Stanford Artificial Intelligence Laboratory
Memo AIM-269

October 1975

Computer Science Department
Report No. STAN-CS-75-522

Automatic Program Verification IV:
PROOF OF TERMINATION WITHIN A WEAK LOGIC OF *PROGRAMS

by

David C. Luckham and Norihisa Suzuki

Research sponsored by

Advanced Researoh Projects Agency
ARPA Order No. 2404

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-269

October 1975

Computer Science Department
Report No. STAN-CS-75-522

Automatic Program Verification IV:
PROOF OF TERMINATION WITHIN A WEAK LOGIC OF PROGRAMS

by

David C. Luckham and Norihisa Suzuki

ABSTRACT

A weak logic of programs is a formal system in which statements that mean "the program halts" cannot be expressed. In order to prove termination, we would usually have to use a stronger logical system. In this paper we show how we can prove termination of both iterative and recursive programs within a weak logic by adding pieces of code and placing restrictions on loop invariants and entry conditions. Thus, most of the existing verifiers which are based on a weak logic of programs can be used to prove termination of programs without any modification. We give examples of proofs of termination and of accurate bounds on computation time that were obtained using the Stanford Pascal program verifier.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435 . The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, or the U. S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 2215 1.

Proof of Termination within a Weak Logic of Programs

by

David Luckham and Nori Suzuki

1. INTRODUCTION.

A weak logic of programs is one in which statements that "a program halts" cannot be expressed. Such a logic has been given by [Hoare 69,71] and its proof theory has been defined and studied in [Igarashi, London & Luckham](referred to as ILL), [Hoare & Lauer], [Cook]. Other recent papers have been devoted to strengthening this logic so that questions of termination are expressable; e.g. Dijkstra's notion of weakest precondition [Dijkstra], and various suggestions for introducing well-orderings into the assertion language.

Here we give a simple application of the method of Virtual Programming which permits strong statements of termination (e.g. program A halts and Q is true) to be deduced from weak statements (if A halts then Q is true) by means of the good old law of excluded middle. (Remark: the notion of virtual program in intuitive terms is simply code added to an actual program which has no effect on the actual values of the result parameters.) The method requires no change whatever in the weak logic, and employs exactly the same automated techniques that are currently used to verify all manner of properties of programs [ILL, Suzuki 75a,b, von Henke & Luckham]. This permits strong proofs of correctness (i.e. termination and consistency with

specifications) to be obtained using the present verification systems based on the weak logic, Similar ideas have been put forward by Knuth [Knuth], in which he showed how one can prove the termination of extended Euclid's algorithm as the byproduct of the correctness proof using Floyd's method.

Virtual program has been used previously to document programs in certain tricky situations (e.g. where the documentation uses data structures not used by the actual program, such as history sequences [Clint], or data structures destroyed by the actual program [v.Henke & Luckham]). More recently it has been used to prove complexity bounds on program computations [Farmwald]. The technique seems to present a natural approach to proving dynamic properties of programs (i.e. properties of the computations themselves as distinct from the final results). Termination is one of these dynamic properties.

Essentially, most programs halt for simple reasons, and the programmer usually knows those reasons. What is needed is a natural way of permitting him to state his reasons, Our proposal here is simply to introduce virtual program counters into the program. The function of these counters is to "count" the number of computation steps that are executed. Each path in the program must have added to it an assignment statement which increments the counter proportionally to the length of the path. The programmer must also add inductive assertions stating in effect that the values of the counters are bounded. Presumably he has an idea of a reasonable upper bound, and that is all that is necessary. The problem of proving termination within the usual weak logic then becomes merely another verification problem --namely the proof of the boundedness of the counters in the augmented program.

There is one' "catch". The counters **must account** for every possible loop and recursion (i.e. every potential source of infinitely long computation), otherwise a correct weak statement will **not** imply termination. A test for this provision can easily be automated.

In section 2 we illustrate the method and the "catch" by simple examples. An outline of a rigorous justification is given in Section 3, and actual results using our present verifier [ILL, Suzuki75a,b] are included in Section 4.

To, simplify matters, we have restricted the discussion to Pascal programs containing Assignment, Conditional, and While statements, function calls, and recursive procedure calls. The extension of the method to GoTo's and other statements is obvious. Also, we have assumed that the reader has an acquaintance with some of the literature on verifiers based on the weak logic of programs (see the references).

2. THE METHOD.

Our method involves the use of very simple virtual programs. Virtual program is defined rather loosely as a set of instructions imbedded into the program to be verified so that it does not interfere with the original program (often called the actual program). We shall .use only virtual assignment statements, and their left hand-sides will be required to be ghost variables--variables which are not used in the original program. No other kind of modifications will be allowed. It is clear that the addition of such instructions cannot change the behaviour of the actual program on the actual program variables.

As an example let us look at the following program for multiplication by addition.

Program 1.

```
ENTRY: a,b:INTEGER;
       x←a;
       y←0;
       while x≠0 do
             begin
             x←x-1;
             y←y+b
             end.
EXIT: y=a*b;
```

If we want to measure the time taken to compute multiplication with the assumption that the assignment statement and testing both take a unit computation time, we can modify the program by introduction of virtual program as follows,

Program 2.

```
ENTRY: a,b:INTEGER;
       x←a;
       y←0;
       counter←2;
       while x≠0 do
             begin
             x←x-1;
             y←y+b;
             counter←counter+3
             end.
EXIT:  y=a*b ∧ counter=3*a*b+2;
```

Suppose we can prove within the weak logic-of programs that program 2 augmented by the new assignment instructions satisfies the new EXIT condition. The value of the counter in the EXIT is a function of the input parameters only. So, we will have proved that whenever the augmented program stops the counter is bounded by a bound that is given before the computation starts. Now assume that we have **put the** virtual assignments "in all the right places" so that every possible computation **path** contains an assignment which increments the counter by the number of instructions **on the path.** Then we will have proved that either the actual program 1 will stop within a number of steps less than the bound, or it will compute forever. This would give **us** a tool for proving bounds on the complexity of computations of **programs using standard** verification techniques.

There is one problem: the user is responsible for putting virtual assignments of **the form counter←f(counter)** that increment the counter correctly in a sufficient number **of places.** Having done this, verifying the computation bounds **becomes a problem of verifying a statement** about the augmented program in the weak logic, and

the **verifier can** be **used to** aid in solving it.. Note that we do not have to extend the weak logic in any way.

We might ask whether we cannot prove termination at the same time. This simple thought presents another problem. See the following example,

Program 3.

```
ENTRY TRUE;
      x←0;
      counter←1;
      while x>0 do
            begin
            x←x+1;
            counter←counter+2
            end.
EXIT counter<1;
```

This program certainly does not terminate. But it is easily proved to be weakly consistent with the output assertion, COUNTER<1. And the counter is clearly counting all possible computation steps. The weak correctness proof goes as follows. We take the inductive assertion $X>0$ as invariant of the loop. Then, three verification conditions are generated corresponding to three paths in the program,

(1). $TRUE \rightarrow 0>0$.

(2). $X>0 \wedge X>0 \rightarrow X+1>0$.

(3). $X>0 \wedge \neg X>0 \rightarrow COUNTER<1$.

They are all valid. Condition (1) requires . that when the control reaches the **while-statement, the invariant** will be satisfied intially. And (2) guarantees that $X>0$ is the invariant of the loop. Condition (3) is valid since the antecedent, **namely** $X>0 \wedge \neg X>0$, is a contradiction; that is, the path to the EXIT is never executed.

The question of proving termination is really asking under what conditions a weak statement about a program with a virtual counter ("if P stops then counter<b") implies a strong statement ("P must stop in <b steps"). As is evident from the above example such implications are not always valid. In order for a weak statement to imply a corresponding strong statement all iterative statements must have an **invariant** assertion stating that the counter is bounded. We shall call these "bound assertions".

In our example, the inductive assertion about the while loop does not even contain the counter., If we try to force the proof above to be a proof of termination, we have to give a stronger loop assertion so that if the program does **not** terminate and the control repeats the loop indefinitely, this assertion eventually becomes **false.** Then, the verification condition corresponding to the loop is no longer valid.

The question is, **"can** we always find such strengthening of the loop invaliant?" And the answer is **,"yes."** The expression which we have to add is COUNTER<g(X), where X is a set **of** input values of program 'parameters. This assertion gives **the** upper **bound** for the value of COUNTER.

The method for While Statements:

Each while statement is associated with a variable, COUNTER, which does not appear in the actual program. A COUNTER may be associated with many while statements. For each while statement, the user must add to the while body a counter **assignment,** COUNTER←f(COUNTER,*X0*), where f is a strictly increasing integer **valued function and** X0 is a set of variables not occurring in the actual program. The

user must also add an inductive assertion, COUNTER<g(*X0*), where g is a well defined function of *X0*, to the While statement.

The same kind of technique can be used to prove termination of procedures with recursive calls. Here, the potential source of infinite computation is the execution of arbitrarily **many** calls. The role of the counter will be to "count" the number of recursive calls by being incremented by COUNTER ← f(COUNTER,*X0*), f a strictly increasing integer valued function, each time a call occurs. So we have to place the counter assignments where they will be executed whenever procedures are evoked. One place which meets that requirement is the beginning of the procedure body. Also we need to add bound assertions that will become false if the depth of procedure calls exceed a certain level. The best candidate is the ENTRY condition of the procedure. The counter is introduced as an additional VARIABLE parameter of . the procedure since it must be global to every call.

We note that the bound assertion, say COUNTER < g(*X0*) , must be fixed for all calls; therefore *X0* must not contain any parameters appearing in procedure calls otherwise the bound would change with the actual values of those parameters. We can think of *X0* as being initial values of parameters of the outermost procedure call.

Below we give an example.

Program 4

```
procedure factorial (var X;N);
ENTRY N≥0;
EXIT X=N!;
        if N=0 then X ← 1 else
        begin
                factorial (X,N-1);
```

```
            X ← N*X
        end.
```

This is a procedure which calculates factorial N and returns the result in X. The entry and the exit conditions are N ≥ 0 and X = N! respectively. We change the program with COUNTER assignments.

Program 5

```
procedure factorial(var X,COUNTER; N);
ENTRY N≥0 ∧ COUNTER≤N0 ∧ COUNTER+N=N0;
EXIT X=N!;
        begin
                COUNTER ← COUNTER + 1;
                if N-0 then X ← 1 else
                begin
                        factorial(X,COUNTER,N-1);
                        X ← N*X
                end.
```

Notice the new entry condition contains not only a bound assertion, COUNTER≤N0, but also an inductive assertion stating an invariant relationship between COUNTER and values of the parameter N in successive calls. Notice also that NO has been introduced so that the bound contains no parameter of the procedure; NO is the initial value of N at the outermost procedure call. So what we are going to prove is

```
N≥0 ∧ COUNTER+N=N0 ∧ COUNTER≤N0
{ COUNTER ← COUNTER t 1;
    if N-0 then X ← 1 else
   begin
        factorial(X,COUNTER,N-1);
        X ← N*X
   end, }
X = N!
```

with the assumption that

```
Y≥0 ∧ COUNTER+Y=N0 ∧ COUNTER≤N0 { factorial(X,COUNTER,Y) } X-Y!
```

Using techniques for proving weak correctness of procedure call [Hoare71, ILL, Suzuki75b], verification conditions are

(I) $N>0 \wedge COUNTER+N=N0 \wedge COUNTER<N0 \supset (N=0 \supset 1=0!)$.

(II) $N>0 \wedge (COUNTER+N=N0 \wedge COUNTER<N0 \supset (N\neq 0 \supset N-1>0 \wedge COUNTER+1+N-1=N0 \wedge$
$COUNTER<N0 \wedge (X00=(N-1)! \supset N*X00!=N!))$

which are all valid. So the actual program 4 is correct and also terminates.

The Method for Recursive Procedures

Each procedure declaration is associated with a variable, COUNTER, not appearing in, the actual program. Then (1) COUNTER is introduced as a new VARIABLE parameter of the, procedure, and all calls are correspondingly modified; (2)the user must place at the beginning of the procedure body, a counter assignment, COUNTER $\leftarrow f(COUNTER, X0)$, where f is strictly increasing and $X0$ is a set of variables not appearing in any procedure body; (3) the user must add a bound assertion, COUNTER $< g(X0)$, to the ENTRY condition of the procedure.

3. JUSTIFICATION OF THE METHOD,

We are going to show that this method of adding virtual program counters to while statements and procedures with recursive calls is sufficient to prove termination, That is, a proof of weak correctness of the augmented program guarantees the termination of the actual program. We have omitted the case for **goto** statements but we can treat them likewise.

(i) *While statements.*

The augmented program for while statements

```
WHILE C 00 S
```

is

```
WHILE C 00
        ( COUNTER ← f (COUNTER, X0) ; S).
```

where f(COUNTER,*X0*) is an integer valued strictly increasing function, that is

COUNTER < f(COUNTER,*X0*),

and *X0* is a set of new variables not occurring in S. The form of the invariant of the **loop** (or the inductive assertion) must be

I A COUNTER < g(*X0*) where I is any Boolean assertion.

Now we are going to prove that if the augmented **program of** this form with the given inductive assertion is verified then the actual program terminates. Various proofs of **the** soundness (i.e. semantic -consistency) of the weak logic of programs have been given (see **[Hoare & Lauer], [Igarashi,** London & Luckham], [Cook]). These proofs construct a model (essentially an abstract interpreter for the programming language) with the property that any statement about a program that is provable in

this logic is true when the program is "run" on the interpreter. This means for example, that if we can prove

I { while L do A } I in the logic of programs,

I being a Boolean assertion invariant of the loop, then when "while L do A" is run on the abstract machine, the computation state at the end of every execution of the loop will satisfy I.

We shall show that the provability of weak statements about programs augmented by counters according to our method implies that the computations of those programs on the abstract machine halt. We present our argument with some degree of informality since we do not wish to burden the reader with the formal details of the model here. We shall simply refer to a "standard machine" which the reader can imagine is an interpreter for the axiomatic semantics of Pascal,

Proof

Suppose the augmented program for a simple while statement as shown with the given inductive assertion is proved. Suppose also that the program does not terminate when run on the standard machine. We are going to show that this assumption produces a contradiction. We are going to number the values of COUNTER so that COUNTER is the value of COUNTER when the control goes around the loop i times. Since the program does not terminate, we are going to have an infinite sequence of values,

$\mathrm{COUNTER}_0, \ldots, \mathrm{COUNTER}_n, \ldots$

Because of the assignment statement

COUNTER ← f(COUNTER,X0),

which takes place between two successive values of COUNTER's, we have the relation

$$COUNTER_{i+1} = f(COUNTER_i, X0).$$

However, f is strictly increasing and also it is an integer function; therefore,

$$COUNTER_{i+1} > COUNTER_i + 1.$$

From the above relation

$$COUNTER_n > COUNTER_0 + n$$

So for any integer k we can select m such that $COUNTER_m > k$. This contradicts the fact that the loop invariant is of the form

I A COUNTER ≤ g(X0), which is true for every iteration and g(X0) must remain constant throughout the computation.

So the program must terminate.

(ii) *Recursive procedures.*

The augmented program for procedure

```
procedure k(X); B
```

is

```
procedure k (X; var COUNTER);
        begin
                COUNTER ← f (COUNTER, X0);
                B
        end.
```

The function f must be strictly increasing as was the case in (i). The bound assertion

is now added to the entry condition of the procedure , which must take the same **form as in the** case of while statements.

Proof

Assume first that B contains no while loops nor calls to procedures **other than k. Suppose the** augmented program with input assertions of the form

$$I \wedge COUNTER \leq g(X0)$$

can be verified. That is

$$I \wedge COUNTER \leq g(X0)\ \{\ \text{begin } COUNTER \leftarrow f(COUNTER, X0);\ B\ \text{end}\ \}\ Q$$

is provable with the assumption that

$$I \wedge COUNTER \leq g(X0)\ (\ k(X)\ )\ Q.$$

Suppose also the program does not terminate. Then the depth of recursive calls to **this procedure** is infinitely large. In this case, we are going to number the values of **COUNTER** at the beginning of the procedure so that $COUNTER_i$ is the value **of COUNTER at the i-th** level of procedure call in the current calling sequence. So we have a sequence of values

$$COUNTER_0, \ldots, COUNTER_n, \ldots$$

Because of the assignment statement

$$COUNTER \leftarrow f(COUNTER, X0)$$

at the beginning of the procedure body so that it is alwaysexecuted after a call, we have the following relation

$$COUNTER_{i+1} = f(COUNTER_i, X0),$$

for all-i. As in the previous case, for any integer k, we can choose m such that

$COUNTER_m > k$.

This is a contradiction because at each procedure call

$I \wedge COUNTER \leq g(X0)$

must hold just at the entry to the body. Note that $g(X0)$ must remain constant over all calls because $X0$ does not contain any program variables.

The above arguments for a single loop and a single recursive procedure can be generalized for nested loops and mutual recursive procedures. Essentially, if there is an infinite computation of an augmented program with n counters, one of those counters will be incremented infinitely many times.

4. EXAMPLES.

All **proofs below** were obtained using the Stanford Pascal Verifier. This system **is implemented** in LISP and runs on a PDP-10 in about **50K** words of memory. The main references for details of this verifier are **[Igarashi,** London **& Luckham,** Suzuki **a,b].**

The first example is Dijkstra's square root program which computes square root of **N.** The problem here is to verify both that the program halts and computes an integer approximation to the square root of **N.** The program has been augmented by operations on the virtual variable COUNTER. Termination of the square root program is verified by proving the bound assertion on COUNTER for the augmented program. The 'documentation is expressed in terms of user-defined concepts such as **"B2** is a power of four" and "the integer logarithm base 4 of **B2". Notice that the EXIT** condition implies that the loop is executed at most **ILOG4(B20)** times, where B20 is the intial value of parameter B2.

```
PASCAL
ENTRY  (N>0) ∧ (B2>N) ∧ POWER_OF_FOUR(B2)∧(B2=B20);
EXIT  (0 ≤ A) ∧ (A*A ≤ N) ∧ (N < (A+1)*(A+1))
        ∧ (COUNTER ≤ ILOG4(B20));

BEGIN
A2 := 0;
AB := 0;
COUNTER:=0;
INVARIANT  POWER_OF_FOUR(B2)∧(AB*AB = A2*B2)∧(AB≥0) ∧ (B2>0)
        ∧ (A2+2*AB+B2>N) ∧ (A2≤N)
        ∧ (ILOG4(B2)+COUNTER=ILOG4(B20))
        ∧ (COUNTER ≤ ILOG4(B20))
WHILE 1 ≠ B2 DO
        BEGIN
        AB := A6 DIV 2;
        B2 := B2 DIV 4;
```

```
        T := A2 + 2*AB + B2;
        COUNTER:=COUNTER+1;
        IF T ≤ N THEN
                BEGIN
                A2 := T;
                AB := AB + B2
                END
        END;
A := AB;
E N D . :
```

*****

FOR THE MAIN PROGRAM
THERE ARE 3 VERIFICATION CONDITIONS

```
# 1
(0<N &
 N<B2 &
 POWER_OF_FOUR(B2) &
 B2=B20
→
 POWER_OF_FOUR(B2)    &
 0*0=0*B2 &
 0≤0 &
 0<B2 &
 N<0+2*0+B2 &
 0≤N &
 0≤ILOG4(B20)   &
 ILOG4(B2)+0=ILOG4(B20) &
 (¬¬1=B200 &
  POWER_OF_FOUR(B200) &
  AB00*AB00=A200*B200 &
  0≤AB00 &
  0<B200 &
  N<A200+2*AB00+B200 &
  A200≤N &
  COUNTER00≤ILOG4(B20) &
  ILOG4(B200) +COUNTER00=ILOG4(B20)
 →
  0≤AB00 &
  AB00*AB00≤N &
  N< (AB00+1)*(AB00+1) &
  COUNTER00≤ ILOG4(B20)))

# 2
(¬A2+2*(AB DIV 2)+B2 DIV 4≤N 6
 ¬1=B2 &
 POWER_OF_FOUR(B2) &
```

```
 AB*AB=A2*B2 &
 0≤AB &
 0<B2 &
 N<A2+2*AB+B2 &
 A2≤N &
 COUNTER≤ILOG4(B20) &
 ILOG4(B2)+COUNTER=ILOG4(B20)
→
 POWER_OF_FOUR(B2 DIV 4) &
 (AB DIV 2)*(AB DIV 2)=A2*(B2 DIV 4) &
 0≤AB DIV 2 8
 0<B2 DIV 4 &
 N<A2+2*(AB DIV 2)+B2 OIV 4 &
 A2≤N 6
 COUNTER+1≤ILOG4(B20) &
 ILOG4(B2 DIV 4)+COUNTER+1=ILOG4(B20))

# 3
(A2+2*(AB DIV 2)+B2 DIV 4≤N &
 ¬1=B2 &
 POWER_OF_FOUR(B2) &
 AB*AB=A2*B2 &
 0≤AB &
 0<B2 &
 N<A2+2*AB+B2 &
 A2≤N &
 COUNTER≤ILOG4(B20) &
 ILOG4(B2)+COUNTER=ILOG4(B20)
→
 POWER_OF_FOUR(B2 DIV 4) 8
 (AB DIV 2+B2 DIV 4)*(AB DIV 2+B2 DIV 4)=
        (A2+2*(AB DIV 2)+B2 DIV 4)*(B2 DIV 4) &
 0≤AB DIV 2+B2 DIV 4 &
 0<B2 DIV 4 &
 N<A2+2*(AB DIV 2)+B2 DIV 4+2*(AB DIV 2+B2 DIV 4)+B2 DIV 4 &
 A2+2*(AB DIV 2)+B2 OIV 4≤N &
 COUNTER+1≤ILOG4(B20) &
 ILOG4(B2 DIV 4)+COUNTER+1=ILOG4(B20))
```

These verification conditions all simplify to TRUE using the SIMPLIFIER with the lemmas (AXIOMS and GOALS) in' the **GOALFILE** below. The total time for the complete **verification** is 39 CPU seconds.

The lemmas describe properties of POWER-OF-FOUR(X), X DIV Y, EVEN(X), and **ILOG4(** X), and are supplied by the user. They are written in a form which indicates

how they are to be used by the SIMPLIFIER [Suzuki al. To read them as logical statements simply ignore all occurrences of "8". A lemma of the form "AXIOM **A↔B**" means **"A=B".** "GOAL A SUB **B**" means **"B→A".**

With this advice readers should be able to understand the lemmas (while those aquainted with our previous reports will also understand how they are used by the SIMPLIFIER in the proofs). Only three of these arithmetical lemmas (those marked by a **"*")** are needed to prove that the program halts within **ILOG4(B20)** executions , of the loop. This reflects the fact that the loop is controlled by a single instruction, **B2:=B2** DIV **4.** So termination is a much simpler problem than correctness of the output in this case, and can be **checked** almost "for free".

```
GOALFILE
        AXIOM @P1<@P2 ↔ P1+1≤P2;
        GOAL @P1≤@P2 SUB (P1≤@P3)∧(@P3≤P2);
        GOAL 0 ≤ @X + @Y SUB (0 ≤ X) A (0 ≤ Y);
        GOAL @P1≤@P2 SUB (P1≤@P3)∧(@P3=P2);
        AXIOM IF(I≠J) THEN @I≥@J ↔ I>J;

        GOAL POWER_OF_FOUR(@I1 OIV 4)SUB POWER_OF_FOUR(I1);
        GOAL 1 ≤ (@I OIV 4) SUB POWER_OF_FOUR(I)∧(1<I);

        AXIOM (@K*@L)DIV @K ↔ L;
        AXIOM IF M+1≤K THEN ((@K*@L)+@M)DIV @K ↔ L;
        GOAL 0≤@P1 DIV @P2 SUB (P2 ≥ 0)∧(P1 ≥ 0);

        GOAL EVEN(@Z) SUB (Z*Z=A2*@X)∧POWER_OF_FOUR(X);
        AXIOM IF EVEN(X) THEN (@X DIV 2)*(@X DIV 2)↔ (X*X)DIV 4;
        AXIOM IF POWER-OF-FOUR(X) THEN @Y*(@X DIV 4)↔ (Y*X)DIV 4;

        GOAL (@X DIV @I)=(@Y DIV @I) SUB X=Y;
        AXIOM IF EVEN(X) THEN 4*(@X OIV 2) ↔ 2*X;
        AXIOM IF POWER-OF-FOUR(I) THEN 4*(@I DIV 4)↔ I;

        GOAL @X*@Y=@Z*@Y SUB (Y≠0)⊃(X=Z);

*       GOAL ILOG4(@X)≥0 SUB X≥1;
*       AXIOM ILOG4(@X DIV 4)↔ ILOG4(X)-1;
*       GOAL ILOG4(@X)≥1 SUB (X>1)∧POWER_OF_FOUR(X);
```

The next example shows how the termination of a recursive procedure can be proved using a counter and the entry assertion which states a bound on the value of the counter. The procedure PCCD computes the greatest common divisor of M and N and returns the value as R. This is stated as the exit assertion,

PASCAL

```
PROCEDURE PGCD(VAR COUNTER,R: INTEGER;M,N: INTEGER);
        ENTRY (M≥0)∧(N>0)∧(COUNTER≤N0)∧(COUNTER+N≤N0) ;
        EXIT R=GCD(M,N);
        BEGIN
        COUNTER := COUNTER+1;
        I := MOD(M,N);
        IF I=0 THEN R := N ELSE PGCD (COUNTER,R, N, I)
        END;
```

*****

```
FOR PGCD
THERE ARE 2 VERIFICATION CONDITIONS

# 1
.(MOD(M,N)=0 &
 0≤M &
 0<N &
 COUNTER≤N0 &
 COUNTER+N≤N0
→
 N=GCD (M, N) )

# 2
(¬MOD(M,N)=0 &
 0≤M &
 0<N &
 COUNTER≤N0 &
 COUNTER+N≤N0
→
 0≤N &
 0<MOD (M, N) &
 COUNTER+1 ≤N0 &
 COUNTER+1+MOD (M, N) ≤N0 6
 (R00=GCD (N, HOD (M, N) )
 →
  R00=GCD (M, N) ))
```

These verification conditions are simplified to TRUE using the following arithmetic lemmas and lemmas describing properties of **GCD(X,Y).** The computation took 8 CPU seconds.

```
GOALFILE
        AXIOM 8 ≤ MOD(@M,@N) - 'TRUE:
        'AXIOM MOD(@M,@N)+1 ≤@N ↔ TRUE:
        AXIOM IF X=Y*@Q THEN MOD(@X,@Y) ↔ 8;

        GOAL @P3 ≤@P1+@P2 SUB (P3≤P1)∧(0≤P2);
        AXIOM @P1<@P2↔P1+1≤P2;
        GOAL @X ≤@Y SUB (X-1≠Y)∧(X-1≤Y);

GOALFILE
        AXIOM IF Y=MOD(@R,X) THEN GCD(@X,@Y) ↔ GCD(R,X);
        AXIOM IF MOD(X,Y)=0 THEN GCD(@X,@Y) ↔ Y;
```

The last example is the procedure **SIFTUP** used in the TREESORTS algorithm. The properties that the output array is ordered and the output array is a permutation of the input array have been proved for the whole **TREESORT3** algorithm by this verifier [Suzuki 75a]. We verify here that **SIFTUP** terminates and the computation time required is proportional to the logarithm of the size N/10,

```
PASCAL

PROCEDURE SIFTUP(I0,N:INTEGER);
ENTRY (K=ILOG2(DIV(N,I0)))∧(COUNTER=0)∧(K≥0);
EXIT  COUNTER≤K+1 ;

VAR COPY:REAL; J, I: INTEGER:

BEGIN
        I ← I0; COPY ← M[I];
    18: J ← 2 * I:
        ASSERT (COUNTER=ILOG2 (DIV( I, I 0)))∧(COUNTER≤K)∧
               (K=ILOG2(DIV(N,I0)))∧(J=2*I);
      COUNTER←COUNTER+1;
        IF J ≤ N THEN
                BEGIN
                IF J < N THEN
                        BEGIN
                        IF M[J+1] > M[J] THEN J ← J+1
                        END:
                IF M[J] > COPY THEN
                        BEGIN M[I] ← M[J]; I ← J; GO TO 10 END;
                END;
       M[I] ← COPY;
END:.:


*****


FOR SIFTUP
THERE ARE 8 VERIFICATION CONDITIONS


# 1
(K=ILOG2(DIV(N,I0)) &
 COUNTER-0  6  ,
 0≤K
→
 COUNTER=ILOG2(DIV(I0,I0)) &
```

```
 COUNTER≤K  &
 K=ILOG2(DIV(N,I0)) &
 2*I0=2*I0)

# 2
(COPY<M[J+1] &
 M[J]<M[J+1] &
 J<N &
 J≤N &
 COUNTER=ILOG2(DIV(I,I0)) &
 COUNTER≤K &
 K=ILOG2(DIV(N,I0)) &
 J=2*I
→
 COUNTER+1=ILOG2(DIV(J+1,I0)) &
 COUNTER+1≤K &
 K=ILOG2(DIV(N,I0)) &
 2*(J+1)=2*(J+1))

# 3
(COPY<M[J] &
 ¬M[J]<M[J+1] 6
 J<N &
 J≤N &
 COUNTER=ILOG2(DIV(I,I0)) &
 COUNTER≤K &
 K=ILOG2(DIV(N,I0)) &
 J=2*I
→
 COUNTER+1=ILOG2(DIV(J,I0)) &
 COUNTER+1≤K &
 K=ILOG2(DIV(N,I0)) &
 2*J=2*J)

# 4
(COPY<M[J] 6
 ¬J<N &
 J≤N &
 COUNTER=ILOG2(DIV(I,I0)) &
 COUNTER≤K &
 K=ILOG2(DIV(N,I0)) &
 J=2*I
→
 COUNTER+1=ILOG2(DIV(J,I0)) 6
 COUNTER+1≤K &
 K=ILOG2(DIV(N,I0)) &
 2*J=2*J)

# 5
(¬COPY<M[J+1] 8
 tl [J]<M[J+1]&
 J<N &
 J≤N &
```

```
  COUNTER=ILOG2(DIV(I,I0)) &
  COUNTER≤K &
  K=ILOG2(DIV(N,I0)) &
  J=2*I
→
  COUNTER+1≤K+1)

# 6
(¬COPY<M[J]       &
 ¬M[J]<M[J+1] &
 J<N &
 J≤N &
 COUNTER=ILOG2(DIV(I,I0)) &
 COUNTER≤K &
 K=ILOG2(DIV(N,I0)) &
 J=2*I
→
 COUNTER+1≤K+1)

# 7
(¬COPY<M[J] &
 ¬J<N &
 J≤N &
 COUNTER=ILOG2(DIV(I,I0)) &
 COUNTER≤K &
 K=ILOG2(DIV(N,I0)) &
 J=2*I
→
 COUNTER+1≤K+1)

# 8
(¬J≤N &
 COUNTER=ILOG2(DIV(I,I0)) &
 COUNTER≤K &
 K=ILOG2(DIV(N,I0)) &
 J=2*I
→
 COUNTER+1≤K+1)

*****
```

The time required to verify these verification conditions is 24 CPU seconds, using the following lemmas.

```
GOALFILE
        AXIOM DIV(@X,@X) ↔ 1;

GOALFILE
        AXIOM ILOG2(1) ↔ 0;
```

```
AXIOM ILOG2(DIV(2*@A,@B)) ↔ ILOG2(DIV(A,B)) + 1;
AXIOM ILOG2(DIV(2*@A+1,@B)) ↔ ILOG2(DIV(A,B)) + 1;
GOAL ILOG2(@X)+1 ≤ILOG2(@Y) SUB 2*X ≤ Y;
```

In these examples here, and in many others, it turns out that termination (and indeed accurate time bounds) are much easier to verify than the intended properties of the output. If, as seems frequently to be the case, the halting of the program is straightforward (and is not the "real" verification problem) this method of virtual programming presents an easy and natural way to obtain a quick check of termination and a verification of time estimates.

REFERENCES

[Clint & Hoare] Clint, M. and C. A. R. Hoare,
Program Proving : Jumps and Functions,
Acta Informatica, Vol. 1, pp.214-224, 1972.

[Clint] Clint, M.,
Program Proving : Coroutine,
Acta Informatica, Vol. 2, pp.50-63, 1973.

[Cook] Cook, S. A.,
Axiomatic and Interpretive Semantics for an Algol Fragment,
. Technical Report 79, University of Toronto, 1975.

[Deutsch] Deutsch, L. P.,
An Interactive Program Verifier,
Ph.D. thesis, University of California, Berkeley, 1973.

[Dijkstra] Dijkstra, E.,
Guarded statements, CACM, forthcoming.

[Farmwald] Farmwald, M.,
Private communications, A I Laboratory, Stanford University.

[Floyd] Floyd, Robert W.,
Assigning Meanings to Programs,
Proc. Symp. Appl. Math. Amer. Math. Soc., Vol 19, 1967, pp.19-32.

[von Henke & Luckham] von Henke, F. W. and D. C. Luckham,
A Methodology for Verifying Programs,
Proceedings of International Conference of Reliable Software, IEEE,
pp.156-164, 1975.

[Hoare69] Hoare, C. A. R.,
An Axiomatic Basis for Computer Programming,
CACM, Vol. 12, 1969, Oct., pp.576-580.

[Hoare71] Hoare, C. A. R.,
Procedures and Parameters: an axiomatic approach,
Symposium on Semantics of Algorithmic Languages,
E. Engeler(ed.), Springer-Verlag, 1971, pp.102-116.

[Hoare & Lauer] Hoare, C. A. R. and P. E. Lauer,
Consistent and Complementary Formal Theories of the Semantics of Programming Languages,
Acta Informatica, Vol. 3, 1974, pp.135-153.

[Hoare & Wirth] Hoare, C. A. R. and N. Wirth,
An Axiomatic Definition of the Programming Language PASCAL,
Acta Informatica, Vol. 2, 1973, pp.335-355.

[Igarashi, London & Luckham] Igarashi, S., R. L. London, and D. C. Luckham,
Automatic Program Verification I: Logical Basis and Its Implementation,
AIM-200, Stanford Artificial Intelligence Project, Stanford University, 1972.
and Acta Informatica, Vol.4, 1975, pp.145-182.

[King] King, J. C.
A Program Verifier,
Ph.D. thesis, Carnegie-Mellon University, 1969.

[Knuth] Knuth, D. E.,
Fundamental Algorithms,
The Art of Computer Programming, Vol.1, pp.17919,
Addison-Wesley, Reading, 1973.

[Sites] Sites, Richard L.,
Proving that Computer Programs Terminate Cleanly,
Computer Science Department Report CS 418,
Stanford University, May 1974.

[Suzuki a] Suzuki, Norihisa,
Verifying Programs by Algebraic and Logical Reduction,
Proceedings of Intl. Conf. on Reliable Software,
IEEE, 1975, pp.4739481.

[Suzuki b] Suzuki, Norihisa,
Automatic Verification of Program with Complex Data Structure

Ph.D. Thesis , Stanford University, 1975.

[Waldinger & Levitt] Waldinger, R. J. and K. N. Levitt,
Reasoning about Programs,
Technical Note 86, SRI Project 2245, Stanford Research Institute.