AD-767 331

AUTOMATIC PROGRAM VERIFICATION I: A LOGICAL BASIS AND ITS IMPLEMENTATION

Shigeru Igarashi, et al

Stanford University

Prepared for:

Advanced Research Projects Agency
National Aeronautics and Space Administration

May 1973

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-200

STAN-CS-73-365

AUTOMATIC PROGRAM VERIFICATION I: A LOGICAL BASIS AND ITS IMPLEMENTATION

BY

SHIGERU IGARASHI
RALPH L. LONDON
AND
DAVID C. LUCKHAM

SUPPORTED BY

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
CONTRACT NSR 05-020-500
AND
ADVANCED RESEARCH PROJECTS AGENCY

ARPA ORDER NO. 457

MAY 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Unclassified
Security Classification

AD-767331

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Stanford University Dept. of Computer Science Stanford, California 94305	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE

AUTOMATIC PROGRAM VERIFICATION I: A LOGICAL BASIS AND ITS IMPLEMENTATION

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

technical report, May, 1973

5. AUTHOR(S) (First name, middle initial, last name)

Shigeru Igarashi, Ralph L. London and David C. Luckham

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
May 1973	approx. 50	

8a. CONTRACT OR GRANT NO ARPA-SD-183	9a. ORIGINATOR'S REPORT NUMBER(S) STAN-CS-73-365
b. PROJECT NO	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) AIM200
d.	

10. DISTRIBUTION STATEMENT

Releasable without limitations on dissemination.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

Defining the semantics of programming languages by axioms and rules of inference yields a deduction system within which proofs may be given that programs satisfy specifications. The deduction system herein is shown to be consistent and also deductive complete with respect to Hoare's system. A subgoaler for the deductive system is described whose input is a significant subset of Pascal Programs plus inductive assertions. The output is a set of verification conditions or lemmas to be proved. Several non-trivial arithmetic and sorting programs have been shown to satisfy specifications by using an interactive theorem prover to automatically generate proofs of the verification conditions. Additional components for a more powerful verification system are under construction.

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6801

Unclassified
Security Classification

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-200

MAY 1973

COMPUTER SCIENCE DEPARTMENT
REPORT STAN-CS-73-365

USC INFORMATION SCIENCES INSTITUTE
REPORT ISI/RR-73-11

AUTOMATIC PROGRAM VERIFICATION I: A LOGICAL BASIS AND ITS IMPLEMENTATION

by

Shigeru Igarashi
Ralph L. London
and
David C. Luckham

ABSTRACT: Defining the semantics of programming languages by axioms and rules of inference yields a deduction system within which proofs may be given that programs satisfy specifications. The deduction system herein is shown to be consistent and also deduction complete with respect to Hoare's system. A subgoaler for the deduction system is described whose input is a significant subset of Pascal programs plus inductive assertions. The output is a set of verification conditions or lemmas to be proved. Several non-trivial arithmetic and sorting programs have been shown to satisfy specifications by using an interactive theorem prover to automatically generate proofs of the verification conditions. Additional components for a more powerful verification system are under construction.

Authors' addresses: Igarashi, Research Institute for Mathematical Sciences, Kyoto University, Kyoto 606, Japan; London, USC Information Sciences Institute, 4676 Admiralty Way, Marina Del Rey, California 90291; Luckham, Computer Science Department, Stanford University, Stanford, California 94305.

This research is supported by the Advanced Research Projects Agency under Contracts SD-183 and DAHC 15-72-C-0308, and by the National Aeronautics and Space Administration under Contract NSR 05-020-500.

AUTOMATIC PROGRAM VERIFICATION I:
A LOGICAL BASIS AND ITS IMPLEMENTATION

by

Shigeru Igarashi, Ralph L. London, and David C. Luckham

1. INTRODUCTION

Verifying that a computer program is correct has been discussed in many recent publications, for example [Hoare 1969, King 1969, McCarthy and Painter 1967]. The "correctness problem" or "verification problem" has become popular essentially because it represents a significant first stsp towards writing programs that can be guaranteed to do what their authors intended. Thsre are eeveral different interpretations of exactly what it means. Here, we adopt the point of view that a program has been "verified" when it is proved within a system of logic to be consistent with documentation, i.e. a statement of what it is supposed to do. Our discussion is restricted to programs that can be written in a very precise modern programming langage, Pascal [Wirth 1971]. Of course, we do not deal with all Pascal programs, but with a subset that is rich enough to include published algorithms such as FIND [Hoare 1971b], TREESORT3 [Floyd 1964], and a simple compiler [McCarthy and Painter 1967]. Since Pascal is an Algol-like language we expect that what is done here can be repeated without much effort for Algol or other euch languages. We adopt a DOCUMENTATION LANGUAGE that is roughly speaking the language of quantified Algol Boolean expressions, (i.e. first-order number theory with definitional extension and some notational conveniences). It does not contain any constructs for representing such notions as tense (time depsndency), possibility (can do), etc. that may well prove useful in describing programs. So the documentation language is a slight extension of what programmers normally use to state those conditions on computations that control their programs. Statements of the documentation language are called ASSERTIONS. A documented program is, for us, a Pascal program in which assertions have been placed between its statements at certain points. We refer to such programs with documentation as ASSERTED PROGRAMS.

The general idea of how to go about verifying an asserted program is to reduce this problem to questions about whether certain associated logical conditions (henceforth called VERIFICATION CONDITIONS) are true of (i.e. theorems in) various standard first-order theories. The usual method of reduction [Floyd 1967] involvss enumerating all possible paths between assertions in the program and then computing a verification condition for each path in terms of operations and assertions on that path; these verification conditions must then be proved. See London [1972] for a bibliography of existing programs for generating verification conditions.

However, in the case of Pascal, a rigorous definition of the semantics has been given in terms of axioms and rules of inference that must be valid for each syntactic constructor; this is contained in the recent work of Hoare and Wirth [1972]. This approach to defining the semantics of a programming language yields a deduction system in which proofs that programs satisfy specifications may be given (see Hoare [1969,1971a]). Such proofs, of course, depend on the truth of first-order conditions, or to put it another way, standard first-order theories are sub-systems of the deduction system for Pascal. For the sake of illustration, Example 1 shows a proof in Hoare's system that the program in step 13 computes the quotient q and remainder r of the inputs x and y. The rules of inference used here are the rules in Table 1 (Section 3.1) and the iteration rule below. The logical conditions assumed by this proof are labeled "lemma".

Iteration: $\dfrac{P \wedge Q\{A\}P,\ P \wedge \neg Q \supset R}{P\{\text{while } Q \text{ do } A\}R}$

1.	$true \rightarrow x = x + y * 0$	Lemma 1
2.	$x = x + y * 0 \{r \leftarrow x\}\ x = r + y * 0$	C1
3.	$x = r + y * 0 \{q \leftarrow 0\}\ x = r + y * q$	C1
4.	$true \{r \leftarrow x\}\ x = r + y * 0$	C5 (1,2)
5.	$true \{r \leftarrow x;\ q \leftarrow 0\}\ x = r + y * q$	C7 (4,3)
6.	$x = r + y * q \wedge y \le r \rightarrow x = (r-y) + y * (1+q)$	Lemma 2
7.	$x = (r-y) + y * (1+q) \{r \leftarrow r-y\}\ x = r + y * (1+q)$	C1
8.	$x = r + y * (1+q) \{q \leftarrow 1+q\}\ x = r + y * q$	C1
9.	$x = (r-y) + y * (1+q) \{r \leftarrow r-y;\ q \leftarrow 1 + q\}$ $x = r + y * q$	C7 (7,8)
10.	$x = r + y * q \wedge y \le r \{r \leftarrow r-y;\ q \leftarrow 1+q\}$ $x = r + y * q$	C5 (6,9)
11.	$x = r + y * q \wedge \neg y \le r \rightarrow \neg y \le r \wedge x = r + y * q$	Lemma 3
12.	$x = r + y * q \{\text{while } y \le r \text{ do}(r \leftarrow r-y;\ q \leftarrow 1 + q)\}$ $\neg y \le r \wedge x = r + y * q$	Iteration (10,11)
13.	$true\{((r \leftarrow x;\ q \leftarrow 0);\ \text{while } y \le r \text{ do } (r \leftarrow r-y;\ q \leftarrow 1+q))\}$ $\neg y \le r \wedge x = r + y * q$	C7 (5,12)

EXAMPLE 1: FORMAL VERIFICATION OF QUOTIENT-REMAINDER PROGRAM

It is possible to generate the verification conditions for an asserted program merely by using a subgoaler for the deduction system. EXAMPLE 2 shows how such a subgoaler works on the Quotient-Remainder program of Example 1; it simply searches for a rule of inference which has the current goal as its conclusion and then generates the premisses of the rule as subgoals.

Goal. $\mathrm{true}\{r \leftarrow x;\ q \leftarrow 0;\ \textbf{assert}\ x = r + y * q;$
$\quad \textbf{while}\ y \le r\ \textbf{do begin}\ r \leftarrow r-y;$
$\quad q \leftarrow 1+q\ \textbf{end}\}\ \neg(y \le r) \wedge (x = r + y * q)$

Subgoal 1. $\mathrm{true}\{r \leftarrow x;\ q \leftarrow 0\}\ x = r + y * q$ C7 (Goal)

Subgoal 2. $x = r + y * q\ \{\textbf{while}\ y \le r\ \textbf{do begin}\ r \leftarrow r-y;$
$\quad q \leftarrow 1+q\ \textbf{end}\}\ \neg(y \le r) \wedge (x = r+y*q)$
C7 (Goal)

Lemma 3. $(x = r + y * q) \wedge \neg (y \le r) \rightarrow \neg (y \le r) \wedge (x = r+y*q)$
Iteration (Subgoal 2)

Subgoal 3. $(x = r+y*q) \wedge (y \le r)\{r \leftarrow r-y; q \leftarrow 1+q\}\ x = r+y*q$
Iteration (Subgoal 2)

Subgoal 4. $(x = r+y*q) \wedge (y \le r)\{r \leftarrow r-y\}\ x = r+y*(1+q)$
C7 (Subgoal 3),
then C1 (Subgoal 3)

Lemma 2. $(x=r+y*q) \wedge (y \le r) \rightarrow x=(r-y)+y*(1+q)$ C1 (Subgoal 4),
then C5 (Subgoal 4)

Subgoal 5. $\mathrm{true}\{r \leftarrow x\}\ x = r+y*0$ C7 (Subgoal 1),
then C1 (Subgoal 1)

Lemma 1. $\mathrm{true} \rightarrow x = x + y * 0$ C1 (Subgoal 5),
then C5 (Subgoal 5)

EXAMPLE 2: GENERATION OF THE VERIFICATION CONDITIONS FOR THE QUOTIENT-REMAINDER PROGRAM

Note that, for example, subgoal 4 is obtained from subgoal 3 by using C7 (composition rule) to split the compound statement at the semi-colon; Q is set to $x = r+y*(1+q)$ by applying C1 (assignment axiom) so that the other subgoal is $x = r+y*(1+q)\{q \leftarrow 1+q\}\ x = r+y*q$ which is an instance of the assignment axiom and hence is satisfied. If the first-order "lemmas" produced by the subgoaler are true of the relevant theories (in this case, number theory) then we know that there will be a proof verifying the Quotient-Remainder program in Hoare's system. These verification conditions are sufficient conditions.

This is the approach to generating verification conditions presented here. We use a simple subgoaling program for Hoare's deduction system. Although this program will accept a significant subset of Pascal programs, it is itself very simple since it does not analyze the object program explicitly but merely repeatedly applies a list of rules of inference. It is easily shown to be sound (see below), easily extended to accept additional syntax (FOR statements, new type declarations, etc.), and easily changed to take account of new definitions of the semantics. We refer to this subgoaler as VCG (Verification Condition Generator); details of its implementation are given in Section 4 and sample outputs in Section 5.

However, there are problems. At any step more than one deduction rule may be applicable to generate further subgoals. To deal with this ambiguity, we have chosen a set of deduction rules (some of them derived rules in Hoare's system) for subgoal generation which is unambiguous. We shall show that this set is deduction complete. This means that if a particular verification can be proved in Hoare's system, then VCG will produce a sufficient set of verification conditions from which such a proof may then be constructed. However, these conditions may not be provable unless the user supplies certain crucial assertions at intermediate points in his program (e.g. an invariant for each loop). Finally we also need to know that the deduction system is consistent.

Section 3 deals with these logical problems. We give a small set of axioms and deduction rules, called the CORE, from which all of Hoare's rules can be derived; some sample derivations are included. A straight-forward set theoretic model of the core is constructed; thie gives us a semantic proof of consistency of the core. The set of rules used by VCG is given and is shown to be consistent with the core and powerful enough to derive the core (hence deduction completeness). Preliminary comments, definitions and examplee concerning Pascal programs, the assertion language and asserted programs are given in Section 2.

VCG is already a useful tool. Numerous example programs have been verified by manually proving the verification conditions. More interestingly, and of more promise, VCG is intended to be the initial part of an automatic verification and debugging system. The overall plan is shown in Figure 1. Asserted programs are input to VCG. The output verification conditions are simplified relative to data filee containing relevant properties of the operators and functions in the conditions. It will become evident from the examples in Section 5 that a great deal of elementary simplification of verification conditions is both necessary and easy to do. The truth of many of the conditions will be established at the simplification stage. Next, the condition Analyzer is intended to reduce problems given to the theorem prover and to find bugs. It attempts to claeeify verification conditions according to probable method of proof and to generate simpler subproblems, and also attempts to find the "closest" similar condition that is provable when a proof of a given condition is not found. This latter kind of analysis is one method of catching bugs--finding missing assumptions in verification conditions. Currently a development of the theorem-prover of Allen and Luckham [1970] is being used successfully by J. Morales to prove conditione output by VCG for various sorting programs (see Section 5.4). This proposed system thus appears to have a good chance of being developed into something useful.

What has become evident is that VCG is not a trivial element in this type of verification system. In order to make such a system practical, the amount of documentation the user is required to supply with his program should be restricted to what would be coneidered natural for human understanding of what the program and its sub-programs do. At the moment VCG places rather more weight on documentation than we would like. However it is already easy to see how to extend VCG by adding some additional rules that will permit it to deduce intermediate documentation for itself in some cases.

```
                        --------------                  -------------
                        | DATA FILES |                  | DATA FILES|
                        --------------                  -------------
                          ^  |                              ^ |
 Input                    |  v                              | v
          ---------     --------------    -----------    ----------
          |  VCG  |     | SIMPLIFIER |    | ANALYZER |-->|THEOREM |
 ---->|   |       |---> |            |--->|          |<--|PROVER  |
          ---------     --------------    -----------    ----------
                              |                |
                              |                |
                              |                |
                              v                v
                         ----------       ----------
                         | OUTPUT |       | OUTPUT |
                         ----------       ----------
```

FIGURE 1: PLANNED AUTOMATIC VERIFICATION AND DEBUGGING SYSTEM

2. PROGRAMS WITH ASSERTIONS

2.1 PASCAL.

A comprehensive definition of Pascal is published by Wirth [1971,1972] and Hoare and Wirth [1972]. Our choice of Pascal as the programming language is motivated by the development of Hoare's deduction system and its use to define the semantics of Pascal. Pascal is an Algol-like language so a reader familiar with Algol will have no trouble understanding the examples of programs and condition generation in this paper. Thus instead of including a definition of Pascal here, we shall point out some of the main differences of concern to us between Pascal and Algol. The following example shows a program containing a procedure definition, variable declarations, a recursive function definition and a program body which calls the procedure and function; it is written first in Algol and then in Pascal.

ALGOL PROGRAM:

```
BEGIN
INTEGER ALPHA, BETA, QUOT, REM, Q, R, X, Y, I;

PROCEDURE QUOTREM(R,Q,X,Y); VALUE X, Y; INTEGER R, Q, X, Y;
BEGIN R := X; Q := 0;
        FOR I := 1 WHILE Y ≤ R DO
                BEGIN R := R - Y; Q := 1 + Q END
END;

INTEGER PROCEDURE FACT(N); INTEGER N;
BEGIN IF N = 0 THEN FACT := 1 ELSE FACT := N * FACT(N-1) END;

BETA := 3; X := 6; Y := 4;
ALPHA := FACT(BETA);
QUOTREM(QUOT, REM, X+Y, X-Y);
Q := QUOT; R := REM
END
```

PASCAL PROGRAM:

```
VAR ALPHA, BETA, QUOT, REM, Q, R, X, Y : INTEGER;

PROCEDURE QUOTREM(VAR R, Q : INTEGER; X, Y : INTEGER);
BEGIN R := X; Q := 0;
        WHILE Y ≤ R DO
                BEGIN R := R - Y; Q := 1 + Q END
END;

FUNCTION FACT(N:INTEGER) : INTEGER;
BEGIN IF N = 0 THEN FACT := 1 ELSE FACT := N * FACT(N-1) END;
```

```
BEGIN BETA := 3; X := 6; Y := 4;
        ALPHA := FACT(BETA);
        QUOTREM(QUOT, REM, X+Y, X-Y);
         Q := QUOT; R := REM
END.
```

EXAMPLE 3: A PROGRAM IN ALGOL AND PASCAL

The differences in declaring variables are unimportant for our purposes. The type of the function Is indicated after the right parenthesis in Pascal rather than before the word "PROCEDURE" in Algol. The opening "BEGIN" in Algol appears just before the main program in Pascal. In the formal parameter part of procedure and function definitions, Pascal includes the specification of the formal parameters inside the parentheses; in Algol this specification is made after the list of parameters to be called by value.

The remaining difference may be ekipped until procedures are discussed in detail later. The word "VAR" in the Pascal formal parameter part means R and Q are variable parameters. The corresponding actual parameters must be variables (and not more general expressions); assignment to R or Q in the body of the procedure affects the corresponding actual parameters. The absence of "VAR" before X and Y means X and Y are value parametere In the Algol 60 sense (representing a change in the revised Paecal from the original definition). The corresponding actual parameters must be expressions (of which a variable is a simple case). A value parameter represents a variable local to the procedure to which the value of the corresponding actual parameter is initially assigned upon activation of the procedure. Assignments to value parameters from within the procedure are permitted, but do not affect the corresponding actual parameters. (For further details of Pascal see Wirth [1971, 1972]).

At the moment VCG will accept a subset of legal Pascal programe built up from: assignment, while, conditional, and go to etatements; recursive procedure and function definitions and calls; one-dimensional arrays are allowed on either side of assignment statements.

2.2 ASSERTIONS

Assertions are conditions on the state of the computation of a program. Thus, if assertion P is placed at some point in program A, the intention is that when A is run, every time P is encountered P must be true of the current computation state of A.

Essentially, our assertion language allows assertions to contain any well-formed formula of a standard first-order theory and in addition, non-standard relations may be introduced by definitions. In practice we have adopted a slightly more useable and readable formal language for the assertions of VCG.

(i) A term in the assertion language is a Pascal expression.

(ii) Atomic assertions are either predicates (i.e. identifiers) with terms as arguments or terms.

(iii) Assertions are well-formed logical formulas constructed from atomic assertions using logical connectives and quantifiers according to the usual well-known rules.

Here are some examples:

(1) $X = Y+Z$

(2) $\neg(Y \le R) \wedge (X = R+Y*Q)$

(3) $Z*POWER(W,I) = POWER(X,Y)$

(4) $\forall K((1 \le K) \wedge (K \le N-1) \supset A[K] \le A[K+1])$&
PERMUTATION(A,A0).

The first three assertions are expressions in Pascal (and in fact Boolean expressions in Algol) and use a precedence among operators to simplify notation (below). Assertion (4) is not a Boolean expression in Algol (because it contains a quantifier) nor an expression in Pascal (because of the quantifier and implication).

The assertion language contains different connective symbols for both IMPLICATION and AND to improve readability of verification conditions. The precedence order of connectives and arithmetical operators, predicates, and quantifiers is:

1. &(and); 2. $\rightarrow$ (implies), $\supset$ (implies); 3. $=$, $\neq$, $<$, $>$, $\le$, $\ge$; 4. $\vee$, $+$, $-$; 5. $\wedge$ (and), $*$, $/$, DIV, MOD; 6. $\neg$, $\forall$, $\exists$.

This agrees with the precedence in Pascal expressions.

NOTATION: Assertions and Boolean expressions will usually be denoted by P,Q,R,S.

2.3 ASSERTED PROGRAMS

Assertions are added to programs as additional statements beginning with the special symbol ASSERT, namely

<assert statement> ::= ASSERT <assertion>

Thus an asserted program is a legal Pascal program if we imagine that the syntax of the Pascal statement is extended by adding the extra clause below to the syntax diagram of "statement" (see appendix to Wirth [1972]):

```
                ------------        ----------------
------>   (  ASSERT   )--------->  |  ASSERTION   |---->
                ------------        ----------------
```

The assertions at the entry and exit of a procedure definition, function definition, or main program have the word "ASSERT" replaced by "ENTRY" and "EXIT" respectively. Both entry and exit statements appear at the beginning of the unit.

There are some further restrictions. The basic rule about placing assertions in a source program is that every loop must contain at least one assertion. This requirement is met if there is an assertion at every iteration statement (i.e., immediately before the statement) and an assertion at every label (i.e., just after the label). Although these requirements are not a necessary condition, they are a simple and convenient sufficient condition to guarantee an assertion in every loop. An assertion is required for the exit of a program. With no loss of generality we assume a single exit. Assertions may optionally be placed anywhere else. If an assertion is missing from the entrance, VCG will assume the entry assertion "UNRESTRICTED", a synonym for "TRUE". A source program with assertions placed to meet these requirements is called an ASSERTED PROGRAM. Examples of asserted programs are given in Section 5.

NOTATION: Asserted programs will be denoted by A,B,C,D.

2.4 LOGIC OF ASSERTED PROGRAMS

We review briefly here the elements of Hoare's inference system for proving properties of programs.

STATEMENTS of the logic are of three kinds.

(i) assertions,

(ii) statements of the form P{A}Q where P,Q are assertions and A is a program or asserted program.

P{A}Q means "if P is true of the input state and A halts (or halts normally in the case that A contains a GO TO to a label not in A) then Q is true of the output state".

(iii) procedure declarations (definitions) of the form p PROC K where p is a procedure name and K is a program or asserted program (the procedure body).

There is an infinite sst of variables p,q,r,... that range over procedures. Thus undsclared procedure names occurring in statemente are free variables ranging over procedures.

A RULE OF INFERENCE is a transformation rule from a set of statements (premises, say $H_1,\ldots,H_n$) to a statement (conclusion, say K) that is always of kind (ii). Such rules are denoted by

$$\text{R1.} \qquad \frac{H_1,\ldots,H_n}{K}$$

The concept of PROOF in Hoare's system is defined in the usual way as a sequence of statements that are either axioms or obtained from previous members of the sequence by a rule. A sequence is a proof of its end statement.

We use H ||- K to denote that K can be proved by assuming H. H |- K denotes the same thing for first order logic.

Some rules have the existence of a subproof as a premise; they are of the form

$$\text{R2.} \qquad \frac{H_1,\ldots,H_n,\ I \ ||\!-\ J}{K}$$

Such rules permit deductions of assertions on recursive procedure calls.

We extend the definition of proof to include the notion of assumption or dependency. An arbitrary well-formed formula can appear in a proof sequence. But in such a case that formula is said to have a formula identical with itself as its (unique) assumption formula. Each formula in the sequence has an associated set of assumption formulas, which can be empty, and which must be empty if it is the end formula in the sequence. Each rule of inference preserves the assumptions unless specified otherwise. Thus the conclusion of a rule of the form R1 is dependent on the set of assumptions that is the set-theoretic union of the sets of assumptions of the premisses. In other words, assumptions are inherited from premisses to conclusions.

Assumptions can be discharged only if the rule is of the form R2. In this case the assumption formula designated by I can be discharged from the set of assumptions associated with the conclusion designated by K, while other assumptions are inherited.

Intuitively I ||- J means I implies J, and a free variable, say r, reads "for any r".

The rules of inference discussed in the following sections all have, with one exception, at most two premisses. Proofs may be represented in the usual way by binary trees.

SUBSTITUTION of an expression t for a variable x in an expression E is denoted by $E|^{x}_{t}$.

We note that the termination of a program A is not expressable in Hoare's system by statements of the form P{A}Q. On the other hand, non-termination can be expressed by statements such as TRUE{A}FALSE. There may be some indirect ways of constructing formulas that mean "A terminates for all inputs satisfying P", and if so, it would be nice to know for what class of programs this can be done.

REMARKS:

We presuppose a standard first-order theory, which shall be denoted by T, representing the properties of the primitive functions and predicates used in Pascal. However, our construction is uniform in that choosing different first-order theories characterizing possibly different functions and predicates does not affect the framework. A standard model of the theory T is fixed and denoted by M.

In our formal system there are three kinds of procedure names we have to distinguish:
1) Procedure names for primitive procedures. For instance a library procedure whose body is inherently written in a language of lower level belongs to this category. (It is even possible for us to regard the assignment statement as such a procedure.)
2) Procedure names for declared procedures. We regard procedure declarations as the "defining axioms" of such procedure names, which constitute nonlogical axioms in our system and shall be denoted by J. We assume J does not assign more than one procedure to a name.
3) Procedure names used in derivations. In the formal system we will use procedure names which should intuitively be regarded as "free variables", which represent arbitrary procedures. In proving metatheorems we will use a name for each declared procedure.

Besides the above, each procedure name is assumed to have "arity", so that it can represent or vary over declared procedures with, say, m variable parameters and n value parameters. Such a procedure will be called (m,n)-ary and the m (variable) parameters and the n (value) parameters will be called the left and the right parameters, respectively.

If a primitive procedure name, say q, occure in a program about which we are to prove a certain theorem, we have to either give a set of (nonlogical) axioms of the form P{q(x;y)}R or a defining axiom for q. In most cases, we shall assume that the procedure can be written in Pascal and that there is a defining axiom for it.

3. THE BASIS INFERENCE SYSTEM FOR VCG.

In this section we study the properties of the set V of axioms and rules of inference used by VCG. One of our main concerns is that the rules of inference in V should be unambiguous in the sense that only one rule is applicable to generate subgoals from any given goal. This will certainly be the case if no two rules have conclusions which have common substitution instances, a property which is true of V. The rules of V, which appear as Tabls 2 in section 3.3, are simple combinations of Hoare's original set of rulee H given in Hoare [1971a, p. 116]. Having chosen V, we must establish that it is both sound and deduction complete. We shall show first that a set C of simple rules (the CORE) is sound and that any rule in H can be derived from C. We then show that V and C are inter-derivable. We shall begin by studying the relative derivability when none of the sets of rules contains go to's or array variables. The rules H are equivalent to the following set of rules.

3.1 THE CORE RULES

The set of axioms and rules of the core is given in Table 1. Rules D3 (iteration), D7 (adaptation) of H have been omitted; D4 (alternation) has been replaced by C8 (conditional). We have added the frame axiom (C2) for procedure calls and the and-or rule (C6); Hoare's substitution rule (D6) corresponds to our left and right substitution rules.

NOTATION: x, y, z - lists of variables; p,q,r - procedure names; s, t - lists of expressions; K - procedure body; p(x;y) - denotes CALL p(x;y) where x and y are the left and right parameters of p. VAR(P) denotes the free variables of P; p(x;y) PROC K denotes a declaration of the form "PROCEDURE p(x;y); BEGIN K END".

AXIOMS

C1. assignment axioms: $P|^{x}_{t}\ \{x \leftarrow t\}P$

C2. frame axioms: P{q(x;t)}P provided $\neg(x \in \mathrm{VAR}(P))$

C3. procedure declarations: p(x;y) PROC K.

C4. logical theorems: P for all P s.t. |- P.

RULES

C5. consequence: $\dfrac{P \supset Q,\ Q\{A\}R}{P\{A\}R}$, $\dfrac{P\{A\}Q,\ Q \supset R}{P\{A\}R}$

C6. and/or: $\dfrac{P\{A\}Q, R\{A\}S}{P \wedge R\{A\}Q \wedge S}$ $\dfrac{P\{A\}Q,\ R\{A\}S}{P \vee R\{A\}Q \vee S}$

C7. composition: $\dfrac{P\{A\}Q,\ Q\{B\}R}{P\{A;B\}R}$

C8. conditional: $\dfrac{P \wedge R\{A\}Q,\ P \wedge \neg R\{B\}Q}{P\{\text{IF } R \text{ THEN } A \text{ ELSE } B\}Q}$

C9. substitution: (L) $\dfrac{P(x;y)\{q(x;y)\}Q(x;y)}{P(z;y)\{q(z;y)\}Q(z;y)}$

(R) $\dfrac{P(x;y)\{q(x;y)\}Q(x;y)}{P(x;s)\{q(x;s)\}Q(x;s)}$

SUBJECT TO THE RESTRICTIONS: (i) s does not contain members of x; (ii) members of z must be distinct and y and z are disjoint.

C10. procedure call: $\dfrac{p(x;y)\ \text{PROC}\ K(p),\ P\{r(x;y)\}Q \Vdash P\{K(r)\}Q}{P\{p(x;y)\}Q}$

where p does not occur in the proof of the right hand premiss, and r does not occur in any other assumption in that proof.

TABLE 1 C: THE CORE RULES.

In order to demonstrate that C is as "powerful" as H we show that any proof in H of P{A}Q can be transformed into a proof in C of P{A'}Q where A' is a program equivalent to A. An application of a rule R (that is not a rule in C) in the given proof is to be replaced by a derivation in C of the conclusion of R assuming the premisses of R. The transformed proof will use only rules of C and will prove essentially the same formal statement. It is clear that applications of Hoare's substitution rule (D6) can be replaced by successive applications of the left and right rules (C9). We therefore need only consider the following three rules.

(D4) Alternation:

$$\frac{P1\{A\}Q, \quad P2\{B\}Q}{\text{if } R \text{ then } P1 \text{ else } P2\{\text{if } R \text{ then } A \text{ else } B\}Q}$$

(D7) Adaptation:

$$\frac{P(a;e)\{p(a;e)\}R(a;e)}{P(a;e) \wedge \forall a(R(a;e) \supset S(a;e))\{p(a;e)\}S(a;e)}$$

(D3) Iteration:

$$\frac{P\{A\}S, \quad S \vdash \text{if } Q \text{ then } P \text{ else } R}{S\{\text{while } Q \text{ do } A\}R}$$

(a) D4 is derivable in C. Let P in the conditional rule (C8) be: if R then P1 else P2.

1. $P1\{A\}Q$, $P2\{B\}Q$ — assumptions (premissee of D4)

2. $P \wedge R \supset P1$, $P \wedge \neg R \supset P2$

3. $P \wedge R\{A\}Q$, $P \wedge \neg R\{B\}Q$ — consequence (C5) 1,2

4. if R then P1 else P2{if R then A else B}Q
   conditional (C8) 3.

(b) D7 is derivable in C.

1. $P(a;e)\{p(a;e)\}R(a;e)$ — assumption (premiss D7)

2. $\forall a(R(a;e) \supset S(a;e))\{p(a;e)\}\forall a(R(a;e) \supset S(a;e))$
   frame axiom (C2).

3. $P(a;e) \wedge \forall a(R(a;e) \supset S(a;e))\{p(a;e)\}R(a;e) \wedge$
   $\forall a(R(a;e) \supset S(a;e))$
   and rule (C6) 1,2.

4. $P(a;e) \wedge \forall a(R(a;e) \supset S(a;e))\{p(a;e)\}S(a;e)$ — C5,3.

Corresponding to any while statement "while Q do A" we can define a recursive procedure:

```
procedure whiledef (x;v);

if Q then begin A; call whiledef(x;v);end
else end
```

where x is the list of variables in A that are subject to change in the body A, and v is the list of all other variables in Q or A.

We consider a modified form of the iteration rule:

(D3') P{A}S, S ⊃ if Q then P else R

S{call whiledef(x;v)}R

(c). D3' is derivable in C.

1.	P{A}S	Assumption (premiss D3').
2.	S∧Q⊃P	Assumption (premiss D3')
3.	S∧¬Q⊃R	Assumption (premiss D3')
4.	S{call r(x;v)}R	Assumption
5.	P{A;call r(x;v)}R	C7, 1,4
6.	S∧Q{A;call r(x;v)}R	C5, 2,5
7.	S{if Q then begin A; call r(x;v);end else end}R	C8, 6,3
8.	S{call whiledef(x;v)}R	C10, 4,7

If we are given a proof in H of P{A}Q we may replace applications of D4 and D7 by the proofs (a) and (b); an application of D3 is replaced by a proof (c) of D3'. We will then have a proof in C of P{A'}Q where A' is the result of replacing each while statement in A by a call to the corresponding whiledef procedure. This is easily proved by induction on the length of the proof. Clearly A' is equivalent to A. This completes the proof that C is as powerful as H.

In the other direction, all of the core rules except the frame axiom and the and-or rule appear in H with minor differences and are easily shown to be derivable in H. Thus, to show that proofs in C can be carried out in H, we need only be concerned with eliminating C2 and C6.

Recall that a Pascal program must contain definitions of all called procedures except library procedures and there are a finite number of those. This places a finite bound on the number of different procedures that can ever be called in any computation of a program.

d. Lemma

||- TRUE{A}TRUE for any program A.

PROOF

We can construct a proof of TRUE{A}TRUE by using the rules (D1-D5) to generate subgoals starting from the goal TRUE{A}TRUE. Assume a list of variables $r_1, r_2, r_3 \ldots$ distinct from the list of procedure names that may be called in a computation of A. Subgoals are generated by applying the rules recursively as follows (D3 and D4 are equivalent to D3* and D4*):

(D2) Subgoals

$$\frac{\text{TRUE}\{A\}\text{TRUE}, \quad \text{TRUE}\{B\}\text{TRUE}}{\text{TRUE}\{A;B\}\text{TRUE}}$$

(D1) Subgoal

(D3)*

Goal

$$\frac{\dfrac{\text{TRUE}\{B\}\text{TRUE}}{\text{TRUE}\wedge P\{B\}\text{TRUE}}, \quad (\text{TRUE}\wedge\neg P)\supset\text{TRUE}}{\text{TRUE}\{\text{while } P \text{ do } B\}\text{TRUE}}$$

(D1) Subgoals

(D4)* Goal

$$\frac{\dfrac{\text{TRUE}\{B\}\text{TRUE}}{\text{TRUE}\wedge P\{B\}\text{TRUE}}, \quad \dfrac{\text{TRUE}\{C\}\text{TRUE}}{\text{TRUE}\wedge\neg P\{C\}\text{TRUE}}}{\text{TRUE}\{\text{if } P \text{ then } B \text{ else } C\}\text{TRUE}}$$

(D5) Subgoal

Goal

$$\frac{\text{TRUE}\{K(r_p)\}\text{TRUE}}{\text{TRUE}\{p(x;v)\}\text{TRUE}}$$

where K is the body of p and r_p is a unique variable to be substituted for the procedure name p in every subsequent subgoal of the goal. The procedure terminates since the subgoals in each of the rules D2 - D4 are shorter than the goals, and D5 can be applied only finitely many times since the list of procedure names that can occur is finite and one of these names is eliminated from all further subgoals of a goal to which D5 applies. The length of any subgoal branch is bounded by 2nl where n is the number of procedures that can be called by A and l is the number of statements in A. The terminal subgoals are of two kinds: TRUE{x←t}TRUE (axioms) or TRUE{r_p(x;v)}TRUE. The second kind is the assumption for an application of D5 to derive a goal below it (i.e. a goal of which it is a subgoal). Thus the final subgoal tree is a proof of TRUE{A}TRUE.

(e) P{q(x;v)}P is provable if $\neg(x \in \text{VAR}(P))$.

This follows from lemma d by applying the adaptation rule (D7):

1. TRUE{q(x;v)}TRUE — lemma d.

2. TRUE $\wedge(\forall x)$(TRUE$\supset$P){q(x;v)}P — D7,1.

3. P{q(x;v)}P — D1,2 since x does not occur in TRUE or in P(by assumption).

This establishes that C2 can always be replaced in a CORE proof by a derivation in Hoare's system. To eliminate C6 from a CORE proof we argue as follows. Suppose a given proof contains an application of AND-OR, without loss of generality, let us say it is the final deduction. We show that this occurrence of AND-OR can be either eliminated altogether or "moved up" the proof tree in the sense that it is replaced by an AND-OR application to the premisees of the premisses of the original application. This gives us a new proof containing only expressions that are in the old proof. We show further that in the second case where the rule is "moved up", if the moving up procedure is repeated the rule will never again need to be applied in any new proof to the same pair of premisses it was applied to originally. Since the given proof contains a finite number of expressions this establishes that our moving up procedure terminates with a proof in which all applications of AND-OR have disappeared.

(f) LEMMA

There is a constructive procedure for eliminating applications of the AND-OR rule from CORE proofs.

PROOF.

Suppose a given CORE proof contains one deduction by AND-OR of the form

```
                  H1,H2    H3,H4    (rule R)
                  -----    -----
D.                  I        J      (AND-OR)
                    ---------
                        K
```

where R is not AND-OR.

We give a procedure whereby either

(a) D can be replaced by a deduction of K from axiome by the rule of consequence,

or

(b) D can be replaced by

$$\text{D1.}\qquad \frac{\dfrac{H1',H3'}{I1}\qquad\dfrac{H2',H4'}{J1}}{K}\qquad \begin{array}{l}\text{(AND-OR)}\\ \text{(rule R)}\end{array}$$

In case (b), for each i, the subproof Hi' in D1 contains only statements occurring in the subproof Hi in D. Repeated application of the procedure cannot result in (AND-OR) being applied to the pair I,J of premisses again.

We note that since the same program part must appear in both premisses of an application of AND-OR, the immediately preceding rules deducing those premisses must either be the same rule R or one of them must be the rule of consequence.

Let us consider the AND-case of this rule first. We give the replacement procedure for different cases of rule R:

(i) AXIOMS.

An application of AND-OR to axioms

$$\frac{P|^{x}_{e}\ (x\leftarrow e)P \qquad\qquad R|^{x}_{e}\ (x\leftarrow e)R}{P|^{x}_{e}\wedge R|^{x}_{e}\ (x\leftarrow e)P\wedge R}$$

is eliminated entirely and replaced by the axiom

$$(P\wedge R)|^{x}_{e}\ (x\leftarrow e)P\wedge R$$

Applications of AND-rule to frame axioms are eliminated similarly.

(ii) CONSEQUENCE.

An occurrence of AND-OR of the form

$$\frac{\dfrac{P\{A\}Q1,\ Q1\supset Q}{P\{A\}Q}\quad ,\quad R\{A\}S}{P\wedge R\{A\}Q\wedge S}$$

is replaced by

$$\frac{\dfrac{P\{A\}Q1,\ R\{A\}S}{P\wedge R\{A\}Q1\wedge S} \quad , \quad Q1\wedge S\supset Q\wedge S}{P\wedge R\{A\}Q\wedge S}$$

The other cases (omitted) are similar.

(iii) WHILE

$$\frac{\dfrac{P\wedge U\{A\}P,\ (P\wedge\neg U)\supset Q}{P\{\text{while } U \text{ do } A\}Q} \qquad \dfrac{R\wedge U\{A\}R,\ (R\wedge\neg U)\supset S}{R\{\text{while } U \text{ do } A\}S}}{P\wedge R\{\text{while } U \text{ do } A\}Q\wedge S}$$

is replaced by

$$\frac{\dfrac{P\wedge U\{A\}P,\ R\wedge U\{A\}R}{(P\wedge R)\wedge U\{A\}(P\wedge R)} \quad , \quad (P\wedge R)\wedge\neg U\supset(Q\wedge S).}{P\wedge R\{\text{while } U \text{ do } A\}Q\wedge S}$$

(iv) CONDITIONAL

$$\frac{\dfrac{P\wedge U\{A\}Q,\ P\wedge\neg U\{B\}Q}{P\{\text{if } U \text{ then } A \text{ else } B\}Q,} \quad \dfrac{R\wedge U\{A\}S,\ R\wedge\neg U\{B\}S}{R\{\text{if } U \text{ then } A \text{ else } B\}S}}{P\wedge R\{\text{if } U \text{ then } A \text{ else } B\}Q\wedge S}$$

is replaced by

$$\frac{\dfrac{P\wedge U\{A\}Q,\ R\wedge U\{A\}S}{(P\wedge R)\wedge U\{A\}Q\wedge S} \quad , \quad \dfrac{P\wedge\neg U\{B\}Q,\ R\wedge\neg U\{B\}S}{(P\wedge R)\wedge\neg U\{B\}Q\wedge S}}{P\wedge R\{\text{if } U \text{ then } A \text{ else } B\}Q\wedge S}$$

Clauses for Composition and Substitution are similar to (iii) and (iv) and are omitted.

(v) PROCEDURE CALL

Procedure p has body K(p).

$$\frac{P\{r\}Q \ \Vdash\ P\{K(r)\}Q}{P\{p\}Q} \quad , \quad \frac{R\{r\}S \ \Vdash\ R\{K(r)\}S}{R\{p\}S}$$

```
        ------------------------------------------
                        P∧R{p}Q∧S
```

is replaced by

```
        P{r}Q||-P{K(r)}Q       R{r}S||-R{K(r)}s
        ----------------       ----------------
            P{r2}Q                  R{r2}S
          [subproof]              [subproof]
          P{K(r2)}Q        ,     R{K(r2)}S
          ----------------------------------
  P∧R{r2}Q∧S ||-    P∧R{K(r2)}Q∧S
  -------------------------------
              P∧R{p}Q∧S
```

This last transformation rule requires a word of explanation. In the replacement, the AND-OR rule has been "pushed up" and applied to assertions on K(p) instead of assertions on call p. The procedure call rule is now applied to P∧R{K(r2)}Q∧S so that the relevant assumption is P∧R{r2}Q∧S. Subproofs for P{K(r2)}Q and R{K(r2)}S have to be appended; the given procedure rule applications ensure the existence of these subproofs. For example, we know there is a subproof of P{K(r)}Q from the assumption P{r}Q; an application of the CALL rule allows us to deduce P{r2}Q, where r2 is a new name for procedure p. The assumption P{r}Q is discharged at this point. We then repeat the subproof again with r2 replacing r everywhere. However, no assumption is necessary in this repetition since P{r2}Q is proved. Thus, the complete subproof trees for the premisses of the new AND-OR application contain copies of the given auxilliary subproofs at "assumption nodes". The statements in each new tree are exactly those of the old tree except possibly for r2 in place of r. If the replacement procedure is applied to this new subproof of P∧R{K(r2)}Q∧S, the AND-OR rule need not be applied to the same pair of hypotheses (with r2 for p) again since P∧R{r2}Q∧S is now assumed true.

This completes the description of the replacement procedure for AND; the OR case contains almost identical clauses except that the replacements in cases (iii) and (iv) contain intermediate applications of consequence: (P∨R)∧U⊃(P∧U)∨(R∧U).

We note that Lemma f shows also that the AND-OR rule can also be omitted from the CORE. In the presence of the other core rules, ADAPTATION may be replaced by the FRAME axioms. The previous discussion may be summarized by the following theorem:

g. THEOREM

If ||- P{A}Q then P{A'}Q is provable from the CORE where A' is equivalent to A. Conversely if P{A}Q is provable from the CORE then ||- P{A}Q.

3.2 A MODEL FOR THE CORE

We assume given a standard model M for the theory T of the true Boolean expressions of Pascal and a set J of procedure definitions. Essentially M is the standard model for arithmetic possibly augmented by standard models for data types other than the integers. The details of M itself do not concern us. We show how to extend M to a model M* for the CORE.

To simplify the notation we assume a fixed ordering of the variables $x_1, x_2, x_3, \ldots$. This allows us to represent computation state vectore over the domain D of M by infinite sequences of elements of D, a= $\langle a_1, a_2, a_3 \ldots\rangle$. D* shall denote the set of all such sequences. Intuitively, state a assigns the value or interpretation a_i to x_i; this is denoted by $(x_i)_I$. The interpretation or value t_I of Boolean expressions t is defined in the usual way from standard interpretation of the primitives +,*,etc. The value of t_I applied to state a will be denoted by $t_I(a)$. A Boolean expreseion of n variables, say $P(x_1,\ldots,x_n)$, is interpreted in M as a subset P_M of D^n. Thus $P(x_1,\ldots,x_n)$ is true for the state vector a if $\langle a_1 \ldots, a_n\rangle \epsilon P_M$.

This allows us to extend the interpretation of $P(x_1,\ldots x_n)$ to D*:

$$P(x_1,\ldots,x_n)_I = \{a | \langle a_1,\ldots,a_n\rangle \epsilon P_M\}.$$

Moreover, the interpretation of substitution instances by definition satisfies:

$$a\epsilon (P(x_1,\ldots,x_n)|_e^{xi})_I \iff \langle a_1,\ldots,a_{i-1}, e_I(a), a_{i+1} \ldots\rangle \epsilon P(x_1 \ldots x_n)_I .$$

The interpretation of an (m,n)-ary procedure is a partial function f of the type $N^m \times D^n \rightarrow (D^* \rightarrow D^*)$ having the following properties:

1) Frame property:

$$(f(i(1),\ldots,i(m);c_1,\ldots,c_n)(a))_j = a_j ,$$

j is different from i(k) for any k such

that $1 \le k \le m$.

2) Substitution property:

$$(f(i(1),\ldots,i(m);c_1,\ldots,c_n)(a))_{i(k)}$$
$$= (f(j(1),\ldots,j(m);\ c_1,\ldots,c_n)(a))_{j(k)},$$
$1 \le k \le m$.

The definition of f proceeds as follows.

We define by cases the computation sequence $F(A,a)$ of program A relative to M given input a as follows.

If a is an infinite state vector, then:

(i) $F(x_i \leftarrow e,a) = \langle a_1,\ldots,a_{i-1},e_I(a),a_{i+1},\ldots\rangle$

(ii) $F(A;B,a) = F(A,a) \circledast F(B,U(A,a))$

(iii)
$$F(\text{if } P(x_1,\ldots x_n) \text{ then } A \text{ else } B,a) = \begin{cases} F(A,a) & \text{if } \langle a_1,\ldots a_n\rangle \in P_I \\ F(B,a) & \text{otherwise.} \end{cases}$$

(iv) $F(q(z;t),a) = a \circledast F(K(z;t),a)$ where J contains a defining axiom for q of the form "q(x;v) PROC K(x;v)" and K(z;t) is obtained by sutstituting the actual parameters z,t for the formal parameters x,v.

Here $a \circledast b$ is the sequence obtained by appending b onto the end of a.

$$U(A,a) = \begin{cases} \text{end state of } F(A,a) \text{ if } F(A,a) \text{ is finite} \\ \text{undefined otherwise.} \end{cases}$$

The interpretation of program A is now defined:

$$A_I = \{\langle a,b\rangle \mid U(A,a) = b\}$$

and M is extended to M* by adding the function A_I for each Pascal CORE program A.

We can now say when a statement of the form P(A)Q is true in M* (denoted by M* |= P(A)Q):

$$M* \models P\{A\}Q \iff A_I(P_I) \subset Q_I .$$

Finally, a statement $S(r_1,\ldots,r_m)$ with assumptions $A_1(r_1,\ldots,r_m),\ldots,$ $A_n(r_1,\ldots,r_m)$ where $r_1,\ldots,r_m$ are free procedure variables, is true in M* if and only if the following condition holds:

> If $A_1(p_1,\ldots,p_m),\ldots A_n(p_1,\ldots,p_m)$ are true for any declared procedure names $p_1,\ldots,p_m$ from J, each p_i having the same arity as r_i $(1 \le i \le m)$, then $S(p_1,\ldots,p_m)$ is true.

Here are some simple properties of this model:

(i) If the range of A_I is empty then for any P and Q, $M* \models P\{A\}Q$

(ii) If $M* \models P\{K(q)\}Q$ then $M* \models P\{q\}Q$ where K is the body of procedure q.

(iii) If p PROC K(r) and q PROC K(s) and $r_I \subset s_I$ then $p_I \subset q_I$.

(iv) A Boolean assertion is true in M* if and only if its universal closure is true in M.

To show that M* is a model for the CORE we will show that the axioms are true in M* and that each of the rules of inference preserves truth (i.e. if the premisses of the rules are true in M* then so also are the conclusions). For simplicity we consider examples of the axioms and rules in which the statements have one free variable (three variables for the substitution rule) and in which the premisses do not have governing assumptions except in the case of the recursion rule; the argument for the general case is identical.

Consider first a typical assignment axiom $P(e)\{x_1 \leftarrow e\}P(x_1)$. We note that $(x_1 \leftarrow e)_I = \{\langle a,b\rangle : b = \langle e_I(a), a_2, a_3, \ldots\rangle\}$, and that $a \in P(e)_I \iff \langle e(a), a_2, \ldots\rangle \in P(x_1)_I$. Thus $(x_1 \leftarrow e)_I(P(e)_I) \subset P(x_1)_I$ so that the assignment axiom is true in M*.

The frame axioms are clearly true in M*: if P does not contain x_1, say, and a,b differ only at the first position, then $a \in P_I \iff b \in P_I$. If $q(x_1,v)$ changes only the value of x_1 then $q_I(P_I) \subset P_I$.

Logical theorems are true in M^* since they are true in M. Procedure declaration axioms are aseumed to be in J.

We consider next the rules of inference. The fact that Coneequence, Composition and Conditional all preserve truth in M^* can be shown by elementary set theoretic arguments on the interpretations of Boolean expressions and programs. Simply note that if $P \supset Q$ is true in M^* then $P_I \subset Q_I$, that $(P \wedge R)_I = P_I \cap R_I$, and that $\neg R_I = D^* - R_I$.

The arguments are as follows:

CONSEQUENCE: If $P_I \subset Q_I$ and $A_I(Q_I) \subset R_I$ then $A_I(P_I) \subset R_I$.

COMPOSITION: If $A_I(P_I) \subset Q_I$ and $B_I(Q_I) \subset R_I$ then $B_I(A_I(P_I)) \subset R_I$.

CONDITIONAL: If $A_I(P_I \cap R_I) \subset Q_I$ and $B_I(P_I \cap \neg R_I) \subset Q_I$ then (if R then A else B)$_I(P_I) \subset Q_I$

SUBSTITUTION

Consider the case when the procedure $g(x_1, x_2; x_3)$ has two left parameters and one right parameter since this is sufficiently general. Let q have body K. Assume that x_1 and x_2 are the only variables whose values can be changed by K_I, and that x_3 is the only value that its computation depends on. We require a eimple lemma which may be proved by induction on the composition of K.

h. LEMMA.

For any a if $K(x_1, x_2; x_3)_I(a) = b$ and $K(x_i\ x_j; x_3)_I(a) = c$ then $b_1 = c_i$ and $b_2 = c_j$ provided $i \ne j \ne 3$.

Let f,g be partial functions mapping D^* into D such that $K(x_1, x_2; x_3)_I(a) = \langle f(a_3), g(a_3), a_3 \ldots \rangle$ and hence also $K(x_4, x_5; x_3)_I(a) = \langle a_1, a_2, a_3, f(a_3)\ g(a_3), \ldots \rangle$. If the premisses of the substitution rule are true, then:

$$a \in P(x_1\ x_2\ x_3) \text{ implies } \langle f(a_3), g(a_3), a_3, \ldots \rangle \in Q(x_1\ x_2\ x_3)_I$$

This is equivalent to:

$\langle a_1, a_2, a_3\rangle \in P_M$ implies $\langle f(a_3), g(a_3), a_3\rangle \in Q_M$.

Suppose $b \in P(x_4, x_5; x_3)_I$ so that $\langle b_4, b_5, b_3\rangle \in P_M$.

Then $\langle f(b_3), g(b_3), b_3\rangle \in Q_M$ and this implies that $K(x_4, x_5; x_3)_I$ $(b) \in Q(x_4, x_5; x_3)_I$. So the conclusion of the L-rule is true. On the other hand, if $b \in P(x_1, x_2; s(x_3))$ then $\langle b_1, b_2, s_I(b_3)\rangle \in P_M$ and therefore $\langle f(s_I(b_3)), g(s_I(b_3)), b_3\rangle \in Q_M$.

By the lemma above,

$K(x_1, x_2; s(x_3))_I$ $(b) = \langle f(s_I(b_3)), g(s_I(b_3)), b_3\rangle$ so that the conclusion of the R-rule is also true.

For each of the previous rules we have shown that truth in M* is preserved.

The case of the recursive procedure call rule is more complicated and depends on the elementary properties of M* stated above.

PROCEDURE CALL

We prove that any proof containing applications of the procedure call rule proves a statement true in M* if all premisses of the proof are true in M*. Our proof is by induction on the number n of applications of the call rule.

Clearly the case n=0 is already proved. Therefore, assume it is proved for proofs containing n call rule applications, and consider the last application in a tree with n+1. Suppose this has P(x;v) {p(x;v)} Q(x;v) as conclusion.

We may assume

I. if M* |= P(x;v) {r(x;v)} Q(x;v)
then M* |= P(x;v) {K(r)} Q(x;v), for any procedure name r,

since the subproof of the premiss of this final application can itself contain at most n occurrences of the call rule.

Let us define a sequence of procedures from p:

II. p0(x;v) PROC K(LOOP),
$p_{m+1}(x;v)$ PROC K(pm)

where LOOP is a procedure that never halts.

CLAIM: For all m, $M*|=P(x;v)\{pm(x;v)\}Q(x;v)$.

PROOF: By induction on m. Clearly the claim is true for m=0 by property (i)and I above.

Suppose $M*|=P\{pm\}Q$. Then, substituting pm for r in (I) we have $M*|=P\{K(pm)\}Q$. Therefore $M*|=P\{pm+1\}Q$ by property (ii). This proves the claim.

Next, we note that p_I is the least upper bound of the sequence $\{(pm)_I\}$:

(1) $(p0)_I \subset (p1)_I \subset (p2)_I \subset ....$

(2) For all i $(pi)_I \subset p_I$.

These follow by induction using property (iii).

(3) For any a, if $p_I(a)$ is defined there is an m such that $p_I(a) = (pm)_I(a)$.

This is so because $U(p,a) = U(pm,a)$ for any m such that $m>|F(p,a)|$, the length of $F(p,a)$.

From the claim and these facts we conclude $p_I(P_I) \subset Q_I$, so that indeed

$M*|=P(x;v)\{p(x;v)\}Q(x;v)$.

Thus we have established the following soundness theorem:

(i) THEOREM If $P\{A\}Q$ is provable in the CORE then $P\{A\}Q$ is true in M*.

3.3 RULES FOR VCG

The rules V used by VCG to generate subgoals and ultimately produce verification conditions are simple combinations of the CORE rules. There are two additions: an extension to the assignment axiom for the case when assignment is made to an array element, and a rule for go to statements provided the corresponding labels are in the same procedure (or block). A rule for array assignments was given in King [1969] and the addition of a go to rule to Hoare's system is considered in Clint and Hoare [1972]. The extended systems C and H remain relatively sound and still have the same deductive power (i.e.

Theorem (g) still holds). The rules for VCG are given in Table 2. It is easily checked that the set is unambiguous in that no two conclusions have a common substitution instance.

V1. SIMPLE ASSIGNMENT

$$\frac{P\{A\}Q(e)}{P\{A;x\leftarrow e\}Q(x)}$$

V2. ARRAY ASSIGNMENT

$$\frac{P\{A\}R(\text{if } i=j \text{ then } e \text{ else } B[i]}{P\{A;B[j]\leftarrow e\}R(B[i])}$$

V3. CONSEQUENCE

$$\text{(i)}\ \frac{P\supset Q}{P\{Null\}Q}\ ,\qquad \text{(ii)}\ \frac{P\{A\}Q,Q\supset R}{P\{A;Q\}R}\ ,$$

$$\text{(iii)}\ \frac{P\{A\}Q\supset R}{P\{A;Q\text{-}if\}R}$$

V4. ITERATION

$$\frac{P\{A\}R,\ R\wedge S\{B\}R,\ R\wedge\neg S\supset Q}{P\{A;R;\text{while } S \text{ do } B\}Q}\qquad \text{where } R \text{ is an assertion}$$

V5. CONDITIONAL

$$\frac{P\{A;Q\text{-}if;B\}R,\ P\{A;\neg Q\text{-}if;C\}R}{P\{A;\text{if } Q \text{ then } B \text{ else } C\}R}$$

V6. GOTO

$$\frac{P\{A\}ASSERT(L)}{P\{A;GOTO\ L\}Q}$$

V7. PROCEDURE CALL

$$U(x;v)\{q(x;v)\}W(x;v)\ ||\!-\ \frac{P\{A\}U(a;e)\wedge\forall a(W(a;e)\supset R)}{P\{A;q(a;e)\}R}$$

V8 PROCEDURE DECLARATION

$$\frac{P\{q(x;v)\}R \;||-\; P\{A\}R}{P\{\text{procedure } q(x;v);A\}R}$$

NOTATION:

P,Q,R,S are Boolean Assertions. Null denotes the empty program. Q(e) denotes the substitution of e for x in Q(x). B[i] denotes the i^{th} element in array B. In each of the rules A can be Null. Q-if denotes a "marked" Boolean assertion Q.

TABLE 2
V:RULES OF VCG

The rules in Table 2 are stated in the form in which they are used to generate subgoals. Thus, for example in the case of the assignment rule V1, the axiom Q(e){x←e}Q(x) is omitted from the premisses since it is true and therefore not generated as a subgoal. The composition rule is not used to generate subgoals (it would be a source of ambiguity) but is included in the other rules. VCG does not require assertions at conditional statements. It "marks" the conditional tests in the subgoals of the conditional rule, and uses them as assertions that permit a slightly different rule of consequence. The normal rule of consequence, V3(ii) would usually lead to a verification condition of the form Q⊃R' where R' is some formula involving R. Most likely the proof of R' would depend on the premiss P and in such a case Q⊃R' is unlikely to be provable. (See examples 3 and 5, Section 5).

It should be clear that any statement that can be proved in V can be proved in C. More precisely:

(j) REMARK

If V||-P{A}Q where A is a program with intermediate assertions then C||-P{A'}Q where A' is an equivalent program without the intermediate assertions.

The converse of remark (j) implies the deduction completeness of V. To prove the converse, first derive from V the composition rule (C7) by an induction argument on the statement length of B, the statement following the ";". Rules C1, C3, C4, C5, and C10 are straightforward to derive. Lemma f shows that C6 is directly derivable in C. It remains to derive C2, C8, and C9.

(C2)	Lemma d holds in V as is easily checked.		
	1.	TRUE{q(x;v)}TRUE	Lemma d
	2.	P → (TRUE ∧ ∀x(TRUE → P))	¬(x∈VAR(P))
	3.	P{null}TRUE ∧ ∀x(TRUE → P)	V3i (2)
	4.	P{q(x;v)}P	V7 (1,3)

(C8)	1.	P∧Q{B}R	P∧¬Q{C}R	Given
	2.	P→(Q→P∧Q)	P→(¬Q→P∧¬Q)	Lemmas
	3.	P{null}Q→P∧Q	P{null}¬Q→P∧¬Q	V3i (2)
	4.	P{Q-if}P∧Q	P{¬Q-if}P∧¬Q	V3iii (3)
	5.	P{Q-if;B}R	P{¬Q-if;C}R	C7 (4,1)
	6.	P{if Q then B else C}R		V5 (5)

(C9)	1.	P(x;v){q(x;v)}R(x;v)	Given
	2.	P(a;e) → P(a;e) ∧ ∀a(R(a;e) → R(a,e))	Lemma
	3.	P(a;e){q(a;e)}R(a;e)	V7 (1,2)

4. DESCRIPTION OF VCG

4.1 COMMENTS ON THE RULES

Array assignment and go to

The rule V2 for array assignment includes the usual conditional substitution operation. This rule is equivalent theoretically to the techniques proposed and implemented by King [1969] in that equivalent verification conditions result. Our rule makes the conditional expressions explicit while at the same time trying to keep the case analysis under control. Though our rule enables us to verify programs involving array assignment, we cannot state which array assignment method is preferable.

The go to rule (V6), following Clint and Hoare [1972], is for simple go to statements. By "simple" we mean jumps which stay, for example, within the current block or procedure definition. The rule is included so that a useful, but restricted class of go to statements could be processed.

Procedures

H and hence V place several restrictions on the definition and use of procedures. First, procedures may contain no global variables. This is only a conceptual restriction; Hoare and Wirth [1972] introduce the notion of "implicit parameter" which makes each global variable into a parameter, at least notationally. Second, a key distinction is made between variable (VAR) and value (non-VAR) parameters. In brief, assignments to variable formal parameters affect the corresponding actual parameters; assignments to value parameters do not (see discussion in section 2.1). The notation, following Hoare [1971a], is:

	variable	value
formal parameters	x	v
actual parameters	a	e

where each of x, v, a, and e represents a list of parameters. The two restrictions are that the list "a" must contain distinct identifiers and that no "a" parameter may appear in any of the expressions of the "e" list. The last restriction could be removed with a slight increase in the complexity of the rules of inference.

Simple examples suffice to show what can happen if these restrictions are violated:

```
a. procedure B(var X1,X2 : integer);
   begin X1 := 2; X2 := 3 end
```

One can verify

true{body}(X1=2)∧(X2=3).

The call B(A,A), which violates the distinct "a" list, will yield

true{call B(A,A)}(A=2)∧(A=3)

an impossibility.

b. procedure C(var X : integer; V : integer);
   begin X := V + 1 end

One can verify

true{body}X=V+1.

The call C(A,A), which has an "a" parameter also appearing as an "e" parameter, will yield

true{call C(A,A)}A=A+1

another impossibility.

For each procedure call the corresponding procedure declaration is assumed to be verified as stated in rule V8. The hypothesis of the procedure call rule is thus achieved so the procedure call rule is applicable to both recursive and non-recursive declarations alike. Recall that the recursion rule (D5), i.e. the procedure declaration rule (V8), allows the desired conclusion to be used as an assumption in verifying (the body of) a recursive procedure declaration.

VCG does not allow a component of an array as an "a" parameter. This restriction is implied by H [Hoare 1971a, p. 115, last paragraph]. VCG does not permit the names of procedures or functions to be (actual) parameters; this could be allowed if one were willing to verify separately the procedure definition for each call involving procedure parameters, or if sufficiently general assertions could be supplied.

The procedure call rule (V7) in V is based on the adaptation rule (D7) in H. Both of these rules provide for extreme generality at an increase in complexity. An alternative rule is used in Hoare and Wirth [1972] which treats a procedure call as generalized and concurrent assignment. That is, for each variable parameter x a function is assumed which, given the entry values of the parameters, computes the exit value of x. These functions accomplish the generalized assignment.

Functions

Four of the rules of V have been expanded to allow function calls to occur in Pascal expressions. Function calls may occur only in assignment, conditional, iteration, or procedure call statements. Since Pascal functions have no global variables and no VAR parameters, none of the restrictions needed for procedures apply in the case of functions. Recursively defined functions are allowed.

To give the expanded rules, let P be the conjunction of the preconditions of all the function calls occurring in a statement. Similarly let R be the conjunction of the results (postconditions). The expanded rules are

assignment $P \wedge (R \rightarrow S(e))\{x := e\}S(x)$

where P and R include any function call if x is an array element

conditional

$$\frac{Q \rightarrow P,\ Q \wedge R \wedge U\{A\}S,\ Q \wedge R \wedge \neg U\{B\}S}{Q\{\text{if } U \text{ then } A \text{ else } B\}S}$$

where P and R only include function calls in U

iteration

$$\frac{Q \rightarrow P,\ Q \wedge R \wedge U\{B\}Q \wedge P,\ Q \wedge R \wedge \neg U \rightarrow S}{Q\{\text{while } U \text{ do } B\}S}$$

where P and R only include function calls in U

procedure call

$$\frac{P.G(x,v)\{G(x,v)\}R.G(x,v)}{P \wedge (R \rightarrow P.G \wedge \forall a(R.G \rightarrow S))\{G(a,e)\}S}$$

where P and R refer to the function calls in "e"; P.G and R.G refer to the procedure G.

function declaration

$$\frac{U\{q(v)\}W \ ||\!-\ U\{A\}W}{U\{\text{function } q(v);A\}W}$$

Each of the first four rules assumes that for each function call, the corresponding function declaration is verified as stated in the function declaration rule. If there are no function calls in a statement, then P and R may be taken as "TRUE". In such cases the expanded rules reduce to the original rules. VCG actually omits such vacuous P and R terms. (The definition of P and R as conjunctions means some loss of generality if nested function calls occur such as in Y := G(H(X)). A more complicated definition of P and R is known for such cases but it is not implemented.)

Questions such as array bounds and division by zero can be handled by treating each such operation as an appropriate precondition of a function.

4.2 A RECURSIVE DEFINITION OF VCG

The operation of the verification condition generator is described by the following equations. Let $H(P,B,R)$ denote the LIST of verification conditions for the formula $P\{B\}R$ where B is an asserted Pascal program and where P and R are assertions. $H(P,B,R)$ is given by cases on the form of B. "A" denotes all but the last statement of B, "@" denotes the append operation on lists, "car" and "cdr" denote the list operations of first and rest, and ";" is the Pascal composition connective.

assignment(V1)	$H(P,\ A;x \leftarrow e,\ R(x)) = H(P,A,R(e))$
array assignment(V2)	$H(P,\ A;c[j] \leftarrow e,\ R(c[i])) =$ $H(P,A,R(\text{if } i=j \text{ then } e \text{ else } c[i]))$
null(V3(i))	$H(P,\ \text{null},\ R) = P \rightarrow R$
assert(V3(ii))	$H(P,\ A;\text{assert } Q,\ R) = H(P,A,Q)\ @\ Q \rightarrow R$
iteration(V4)	$H(P,\ A;\text{assert } Q;\text{while } S \text{ do } C,\ R) =$ $H(P,A,Q)\ @\ H(Q \wedge S,C,Q)\ @\ \neg Q \wedge S \rightarrow R$
conditional (V5 and V3(iii))	$H(P,\ A;\text{if } S \text{ then } C \text{ else } D,\ R) =$ $H(P,A,\text{car}(H(S,C,R)))@\text{cdr}(H(S,C,R))@$ $H(P,A,\text{car}(H(\neg S,D,R)))@\text{cdr}(H(\neg S,D,R))$ where a missing "else" means D is null
go to(V6)	$H(P,\ A;\text{go to } L,\ R) = H(P,A,\text{assertion at } L)$
procedure call(V7)	$H(P,\ A;q(a,e),\ R) = H(P,A,U(a,e) \wedge \forall a(W(a,e) \rightarrow R(a,e)))$ where $U(x,v)\{q(x,v)\}W(x,v)$ is an assumption for the procedure q
procedure declaration(V8)	$H(P,\ \text{procedure } q(x,v);C,\ R) = H(P,C,R)$ where $P\{q(x,v)\}R$ is assumed in evaluating $H(P,C,R)$
compound	$H(P,\ A;\text{begin } C \text{ end},\ R) = H(P,\ A;C,\ R)$

The equations for defining $H(P,A,R)$ may be explained by the following: An asserted Pascal program is recursively processed top-down from the outermost syntactic structure to its innermost constituents. The constituents of a compound statement are processed starting with the last constituent. Accordingly, there is a unique rule of inference that is applicable to each constituent. Each rule of inference is applied in the reverse sense from its use in a formal derivation. Thus, from the desired conclusion the appropriate premises are generated as subgoals to be processed recursively. The two assignment rules and go to rule are each applied directly by computing the assertion on the right of the premise from the assertion on the right of the conclusion. The procedure call rule works somewhat analogously: the assertion on the right of the premise is computed from the assertion on the right of the conclusion and from the two assertions of the hypothesis. In all cases this means VCG uses what is called "backward substitution" by King [1969], that is VCG works backwards (opposite to the execution direction) through the program.

That this is possible is far from accidental: Hoare and Wirth [1972, p. 19] state. "The rules of inference are formulated in such a way that the . . . process of deriving necessary properties of the constituents from postulated properties of the composite statement is facilitated. The reason for this orientation is that in deducing proofs of properties of programs it is most convenient to proceed in a 'top-down' direction."

While the notion of "a path between assertions" is not an explicit part of VCG, the recursive processing of subgoals implicitly computes all the required paths between assertions. Each resulting verification condition covers one such path.

A Pascal source program consists of zero or more procedure definitions, zero or more function definitions, and a single main program. VCG produces a separate set of verification conditions for each procedure definition, each function definition, and the main program. If P represents the initial assumption (entry assertion) for a unit and if R represents the desired result (exit assertion) from that unit, then the verification conditions are computed from

P {procedure body} R
P {function body} R
P {main program} R

The assertion R must be present; if P is missing, the assertion "UNRESTRICTED" is assumed which is a synonym for "TRUE". Since Pascal returns a function value by assigning the value to the function identifier (as in Algol), the exit assertion must be modified by deleting the arguments from the defined function name. This is necessary in order that the assignment rules work properly.

To illustrate the equations for defining H(P,A,R) two examples are given. The first shows the subgoaling process on the Quotient-Remainder algorithm of Examples 1 and 2 where the while statement has been replaced by an equivalent go to construction.

```
Goal.        true{r←x;q←0; 10:assert x=r+y*q;
                  if y≤r then begin r←r-y;
                  q←1+q; go to 10 end}¬(y≤r)∧(x=r+y*q)
```

Only V5 is applicable to the goal; first the arguments of the two cars are computed.

```
Subgoal 1.       y≤r{r←r-y;q←1+q; go to 10}
                           ¬(y≤r)∧(x=r+y*q)          V5(Goal)
Subgoal 2.       ¬(y≤r){null}¬(y≤r)∧(x=r+y*q)        V5(Goa ,missing else)
Argument 1.      ¬(y≤r)→¬(y≤r)∧(x=r+y*q)             V3i(Subgoal 2)
Subgoal 3.       y≤r{r←r-y;q←1+q;}x=r+y*q            V6(Subgoal 1),
                                                     assertion at 10
                                                     is x=r+y*q
Subgoal 4.       y≤r{r←r-y}x=r+y*(1+q)               V1(Subgoal 3)
Subgoal 5.       y≤r{null}x=(r-y)+y*(1+q)            V1(Subgoal 4)
Argument 2.      y≤r→x=(r-y)+y*(1+q)                 V3i(Subgoal 5)
Hence the application of V5 to the Goal requires, since the cdr terms
are null,
Subgoal 6.       true{r←x;q←0;assert x=r+y*q}
                           ¬(y≤r)→¬(y≤r)∧(x=r+y*q)   V5(Goal),argument 1
Subgoal 7.       true{r←x;q←0;assert x=r+y*q}
                           y≤r→x=(r-y)+y*(1+q)       V5(Goal),argument 2
Lemma 3.         x=r+y*q→¬(y≤r)→¬(y≤r)∧(x=r+y*q)     V3ii(Subgoal 6)
Lemma 2.         x=r+y*q→y≤r→x=(r-y)+y*(1+q)         V3ii(Subgoal 7)
Subgoal 8.       true{r←x;q←0}x=r+y*q                V3ii(Subgoals 6,7)
Subgoal 9.       true{r←x}x=r+y*0                    V1(Subgoal 8)
Subgoal 10.      true{null}x=x+y*0                   V1(Subgoal 9)
Lemma 1.         true→x=x+y*0                        V3i(Subgoal 10)
```

EXAMPLE 4: SUBGOALING ON QUOTIENT-REMAINDER WITH A GO TO CONSTRUCTION

After logical simplification the three lemmas in Example 4 are identical to the lemmas in Examples 1 and 2. The second example, taken from Hoare [1971a], shows the subgoaling process on a recursive procedure for computing the factorial function.

```
Goal.            a≥0{procedure fact(var r:integer, a:integer)}r=a!
Subgoal 1.       a≥0{fact(r,a)}r=a! ||-
                           a≥0{if a=0 then r←1 else
                           begin fact(r,a-1);
                           r←a*r end}r=a!             V8(Goal)
Only V5 is applicable to Subgoal 1; first the arguments of the two
cars are computed.
Subgoal 2.       a=0{r←1}r=a!                         V5(Subgoal 1)
Subgoal 3.       ¬(a=0){fact(r,a-1);r←a*r}r=a!        V5(Subgoal 1)
Subgoal 4.       a=0{null}1=a!                        V1(Subgoal 2)
Argument 1.      a=0→1=a!                             V3i(Subgoal 4)
Subgoal 5.       ¬(a=0){fact(r,a-1)}a*r=a!            V1(Subgoal 3)
Subgoal 6.       ¬(a=0){null}(a-1≥0)∧∀r#(r#=(a-1)!→a*r#=a!)
                                                      V7(Subgoal 5,
                                                      assumption of
                                                      Subgoal 1)
Argument 2.      ¬(a=0)→(a-1≥0)∧∀r#(r#=(a-1)!→a*r#=a!)
                                                      V3i(Subgoal 6)
Hence the application of V5 to Subgoal 1 requires, since the cdr
terms are null,
Subgoal 7.       a≥0{null}¬(a=0)→(a-1≥0)∧∀r#(r#=(a-1)!→a*r#=a!)
                                              V5(Subgoal 1), argument 2
```

Lemma 2.	$a\geq 0 \rightarrow \neg(a=0) \rightarrow (a-1\geq 0) \wedge \forall r\#(r\#=(a-1)! \rightarrow a*r\#=a!)$	V3i(Subgoal 7)
Subgoal 8.	$a\geq 0 \{null\} a=0 \rightarrow 1=a!$	V5(Subgoal 1), argument 1
Lemma 1.	$a\geq 0 \rightarrow a=0 \rightarrow 1=a!$	V3i(Subgoal 8)

EXAMPLE 5: SUBGOALING ON THE FACTORIAL PROCEDURE

4.3 SPECIFIC IMPLEMENTATION OF VCG

The verification condition generator is written in MLISP2 [Smith and Enea 1973], a version of Lisp which has an Algol-like syntax and an extendable parser. Writing BNF-like syntax equations and associated semantics for each equation permits the rapid, easy construction of a parser for Pascal source programs. The parser handles all details of scanning such as creating identifiers and numbers from individual characters, recognizing delimiters, and processing blanks. The parser produces a list-structured representation of the Pascal source in which all statements and expressions are converted from infix to prefix notation.

The generator is a loop each cycle of which processes one of the subgoals of the form P{A}R. This loop repeatedly determines for each subgoal the single next applicable rule of inference and applies it to the subgoal. As new subgoals are created they are stacked. The result is a list of verification conditions for the input Pascal source program.

Tables 3 and 4 give more detailed information on the subset of Pascal which VCG processes.

statements	implementation status and comments
assignment	left hand side must be either an identifier or a 1-dimensional array element
procedure call	there must be at least one actual parameter (a zero parameter call is no use without global variables); restrictions on actual parameters apply
compound	no restrictions
if-then-else and if-then	no restrictions
case	not implemented - no problems forseen
while	no restrictions
repeat	not implemented - no problems forseen
for	not implemented - revised Pascal has a changed definition of the for statement and a new rule of inference
with	not implemented
go to	a label may appear at most once in the entire source program; go to's

	may only be "local" jumps within a block.
null	deleted by parser

TABLE 3: PASCAL STATEMENTS IN VCG

other syntactic units	implementation status and comments
procedure and function definitions	no global variables permitted
variable and 1-dimensional array declarations	syntax implemented; not further included in verification conditions - no problems forseen
formal parameter declarations	crucial to operation of procedure call rule
const declarations	not implemented - no problems forseen
type declarations	not implemented - problem status not clear
expression	no restrictions; augmented to allow assertions to include quantifiers (∀,∃), implication (→,⊃), and a second type of conjunction (&) (∨ and ∧ are already in Pascal); & is used to conjoin assertions user fewer parentheses than ∧ requires
pointer, set, scalar, record, file	not implemented - some problems expected
constant	integer only; no real numbers or strings

TABLE 4: OTHER SYNTACTIC UNITS IN VCG

The substitution done in the assignment rules (V2 and V3) need not check for a variable becoming bound by the substitution because of three circumstances. First, by convention all quantified variables in the supplied assertions are assumed to be distinct from the program variables. Second, the bound variables introduced by the procedure call rule (V7) are distinct from the program variables because such introduced bound variables all contain the character "#" while no program variable (or supplied assertion variable) may include a "#". Third, these are the only occurrences of quantifiers.

The existential quantifier in the adaptation rule (D7) can be eliminated similarly by notation conventions.

VCG makes very few checks on its input. The major assumption is that the source program obeys all the restrictions of the Pascal language. While these restrictions could relatively easily be checked, they are not since it is reasonable to assume that all input has been processed by a Pascal compiler. There are additional restrictions on the source program imposed by V. Since these might also be enforced by an augmented compiler, little effort was expended in this direction in VCG. Another simplifying and unchecked assumption is that a source program does not contain duplicated variable names; the introduction of fresh variables for duplicated names, using the declaration rule (D8), will remove this restriction.

4.4 TERMINATION OF THE TOP LEVEL OF VCG

The essential reasons why VCG terminates are as follows: All rules except the conditional rule generate one or two subgoals as they process a goal each with fewer statements. The conditional rule generates two subgoals each including a set of statements from just before the if statement back until an assert statement is reached such that this assertion is at the same "indentation level" as the if statement. But even this process is "decreasing" since any further replication of subgoals will be bounded by the same assert statement.

The existence of the assertion needed for the conditional rule follows since each subgoal is well-formed, i.e., there is an assertion at least at the start of each subgoal. Recall the inclusion of "UNRESTRICTED" if needed. No claim is made that the recursive manipulation of the expressions in the assertions will always terminate, but this is separate from the termination of the top level of VCG.

5. EXAMPLES

5.1 FACTORIAL AS A FUNCTION

This example shows the factorial function written as a Pascal recursive function. The next example illustrates the factorial function written as a Pascal recursive procedure. Upper case 'FACT' denotes the program and lower case 'factorial' denotes the mathematical object usually denoted by !. Except for a 'change of notation' the verification conditions are the same in both examples.

```
PASCAL EXIT ARBITRARY;
FUNCTION FACT(N:INTEGER):INTEGER;
        ENTRY N≥0; EXIT FACT(N) = Factorial(N);
        BEGIN IF N = 0 THEN FACT ← 1 ELSE FACT ← N * FACT(N-1) END;
BEGIN X ← X END.;


*****
PASCAL PROGRAM SUCCESSFULLY PARSED


FOR FACT THE
2 VERIFICATION CONDITIONS ARE:


# 1 N≥0→N=0
     →
    1=Factorial(N)

# 2 N≥0→¬(N=0)
     →
    (N-1≥0)∧(FACT(N-1)=Factorial(N-1)→N*FACT(N-1)=Factorial(N))

FOR THE MAIN PROGRAM THE
1 VERIFICATION CONDITIONS ARE:


# 1 UNRESTRICTED
     →
    ARBITRARY
*****
```

5.2 FACTORIAL AS A PROCEDURE

See comments for previous example.

```
PASCAL ENTRY B≥0; EXIT C = Factorial(B);
PROCEDURE FACT(VAR R:INTEGER; A:INTEGER);
ENTRY A ≥ 0; EXIT R = Factorial(A);
BEGIN IF A = 0 THEN R ← 1 ELSE
        BEGIN FACT(R,A-1); R ← A*R END
END;

BEGIN FACT(C,B) END.;


*****
PASCAL PROGRAM SUCCESSFULLY PARSED


FOR FACT THE
2 VERIFICATION CONDITIONS ARE:


# 1 A≥0→A=0
     →
     1=Factorial(A)

# 2 A≥0→¬(A=0)
     →
     (A-1≥0)∧∀R#(R#=Factorial(A-1)→A*R#=Factorial(A))

FOR THE MAIN PROGRAM THE
1 VERIFICATION CONDITIONS ARE:


# 1 B≥0
     →
     (B≥0)∧∀C#(C#=Factorial(B)→C#=Factorial(B))
*****
```

5.3 INTERCHANGE SORT

This example, taken from King [1969], sorts by successively finding the smallest element of the array A. The assertions include provision for showing that the array A at the exit is a permutation of the array A at the entry. The entry array is denoted by the array name A0. The assertions contain two definitions. SAMESET(A,A0,A[arbitrary]) denotes that A and A0 are the same set of elements including repetition. The term A[arbitrary] is a trick to allow VCG to check that an array is unaltered over a path between assertions. The trick is needed because array substitution is done by array element, not by array name. The second definition is for MULTISET(A,A0,J,K,L,M) where K and M denote array elements of A, and J and L denote subscripts of A. MULTISET denotes that A and A0 are the same set of elements including repetition even if J:=K and L:=M are simultaneously done. Thus, e.g.,

    MULTISET(A,A0,J,A[J],LOC,A[LOC])

and

    MULTISET(A,A0,J,A[LOC],LOC,A[J])

both are true, but

    MULTISET(A,A0,J,A[J],J+1,A[J])

is not true generally.

This asserted program and resulting verification conditions were the initial input to the Allen-Luckham theorem prover when it was able to discover the verification condition which could not be proved.

```
PASCAL ENTRY N ≥ 1&SAMESET(A,A0,A[ARBITRARY]);
EXIT ∀K((1≤K)∧(K≤N-1) ⊃ A[K]≤A[K+1])&SAMESET(A,A0,A[ARBITRARY]);
BEGIN J←N;
ASSERT ∀K((J+1≤K)∧(K≤N-1) ⊃ A[K]≤A[K+1]) &
       ∀M((1≤M)∧(M≤J)∧(J≤N-1) ⊃ A[M]≤A[J+1]) &
       1≤J&J≤N & MULTISET(A,A0,J+1,A[J+1],LOC,A[LOC]);
WHILE J ≥2 DO
BEGIN
        BIG ← A[1]; LOC ← 1; I ← 2;
        ASSERT ∀K((J+1≤K)∧(K≤N-1) ⊃ A[K]≤A[K+1]) &
               ∀L((1≤L)∧(L≤I-1)∧(I-1≤N) ⊃ A[L]≤BIG) &
               ∀M((1≤M)∧(M≤J)∧(J≤N-1) ⊃ A[M]≤A[J+1]) &
               BIG=A[LOC]&1≤LOC&LOC≤J&I≥2 &
               2≤J&J≤N & SAMESET(A,A0,A[ARBITRARY]);
        WHILE I≤J DO
                BEGIN IF A[I]>BIG THEN
                        BEGIN BIG←A[I]; LOC←I END;
                I←I+1
                END;
        A[LOC] ← A[J];
        A[J] ← BIG;
        J←J-1
END
END.;
```

```
*****
PASCAL PROGRAM SUCCESSFULLY PARSED


FOR THE MAIN PROGRAM THE
6 VERIFICATION CONDITIONS ARE:


# 1 N≥1&SAMESET(A,A0,A[ARBITRARY])
    →
    ∀K((N+1≤K)∧(K≤N-1)⊃A[K]≤A[K+1])&
    ∀M((1≤M)∧(M≤N)∧(N≤N-1)⊃A[M]≤A[N+1])&1≤N&N≤N&
    MULTISET(A,A0,N+1,A[N+1],LOC,A[LOC])

    Note: A[N+1] is undefined and, since LOC is undefined, eo is A[LOC].
    Nevertheless, by convention this MULTISET term may be coneidered true.

# 2 (∀K((J+1≤K)∧(K≤N-1)⊃A[K]≤A[K+1])&∀M((1≤M)∧(M≤J)∧(J≤N-..)⊃A[M]≤A[J+1])&
    1≤J&J≤N&MULTISET(A,A0,J+1,A[J+1],LOC,A[LOC]))∧(J≥2)
    →
    ∀K((J+1≤K)∧(K≤N-1)⊃A[K]≤A[K+1])&
    ∀L((1≤L)∧(L≤2-1)∧(2-1≤N)⊃A[L]≤A[1])&∀M((1≤M)∧(M≤J)∧(J≤N-1)⊃A[M]≤A[J+1])&
    A[1]=A[1]&1≤1&1≤J&2≥2&2≤J&J≤N&SAMESET(A,A0,A[ARBITRARY])

# 3 (∀K((J+1≤K)∧(K≤N-1)⊃A[K]≤A[K+1])&∀L((1≤L)∧(L≤I-1)∧(I-1≤N)⊃A[L]≤BIG)&
    ∀M((1≤M)∧(M≤J)∧(J≤N-1)⊃A[M]≤A[J+1])&BIG=A[LOC]&1≤LOC&LOC≤J&I≥2&2≤J&J≤N&
    SAMESET(A,A0,A[ARBITRARY]))∧(I≤J)→A[I]>BIG
    →
    ∀K((J+1≤K)∧(K≤N-1)⊃A[K]≤A[K+1])&∀L((1≤L)∧(L≤I+1-1)∧(I+1-1≤N)⊃A[L]≤A[I])&
    ∀M((1≤M)∧(M≤J)∧(J≤N-1)⊃A[M]≤A[J+1])&A[I]=A[I]&1≤I&I≤J&I+1≥2&2≤J&J≤N&
    SAMESET(A,A0,A[ARBITRARY])

# 4 (∀K((J+1≤K)∧(K≤N-1)⊃A[K]≤A[K+1])&∀L((1≤L)∧(L≤I-1)∧(I-1≤N)⊃A[L]≤BIG)&
    ∀M((1≤M)∧(M≤J)∧(J≤N-1)⊃A[M]≤A[J+1])&BIG=A[LOC]&1≤LOC&LOC≤J&I≥2&2≤J&J≤N&
    SAMESET(A,A0,A[ARBITRARY]))∧(I≤J)→¬(A[I]>BIG)
    →
    ∀K((J+1≤K)∧(K≤N-1)⊃A[K]≤A[K+1])&∀L((1≤L)∧(L≤I+1-1)∧(I+1-1≤N)⊃A[L]≤BIG)&
    ∀M((1≤M)∧(M≤J)∧(J≤N-1)⊃A[M]≤A[J+1])&BIG=A[LOC]&1≤LOC&LOC≤J&I+1≥2&2≤J&J≤N&
    SAMESET(A,A0,A[ARBITRARY])

# 5 (∀K((J+1≤K)∧(K≤N-1)⊃A[K]≤A[K+1])&∀L((1≤L)∧(L≤I-1)∧(I-1≤N)⊃A[L]≤BIG)&
    ∀M((1≤M)∧(M≤J)∧(J≤N-1)⊃A[M]≤A[J+1])&BIG=A[LOC]&1≤LOC&LOC≤J&I≥2&2≤J&J≤N&
    SAMESET(A,A0,A[ARBITRARY]))∧¬(I≤J)
    →
    ∀K((J-1+1≤K)∧(K≤N-1)⊃ IF J=K THEN BIG ELSE  IF LOC=K THEN A[J] ELSE A[K]   ≤
    IF J=K+1 THEN BIG ELSE  IF LOC=K+1 THEN A[J] ELSE A[K+1]   )&
    ∀M((1≤M)∧(M≤J-1)∧(J-1≤N-1)⊃ IF J=M THEN BIG ELSE  IF LOC=M THEN A[J] ELSE A[M]≤
    IF J=J-1+1 THEN BIG ELSE  IF LOC=J-1+1 THEN A[J] ELSE
    A[J-1+1]   )&1≤J-1&J-1≤N&
    MULTISET(A,A0,J-1+1, IF J=J-1+1 THEN BIG ELSE  IF LOC=J-1+1 THEN A[J] ELSE A[J-1+1],
    LOC, IF J=LOC THEN BIG ELSE  IF LOC=LOC THEN A[J] ELSE A[LOC]   )
```

```
# 6  (∀K((J+1≤K)∧(K≤N-1)⊃A[K]≤A[K+1])&∀M((1≤M)∧(M≤J)∧(J≤N-1)⊃A[M]≤A[J+1])&
     1≤J&J≤N&MULTISET(A,A0,J+1,A[J+1],LOC,A[LOC]))∧¬(J≥2)
     →
     ∀K((1≤K)∧(K≤N-1)⊃A[K]≤A[K+1])&SAMESET(A,A0,A[ARBITRARY])
*****
```

5.4 A SAMPLE PROOF FOR ONE OF THE VERIFICATION CONDITIONS OF THE PROGRAM INTERCHANGE SORT

Below we give a proof of part of the last verification condition (#6 from Section 5.3). This proof was obtained by a theorem proving program [Allen and Luckham] from the set of axioms and statements shown below. This simple set of axioms was found to be sufficient to obtain proofs of all parts of verification conditions for interchange sort not involving the theory of permutations.

Below P(X) means X-1 and S(X) means X+1.

```
VAR: X,Y,Z,K,M,L;
INF_PRED:≤,=,<;
PRE_OP:P,S,A,J,N,1,2;
EQUALITY:=;
AXIOMS: X≤X;
        (X≤Y∧Y≤Z)→X≤Z;
        (X≤Y∧Y≤X)→Y=X;
        X<Y↔(X≤Y∧¬(X=Y));
        X≤Y∨Y≤X;
        X<S(X);
        P(X)<X;
        S(P(X))=X;
        P(S(X))=X;
        S(1)=2;
        P(2)=1;
        ((X<Y∧P(Y)≤X)→P(Y)=X);
        (X<Y→X≤P(Y));
        (X<Y→S(X)≤Y);
        (X≤Y→P(X)<Y);
LEMMA:  J=1;
PREMISSES: ((S(J)≤K)∧(K≤P(N)))→A(K)≤A(S(K));
 ((1≤M)∧(M≤J)∧(J≤P(N)))→A(M)≤A(S(J));
1≤J;
J≤N∧¬( 2≤J);
THEOREM:(∀K)(1≤K∧K≤P(N))→A(K)≤A(S(K));
;
```

Note that we have added as hypothesis the fact that J=1. The proof of this statement required some computation and was derived by the theorem prover while trying to prove the theorem. The proof that J=1 follows below:

```
  1 = J; 1 2
1 P(2)= 1;AXIOM
2 P(2)= J;3 4
3 1 ≤ J;AXIOM
4 1 ≤ J⊃P(2)= J;5 6
5 P(2)= 1;AXIOM
6 P(2)≤ J⊃P(2)= J;7 8
```

```
7 Y < X∧P(X)≤ Y⊃P(X)= Y;AXIOM
8 J < 2;9 10
9 X ≤ X;AXIOM
10 J ≤ X∧2 ≤ X⊃J < X;11 12
11 X ≤ Y⊃X < Y∨X = Y;AXIOM
12 ¬2 ≤ J;AXIOM  J = 1; 1 2
```

The proof of the last verification condition follows (the constant THEOREM2 arises from the negation of the theorem):

```
NIL 1 12
1 A(THEOREM2)≤ A(S(THEOREM2));3 4
3 J ≤ J⊃A(J)≤ A(S(J));5 6
4 X ≤ X;AXIOM
5 1 ≤ J;AXIOM
6 1 ≤ X∧X ≤ J⊃A(X)≤ A(S(J));7 8
7 J ≤ P(N);9 18
8 1 ≤ X∧(X ≤ J∧J ≤ P(N))⊃A(X)≤ A(S(J));AXIOM
9 J = THEOREM2;11 12
11 J = THEOREM2∨A(THEOREM2)≤ A(S(THEOREM2));13 14
12 ¬A(THEOREM2)≤ A(S(THEOREM2));THEOREM
13 J = THEOREM2∨S(J)≤ THEOREM2;15 16
14 S(J)≤ THEOREM2⊃A(THEOREM2)≤ A(S(THEOREM2));17 18
15 1 < THEOREM2∨1 = THEOREM2;19 20
16 X < Y⊃S(X)≤ Y;AXIOM
17 S(J)≤ X∧X ≤ P(N)⊃A(X)≤ A(S(X));AXIOM
18 THEOREM2 ≤ P(N);THEOREM
19 X ≤ Y⊃X < Y∨X = Y;AXIOM
20 1 ≤ THEOREM2;THEOREM
```

5.5 BINARY TABLE SEARCH

This example, from Clint and Hoare [1972], is a table lookup routine which tries to find the location of the input X in the array A. A is a sorted array of distinct elements, a fact denoted in the assertions by SORTED(A). If X is not in the array an ERROR exit is to be taken. (Our conversion of their program renders this as setting the flag ERROR to TRUE.) Note the use of a go-to for leaving the while loop and the other go-to's. NOTFOUND(X,M,N) expresses that X is not in the array segment from A[M] to A[N]. This program for binary table search is essentially the same as the example in Floyd [1972]. The last verification condition is of the form A → A because VCG does not allow a transfer to the EXIT assertion.

```
PASCAL ENTRY (1<N)∧SORTED(A)∧(A[1]≤X)∧(X<A[N]);
EXIT (A[LOOKUP] = X)∧(ERROR=FALSE) ∨ NOTFOUND(X,M,N) ∧(ERROR = TRUE);
BEGIN M←1; N←N; ERROR←FALSE;
ASSERT (M<N)∧(A[M]≤X)∧(X<A[N])∧SORTED(A) ∧(ERROR=FALSE);
WHILE M+1<N DO BEGIN
        I←(M+N) DIV 2;
        IF X < A[I] THEN N←I ELSE IF A[I] < X THEN M ← I
                ELSE BEGIN LOOKUP ← I; GO TO 1 END
        END;
IF A[M] ≠ M THEN GO TO 2 ELSE BEGIN LOOKUP ← M; GO TO 1 END;
2: ASSERT NOTFOUND(X,M,N); ERROR ← TRUE;
1: ASSERT (A[LOOKUP] = X)∧(ERROR=FALSE) ∨ NOTFOUND(X,M,N) ∧(ERROR = TRUE)
END.;


*****
PASCAL PROGRAM SUCCESSFULLY PARSED


FOR THE MAIN PROGRAM THE
8 VERIFICATION CONDITIONS ARE:


# 1  (1<N)∧SORTED(A)∧(A[1]≤X)∧(X<A[N])
     →
     (1<N)∧(A[1]≤X)∧(X<A[N])∧SORTED(A)∧(FALSE=FALSE)

# 2  (M<N)∧(A[M]≤X)∧(X<A[N])∧SORTED(A)∧(ERROR=FALSE)∧(M+1<N)→
     X<A[(M+N) DIV 2]
     →
     (M<(M+N) DIV 2)∧(A[M]≤X)∧(X<A[(M+N) DIV 2])∧SORTED(A)∧
     (ERROR=FALSE)

# 3  (M<N)∧(A[M]≤X)∧(X<A[N])∧SORTED(A)∧(ERROR=FALSE)∧(M+1<N)→
     ¬(X<A[(M+N) DIV 2])→A[(M+N) DIV 2]<X
     →
     ((M+N) DIV 2<N)∧(A[(M+N) DIV 2]≤X)∧(X<A[N])∧SORTED(A)∧
     (ERROR=FALSE)
```

```
# 4  (M<N)∧(A[M]≤X)∧(X<A[N])∧SORTED(A)∧(ERROR=FALSE)∧(M+1<N)→
     ¬(X<A[(M+N) DIV 2])→¬(A[(M+N) DIV 2]<X)
     →
     (A[(M+N) DIV 2]=X)∧(ERROR=FALSE)∨NOTFOUND(X,M,N)∧(ERROR=TRUE)

# 5  (M<N)∧(A[M]≤X)∧(X<A[N])∧SORTED(A)∧(ERROR=FALSE)∧¬(M+1<N)→A[M]=M
     →
     NOTFOUND(X,M,N)

# 6  (M<N)∧(A[M]≤X)∧(X<A[N])∧SORTED(A)∧(ERROR=FALSE)∧¬(M+1<N)→¬(A[M]=M)
     →
     (A[M]=X)∧(ERROR=FALSE)∨NOTFOUND(X,M,N)∧(ERROR=TRUE)

# 7  NOTFOUND(X,M,N)
     →
     (A[LOOKUP]=X)∧(TRUE=FALSE)∨NOTFOUND(X,M,N)∧(TRUE=TRUE)

# 8  (A[LOOKUP]=X)∧(ERROR=FALSE)∨NOTFOUND(X,M,N)∧(ERROR=TRUE)
     →
     (A[LOOKUP]=X)∧(ERROR=FALSE)∨NOTFOUND(X,M,N)∧(ERROR=TRUE)
*****
```

5.6 THE McCARTHY-PAINTER COMPILER AS A FUNCTION

This example is the McCarthy-Painter compiler for arithmetic expressions [McCarthy and Painter 1967] written as a Pascal recursive function. The assertions given in this example are the same statements that W. Diffie used when he proof-checked the published proof of the compiler correctness. If a "library function" ALPHA is unknown to VCG, it prints a message "ALPHA NOT FOUND". For preconditions and results of that function, the names "PRE_ALPHA" and "RES_ALPHA" are invented.

```
PASCAL EXIT RESULT;

FUNCTION COMPILE(E:EXPRESSION; T:INTEGER):CODE;
ENTRY ISEXP(E)∧(T>AC) ∧ (ISVAR(V)⊃((LOC(V,MAP) < T) ∧ (C(LOC(V,MAP))=C(V,SRST))));
EXIT   (C(AC,OUTCOME(COMPILE(E,T),OBST))=VALUE(E,SRST))
      ∧
      ((U<T) ⊃ (C(U,OBST)=C(U,OUTCOME(COMPILE(E,T),OBST))));
BEGIN IF ISCONST(E) THEN COMPILE ← MKLI(VAL(E))
         ELSE IF ISVAR(E) THEN COMPILE ← MKLOAD(LOC(E,MAP))
         ELSE IF ISSUM(E) THEN
                 COMPILE ←
                   COMPILE(S1(E),T)*MKSTO(T)*COMPILE(S2(E),T+1)*MKADD(T)
END;

BEGIN RESULT ← COMPILE(EXPRESSION,LENGTH(VARS))END.;



*****
PASCAL PROGRAM SUCCESSFULLY PARSED

ISCONST NOT FOUND

ISVAR NOT FOUND

ISSUM NOT FOUND

S1 NOT FOUND

MKSTO NOT FOUND

S2 NOT FOUND

MKADD NOT FOUND

MKLOAD NOT FOUND

LOC NOT FOUND

MKLI NOT FOUND
```

```
VAL NOT FOUND


FOR COMPILE THE
4 VERIFICATION CONDITIONS ARE:


# 1  ISEXP(E)∧(T>AC)∧(ISVAR(V)⊃(LOC(V,MAP)<T)∧(C(LOC(V,MAP))=C(V,SRST)))→
     PRE_ISCONST(E)∧(RES_ISCONST(E)∧ISCONST(E)
     →
     PRE_MKLI(VAL(E))∧PRE_VAL(E)∧(RES_MKLI(VAL(E))∧RES_VAL(E)→
     (C(AC,OUTCOME(MKLI(VAL(E)),OBST))=VALUE(E,SRST))∧
     (U<T⊃C(U,OBST)=C(U,OUTCOME(MKLI(VAL(E)),OBST)))))

# 2  ISEXP(E)∧(T>AC)∧(ISVAR(V)⊃(LOC(V,MAP)<T)∧(C(LOC(V,MAP))=C(V,SRST)))→
     RES_ISCONST(E)∧¬ISCONST(E)→PRE_ISVAR(E)∧(RES_ISVAR(E)∧ISVAR(E)
     →
     PRE_MKLOAD(LOC(E,MAP))∧PRE_LOC(E,MAP)∧(RES_MKLOAD(LOC(E,MAP))∧
     RES_LOC(E,MAP)→(C(AC,OUTCOME(MKLOAD(LOC(E,MAP)),OBST))=VALUE(E,SRST))∧
     (U<T⊃C(U,OBST)=C(U,OUTCOME(MKLOAD(LOC(E,MAP)),OBST)))))

# 3  ISEXP(E)∧(T>AC)∧(ISVAR(V)⊃(LOC(V,MAP)<T)∧(C(LOC(V,MAP))=C(V,SRST)))→
     RES_ISCONST(E)∧¬ISCONST(E)→RES_ISVAR(E)∧¬ISVAR(E)→PRE_ISSUM(E)∧
     (RES_ISSUM(E)∧ISSUM(E)
     →
     ISEXP(S1(E))∧(T>AC)∧(ISVAR(V)⊃(LOC(V,MAP)<T)∧(C(LOC(V,MAP))=C(V,SRST)))∧
     PRE_S1(E)∧PRE_MKSTO(T)∧ISEXP(S2(E))∧(T+1>AC)∧(ISVAR(V)⊃
     (LOC(V,MAP)<T+1)∧(C(LOC(V,MAP))=C(V,SRST)))∧PRE_S2(E)∧PRE_MKADD(T)∧
     ((C(AC,OUTCOME(COMPILE(S1(E),T),OBST))=VALUE(S1(E),SRST))∧(U<T⊃
     C(U,OBST)=C(U,OUTCOME(COMPILE(S1(E),T),OBST)))∧RES_S1(E)∧RES_MKSTO(T)∧
     (C(AC,OUTCOME(COMPILE(S2(E),T+1),OBST))=VALUE(S2(E),SRST))∧(U<T+1⊃
     C(U,OBST)=C(U,OUTCOME(COMPILE(S2(E),T+1),OBST)))∧RES_S2(E)∧RES_MKADD(T)→
     (C(AC,OUTCOME(COMPILE(S1(E),T)*MKSTO(T)*COMPILE(S2(E),T+1)*MKADD(T),OBST))=
     VALUE(E,SRST))∧(U<T⊃C(U,OBST)=C(U,OUTCOME(COMPILE(S1(E),T)*MKSTO(T)*
     COMPILE(S2(E),T+1)*MKADD(T),OBST)))))

# 4  ISEXP(E)∧(T>AC)∧(ISVAR(V)⊃(LOC(V,MAP)<T)∧(C(LOC(V,MAP))=C(V,SRST)))→
     RES_ISCONST(E)∧¬ISCONST(E)→RES_ISVAR(E)∧¬ISVAR(E)→RES_ISSUM(E)∧
     ¬ISSUM(E)
     →
     (C(AC,OUTCOME(COMPILE,OBST))=VALUE(E,SRST))∧
     (U<T⊃C(U,OBST)=C(U,OUTCOME(COMPILE,OBST)))

LENGTH NOT FOUND


FOR THE MAIN PROGRAM THE
1 VERIFICATION CONDITIONS ARE:
```

```
# 1 UNRESTRICTED
    →
    ISEXP(EXPRESSION)∧(LENGTH(VARS)>AC)∧(ISVAR(V)⊃(LOC(V,MAP)<LENGTH(VARS))∧
    (C(LOC(V,MAP))=C(V,SRST)))∧PRE_LENGTH(VARS)∧
    ((C(AC,OUTCOME(COMPILE(EXPRESSION,LENGTH(VARS)),OBST))=
    VALUE(EXPRESSION,SRST))∧(U<LENGTH(VARS)⊃C(U,OBST)=
    C(U,OUTCOME(COMPILE(EXPRESSION,LENGTH(VARS)),OBST)))∧RES_LENGTH(VARS)→
    COMPILE(EXPRESSION,LENGTH(VARS)))
*****
```

REFERENCES

Allen, J.R.; and Luckham, D. 1970. An interactive theorem-proving program, Machine Intelligence 5, Meltzer, B. and Michie, D. (eds.), Edinburgh University Press, 1970, 321-336.

Clint, M.; and Hoare, C.A.R. 1972. Program proving: Jumps and functions, Acta Informatica, 1, 3, 1972, 214-224.

Floyd, R.W. 1964. Algorithm 245, TREESORT 3, Comm. ACM, 7, 12, December 1964, 701.

Floyd, R.W. 1967. Assigning meanings to programs, Proc. of a Symposium in Applied Mathematics, Vol. 19--Mathematical Aspects of Computer Science, Schwartz, J. T. (ed.), American Mathematical Society, 1967, 19-32.

Floyd, R.W. 1972. Toward interactive design of correct programs, Proc. of the IFIP Congress 71, Vol. 1, 1972, 7-10.

Hoare, C.A.R. 1969. An axiomatic basis for computer programming, Comm. ACM, 12, 10, October 1969, 576-580, 583.

Hoare, C.A.R. 1971a. Procedures and parameters: An axiomatic approach, in Symposium on Semantics of Algorithmic Languages, Engeler, E. (ed.), Springer-Verlag, 1971, 102-116.

Hoare, C.A.R. 1971b. Proof of a program: FIND, Comm. ACM. 14, 1, January 1971, 39-45.

Hoare, C.A.R.; and Wirth, N. 1972. An axiomatic definition of the programming language Pascal, Berichte der Fachgruppe Computer-Wissenschaften 6, E.T.H., Zurich, November 1972.

King, J.C. 1969. A program verifier, Ph.D. thesis, Carnegie-Mellon University, 1969. See also IFIP Congress 71 Booklet TA-2, 142-146.

London, R.L. 1972. The current state of proving programs correct, Proc. of ACM Annual Conference, ACM, 1972, 39-46.

McCarthy, J; and Painter, J.A. 1967. Correctness of a compiler for arithmetic expressions, Proc. of a symposium in Applied Mathematics, Vol. 19--Mathematical Aspects of Computer Science, Schwartz, J. T. (ed.), American Mathematical Society, 1967, 33-41.

Smith, D.C.; and Enea, H. J. 1973. MLISP2, Artificial Intelligence Memo AIM-195, Stanford University, April 1973.

Wirth, N. 1971. The programming language Pascal, Acta Informatica, 1, 1. 1971, 35-63.

Wirth, N. 1972. The programming language Pascal (Revised Report), Berichte der Fachgruppe Computer-Wissenschaften 5, E.T.H., Zurich, November 1972.

ACKNOWLEDGEMENTS

We are indebted [illegible] R. Hoare and Niklaus Wirth for preprints of several of thei[illegible]pers, especially drafts of the Pascal axiomatization. We als[illegible]nk Nori Suzuki for pointing out errors in VCG and Jorge Morales for p[illegible]ing the sample proof in Section 5.4.

Most of this work was completed w[illegible]he authors were members of the Stanford Artificial Intelligence [illegible]t during the academic year 1971-72. We are grateful to John McCar[illegible] making it possible for us to be together in such a stimulating env[illegible]ent. Horace Enea and David S[illegible]ith, developers of the MLISP2 system, p[illegible]ntly answered our many questions about its use.