initial path condition}$

$ \text{m}_n \text{ loop invariant}$

$ \text{false m}_n \text{ loop exit condition}$

$ \text{i}_j \text{ loop invariant}$

$ \text{false i}_j \text{ loop exit condition}$

\begin{verbatim}
PROVE IF AND
  f_Init;
  1 <= m_1;
  m_1 <= f_1;
  f_1 <= n_1;
  n_1 <= 100;
  NOT m_1 >= n_1;
  AND
  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;
\end{verbatim}
VC7 THEOREM--IF PART 2

S less_r loop invariant

\[ i_3 \leq f_1; \]
\[ A_2(f_1) \leq A_3(f_1); \]

S j_2 for \([i,j](j)\)
\[ j_2 \leq f_2 \]
A_2(f_1) \leq A_3(j_2 + 1); S j_2 for \([i,j](j)\)

\[ j_3 \leq j_2 + 1; \]
A_2(f_1) \leq A_3(j_2 + 1); S j_2 for \([i,j](j)\)

S less_r loop exit condition
i_3 \geq i_2; S NEW( i ) \geq OLD( i )

S greater_r loop invariant

\[ f_1 \leq j_3; \]
A_2(f_1) \geq A_3(f_1);\]

\[ i_2 - 1 \leq j_3; \]
A_2(f_1) \geq A_3(i_2 - 1); S i_2 for \([i,j](i)\)

S greater_r loop exit condition
j_3 \leq j_2; S NEW( j ) \leq OLD( j )

S if branch condition
i_3 \leq j_3; S result of directive \(\text{PROVE (..next two statements..)}\) USING \SWAP(A)

A_5(i_3) \leq A_2(f_1);
A_5(j_3) \geq A_2(f_1);
VC7 THEOREM--THEN PART

$\text{new i}_j \text{ loop invariant}$

\[
\begin{align*}
\text{THEN AND} & \quad (m_1 \leq i_3 + 1; \\
& \quad j_3 - 1 \leq n_1; \\
& \quad \text{IF i}_3 + 1 \leq j_3 - 1 \\
& \quad \text{THEN AND} \quad (c \\
& \quad \text{OR} \quad \text{c} \\
& \quad \text{AND} \quad (i_3 + 1 \leq f_1; \\
& \quad \quad A_2(f_1) \leq A_5(f_1); \\
& \quad \text{AND} \quad (i_3 + 1 \leq (j_3 - 1) + 1; \\
& \quad \quad A_2(f_1) \leq A_5((j_3 - 1) + 1); \\
& \quad \text{OR} \quad \text{c} \\
& \quad \text{AND} \quad (i_3 + 1 \leq (j_3 - 1) - 1; \\
& \quad \quad A_2(f_1) \geq A_5(f_1); \\
& \quad \text{AND} \quad ((i_3 + 1) - 1 \leq j_3 - 1; \\
& \quad \quad A_2(f_1) \geq A_5((i_3 + 1) - 1); \\
& \quad \text{)} \\
& \quad \text{)} \\
& \text{)} \\
& \text{)} \\
& \text{)} \\
& \text{)} \end{align*}
\]
References


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


