techniques," in *Proc. Purdue Centennial Year Symp. Inform. Processing*, vol. 1. Lafayette, Ind.: Purdue Univ. Eng. Exp. Sta., Apr. 28–30, 1969, pp. 81–91.
[10] D. T. Brown, "Error detecting and correcting binary codes for arithmetic operations," *IRE Trans. Electron. Comput.*, vol. EC-9, pp. 333–337, Sept. 1960.
[11] W. W. Peterson, *Error Correcting Codes*. New York: Wiley, 1961, pp. 236–244.
[12] D. S. Henderson, "Residue class error checking codes," in *Preprints Papers, 16th Natl. Meet. Ass. Comput. Mach.*, 1961.
[13] H. L. Garner, "Error codes for arithmetic operations," *IEEE Trans. Electron. Comput.*, vol. EC-15, pp. 763–770, Oct. 1966.
[14] W. W. Peterson, "On checking an adder," *IBM J. Res. Develop.*, vol. 2, pp. 166–168, Apr. 1958.
[15] H. L. Garner, "Generalized parity checking," *IRE Trans. Electron. Comput.*, vol. EC-7, pp. 207–213, Sept. 1958.
[16] T. R. N. Rao, "Biresidue error-correcting codes for computer arithmetic," *IEEE Trans. Comput.*, vol. C-19, pp. 398–402, May 1970.
[17] T. R. N. Rao and O. N. Garcia, "Cyclic and multiresidue codes for arithmetic operations," *IEEE Trans. Inform. Theory*, vol. IT-17, pp. 85–91, Jan. 1971.

Proving Programs to be Correct

JAMES C. KING

Abstract—This paper formally describes a technique for proving that computer programs will always execute correctly. In order to do this, an abstract model for a program and its execution is given. Then, correctness of programs and proofs of correctness of programs are defined with respect to that model.

Index Terms—Abstract programs, debugging, program correctness, program execution model, state vector, theory of programming.

Introduction

THIS paper is concerned with proving that computer programs will always execute correctly. The motivation for this has existed since shortly after people began to write programs for computers. At that time, von Neumann and Goldstine presented ideas similar to those explained here [6]. Computers and the programs that make them operate are playing a more and more important role in our everyday lives. Some means for assuring that these programs will always run properly is critically needed.

A Model of Computation

A firm basis for discussing the correctness of programs is provided by establishing an abstract model for computations. The model is intended to characterize grossly the way most current digital computers perform their computations. A "program" is defined to be a finite ordered set of "statements" $s_1, s_2, \cdots, s_m$. A program operates on a fixed set of variables $x_1, x_2, \cdots, x_n$ that assume values from the domains $D_1, D_2, \cdots, D_n$, respectively. There are three types of program statements of the forms:

1) assign: $\{x_k, f(x_1, x_2, \cdots, x_n), j\}$;
2) test: $\{p(x_1, x_2, \cdots, x_n), j_1, j_2\}$;
3) halt.

The program variables can be interpreted to be storage cells of a computer memory and the three basic statement types are a means of summarizing the gross effect on that memory caused by a computer executing sequences of instructions.

The value of any one program variable (x_k) can be changed by the execution of an assign statement. The function f, possibly different for each assign statement, is a total function over the program variables $x_1, x_2, \cdots, x_n$ and results in a value in domain D_k which is used to replace the value of x_k. Each of j, j_1, and j_2 are integer constants between 1 and m (the total number of program statements) and are program statement subscripts. They occur in the program statements to indicate the order of execution of statements as explained later. Each test statement contains a total predicate $p(x_1, x_2, \cdots, x_n)$ (possibly different for each test statement) which results in *true* or *false* when evaluated for fixed values of the program variables.

A "state vector" of a program is an $(n+1)$-tuple of values $\langle N, a_1, a_2, \cdots, a_n \rangle$ where N is the statement subscript of the "next" statement to be executed ($1 \leq N \leq m$) and for $1 \leq i \leq n$ a_i is the value associated with the program variable x_i and as such must come from the domain D_i. An execution of a program $P = \{s_1, s_2, \cdots, s_m\}$ begins with an "initial state vector" $v_1 = \langle 1, a_1, a_2, \cdots, a_n \rangle$ and develops an "execution sequence" of state vectors $v_1, v_2, v_3, \cdots$. Sup-

Manuscript received March 1, 1971; revised June 11, 1971. This work was performed while the author was at the Carnegie-Mellon University, Pittsburgh, Pa., and is part of his Ph.D. dissertation [4]. This work was supported by the Advanced Research Project Agency of the Office of the Secretary of Defense (F44620-67-C0058).

The author is with the IBM Thomas J. Watson Research Center, Yorktown Heights, N. Y.

pose a program has been partially executed creating the partial execution sequence $v_1, v_2, \cdots, v_j$. Suppose further that $v_j = \langle i, b_1, b_2, \cdots, b_n \rangle$. Then the next vector in the sequence (v_{j+1}) is determined by execution of program statement s_i, selected by examining the first component of the vector v_j (i.e., i). Statement s_i can have one of the three following forms.

1) s_i=assign: $\{x_k, f(x_1, x_2, \cdots, x_n), q\}$. In this case, execution of s_i results in the new vector $v_{j+1} = \langle q, b_1, b_2, \cdots, b_k', \cdots, b_n \rangle$ where $b_k' = f(b_1, b_2, \cdots, b_n)$, a constant from domain D_k. The constant q in s_i indicates that statement s_q is to be executed next.

2) s_i=test: $\{p(x_1, x_2, \cdots, x_n), j_1, j_2\}$. In this case, the execution of s_i determines a choice of subsequent statements to be executed (j_1 or j_2) based on the "current" value of the predicate p. The test statements make no change to any of the program variable's values. If $p(b_1, b_2, \cdots, b_n) = true$ then the new vector $v_{j+1} = \langle j_1, b_1, b_2, \cdots, b_n \rangle$. If $p(b_1, b_2, \cdots, b_n) = false$ then $v_{j+1} = \langle j_2, b_1, b_2, \cdots, b_n \rangle$.

3) s_i=halt. In this case, there is no next vector and the sequences of state vectors is finite. The vector v_j is denoted as the "final vector" for this execution of the program.

Thus, an execution of a program begins with an initial state vector and generates a sequence of state vectors by executing individual statements in the program. If the sequence is finite, the last vector is called the final vector and the program is said to "terminate" for this initial vector, otherwise it is "nonterminating."

For example, let x_1, x_2, x_3, and x_4 be variables over the domain of all signed integers. Define the program $P' = \{0$

s_1: assign: $\{x_4, x_2, 2\}$,
s_2: assign: $\{x_3, 0, 3\}$,
s_3: test: $\{x_4 = 0, 4, 5\}$,
s_4: halt,
s_5: assign: $\{x_3, x_1 + x_3, 6\}$,
s_6: assign: $\{x_4, x_4 - 1, 3\}\}$.

The initial state vector $\langle 1, 3, 2, 40, 0 \rangle$ forms the execution sequence $\{\langle 1, 3, 2, 40, 0 \rangle, \langle 2, 3, 2, 40, 2 \rangle, \langle 3, 3, 2, 0, 2 \rangle, \langle 5, 3, 2, 0, 2 \rangle, \langle 6, 3, 2, 3, 2 \rangle, \langle 3, 3, 2, 3, 1 \rangle, \langle 5, 3, 2, 3, 1 \rangle, \langle 6, 3, 2, 6, 1 \rangle, \langle 3, 3, 2, 6, 0 \rangle, \langle 4, 3, 2, 6, 0 \rangle\}$. The sequence is finite and the final vector is $\langle 4, 3, 2, 6, 0 \rangle$ giving $x_1 = 3$, $x_2 = 2$, $x_3 = 6$, and $x_4 = 0$. Notice that in this case the final values satisfy $x_3 = x_1 * x_2$. It will be proved later that all final vectors satisfy this equation. On the other hand, the initial state vector $\langle 1, 3, -1, 40, 0 \rangle$ forms the infinite execution sequence $\{\langle 1, 3, -1, 40, 0 \rangle, \langle 2, 3, -1, 40, -1 \rangle, \langle 3, 3, -1, 0, -1 \rangle, \langle 5, 3, -1, 0, -1 \rangle, \langle 6, 3, -1, 3, -1 \rangle, \langle 3, 3, -1, 3, -2 \rangle, \langle 5, 3, -1, 3, -2 \rangle, \langle 6, 3, -1, 6, -2 \rangle, \langle 3, 3, -1, 6, -3 \rangle, \cdots\}$. The program is nonterminating for $\langle 1, 3, -1, 40, 0 \rangle$.

Correctness

The "correctness" of a program $P = \{s_1, s_2, \cdots, s_m\}$ is defined with respect to two predicates over the variables of P: the "initial predicate," say, $I(x_1, x_2, \cdots, x_n)$, and the "final predicate," say, $F(x_1, x_2, \cdots, x_n)$. Assume v_1 is an initial state vector for P. P is then "correct for v_1" with respect to I and F if either: 1) P is nonterminating for v_1; or 2) P terminates for v_1, generating the execution sequence $\{v_1, v_2, \cdots, v_k\}$, and $I(v_1) \supset F(v_k)$. (Notation: If $v_i = \langle q, a_1, a_2, \cdots, a_n \rangle$ then $I(v_i)$ is used as an abbreviation for $I(a_1, a_2, \cdots, a_n)$.) If the initial values of a program execution satisfy the condition I and the execution terminates, then the final values must satisfy F. Program P is simply "correct" with respect to I and F if it is correct for v_1 with respect to I and F for all possible initial vectors v_1. Note the following three points.

1) The program $\{$assign: $\{x_1, x_1, 1\}\}$ is nonterminating for any initial vector and is, therefore, vacuously *correct* with respect to *any* predicates I and F.

2) Any program P is trivially correct with respect to any initial predicate $I(v)$ and the final predicate *true*.

3) If *a*) F is identically *false*, b) $I(v) = true$ for some initial vector v, and c) P is correct for v with respect to this I and F, then P is nonterminating for v. If this were not the case, one would have proved $I(v) \supset false$, which is impossible since $I(v) = true$. This observation is the basis for results dealing with the termination of programs in [5].

Given a program P, an initial predicate I, and a final predicate F, the problem is to devise a calculus which would enable one to "verify" P; that is, to construct a rigorous proof that P is correct with respect to I and F. Such a technique is described below and is essentially that developed by Floyd in [2] and [3]. In [5] Manna also presents a similar calculus. Grossly, a proof is a deduction over execution sequences. With each vector in any finite execution sequence, say, $V = \{v_1, v_2, \cdots, v_k\}$, one associates a predicate, say, $A_1, A_2, \cdots, A_k$, each over the variables of P such that $A_1(v_1) \supset A_2(v_2)$, $A_2(v_2) \supset A_3(v_3)$, $\cdots$, and $A_{k-1}(v_{k-1}) \supset A_k(v_k)$. If $I = A_1$ and $F = A_k$, one may conclude that $I(v_1) \supset F(v_k)$ and establish that P is correct for v_1.

To begin a more complete description of the method, the concept of a control path of a program is useful. A "control path," or simply a "path," is a sequence of statements of the program, say, $\{r_1, r_2, \cdots\}$, with the condition that for any two adjacent statements, r_i and r_{i+1}, in the list: 1) if r_i is of the form assign: $\{x_k, f(x_1, x_2, \cdots, x_n), j\}$, then r_{i+1} is s_j; or 2) if r_i is of the form test: $\{p(x_1, x_2, \cdots, x_n), j, k\}$, then r_{i+1} may be either s_j or s_k. In addition, no r_i ($i \geq 1$) may be a halt statement except (possibly) the last member of a finite sequence. Each execution of a program defines a control path if one lists the statements excuted in order. A sequence of state vectors "executes a path" if the sequence of statements executed in forming the vectors is that path. A control path does not necessarily begin with statement 1 nor necessarily end, if at all, with a halt statement.

A program is "loop free" or has no "loops" if all possible control paths are finite, or equivalently, if no control path has more than one occurrence of the same statement. A loop-free program may be verified by constructing a proof over each

control path which begins with statement 1 and ends with a halt statement. Such proofs are done by application of the following more general procedure.

Verifying a Path

Given: a program $P=\{s_1, s_2, \cdots, s_N\}$, an arbitrary finite control path $R=\{r_1, r_2, \cdots, r_m\}$ of P, and two predicates over the variables of the program, say, $A(x_1, x_2, \cdots, x_n)$ and $B(x_1, x_2, \cdots, x_n)$. R is "verified" with respect to A and B if for any state vector v_1 that, beginning at r_1, causes the path R to be executed up to but not including r_m, resulting in the vector v_m, one can conclude $A(v_1) \supset B(v_m)$. That is, if A is satisfied by a vector at the beginning of the path and the path is executed, then B must be satisfied by the resulting vector.

This definition applies to the path R independently of its context in the encompassing program. The statement "for *any* state vector v_1" refers to all possible vectors drawn from the domains D_i, $1 \leq i \leq n$ even though no execution sequence for the program may contain such a vector corresponding to the beginning of path R.

R can be verified by constructing a sequence of predicates corresponding to the statements in R. Let $A_1 = A$. Then develop, in order, $A_1, A_2, \cdots, A_m$. Suppose this has been done up to A_i, then A_{i+1} can be constructed as follows.

1) If r_i is of the form assign: $\{x_k, f(x_1, x_2, \cdots, x_n), j\}$, then $A_{i+1}(x_1, x_2, \cdots, x_n) = \exists x_k' [A_i(x_i, \cdots, x_k', \cdots, x_n) \wedge x_k = f(x_1, \cdots, x_k', \cdots, x_n)]$. Here x_k' occurs in the kth position for A_i and f, and $\exists x_k'$ is read "there exists a value in D_k, say, x_k', such that."

If x_k' is thought of as the "old" value of x_k then the new predicate A_{i+1} incorporates what had been known about the old x_k, namely, $A_i(x_1, \cdots, x_k', \cdots, x_n)$, as well as exhibits the functional relationship of the old and new values of x_k by the expression $x_k = f(x_1, \cdots, x_k', \cdots, x_n)$.

2) If r_i is of the form test: $\{p(x_1, x_2, \cdots, x_n), j_1, j_2\}$, then: a) if r_{i+1} is statement s_{j_1}, then $A_{i+1}(x_1, x_2, \cdots, x_n) = [p(x_1, x_2, \cdots, x_n) \wedge A_i(x_1, x_2, \cdots, x_n)]$; otherwise, b) r_{i+1} is s_{j_2} and then $A_{i+1}(x_1, x_2, \cdots, x_n) = [\sim p(x_1, x_2, \cdots, x_n) \wedge A_i(x_1, x_2, \cdots, x_n)]$.

The test statement does not change any program variables' values so that the new predicate A_{i+1} includes the predicate A_i unchanged. Additionally, A_{i+1} includes the new information discovered by the test, that either $p(x_1, x_2, \cdots, x_n)$ or $\sim p(x_1, x_2, \cdots, x_n)$ is *true* depending on which test branch this path follows.

After building the sequence $A_1, A_2, \cdots, A_m$, we form the expression $A_m(x_1, x_2, \cdots, x_n) \supset B(x_1, x_2, \cdots, x_n)$ which is called a "verification condition." This name is derived from the proposition that: the path R is verified with respect to A and B if and only if this verification condition is *true* for all $x_1, x_2, \cdots, x_n$ in $D_1, D_2, \cdots, D_n$, respectively.

The proof of this proposition follows in two parts.

Case 1: Consistency (Verification condition $=$ *true* implies R is verified.) Assume that the verification condition is *true*. Let $V=(v_1, v_2, \cdots, v_n)$ be a sequence of state vectors which execute the path R such that $A(v_1) = true$. (If no such sequence exists then R is trivially verified with respect to A and B.) One can show by induction on i that each A_i is *true* when evaluated at the corresponding state vector v_i $(1 \leq i \leq m)$. Since the verification condition is *true*, $A_m(v_m) \supset B(v_m)$, and $B(v_m)$ must also be *true* or $A(v_1) \supset B(v_m)$ and R is verified.

Let $v_i = <j_1, a_1, a_2, \cdots, a_n>$, $v_{i+1} = <j_2, b_1, b_2, \cdots, b_n>$, and assume that $A_i(v_i) = true$. The induction step is then to show that $A_{i+1}(v_{i+1}) = true$. There are two cases.

1) If r_i is assign: $\{x_k, f(x_1, x_2, \cdots, x_n), j\}$, then $A_{i+1}(v_{i+1}) = \exists x_k' [A_i(b_1, \cdots, x_k', \cdots, b_n) \wedge b_k = f(b_1, \cdots, x_k', \cdots, b_n)]$. But the execution of statement r_i determines that $b_i = a_i$ for all $i \neq k$, $1 \leq i \leq n$, and that $b_k = f(a_1, \cdots, a_k, \cdots, a_n)$. By choosing $x_k' = a_k$, $A_{i+1}(v_{i+1}) = (A_i(v_i) \wedge b_k = f(v_i))$ which is *true*.

2) If r_i is test: $\{p(x_1, x_2, \cdots, x_n), j_1, j_2\}$, then $A_{i+1}(v_{i+1}) = [q(v_{i+1}) \wedge A_i(v_{i+1})]$ where $q(v_{i+1}) = p(v_{i+1})$ if $r_{i+1} = s_{j_1}$, or $q(v_{i+1}) = \sim p(v_{i+1})$ and $r_{i+1} = s_{j_2}$. But $b_i = a_i$ for all i, $1 \leq i \leq n$, so $A(v_{i+1}) = A(v_i)$, and since the vectors were assumed to execute this path $q(v_{i+1}) = q(v_i) = true$ in either case.

This concludes Case 1 of the proof.

Case 2: Completeness (R is verified implies the verification condition is *true*.) This case is proved by contradiction. Suppose that the verification condition is *false*. (Note that the verification condition is an expression over the program variables $x_1, x_2, \cdots, x_n$ and exists regardless of whether or not the path can be executed.) Since the verification condition is *false* there are values $d_1, d_2, \cdots, d_n$ such that $A_m(d_1, d_2, \cdots, d_n) = true$ and $B(d_1, d_2, \cdots, d_n) = false$. Starting with $v_m = <j, d_1, d_2, \cdots, d_n>$ (j is determined by assuming that $r_m = s_j$) and $A_m(v_m) = true$ one can construct, in reverse order, a set of state vectors $v_1, v_2, \cdots, v_m$ which executes R and for which $A_i(v_i) = true$ for $1 \leq i \leq m$. In particular, $A(v_1) = true$, giving a contradiction since R cannot be verified for v_1 with respect to A and B ($B(v_m) = false$).

The proof of this case is concluded by showing the construction of a sequence $v_1, v_2, \cdots, v_m$. Assume $v_{i+1}, \cdots, v_m$ has been constructed to execute $\{r_{i+1}, \cdots, r_m\}$. Let $v_{i+1} = <k_2, b_1, b_2, \cdots, b_n>$ ($s_{k_2} = r_{i+1}$) and assume $A_{i+1}(v_{i+1}) = true$. The induction step is to show that there exists a $v_i = <k_1, a_1, a_2, \cdots, a_n>$ such that $A_i(v_i) = true$ and such that $v_i, v_{i+1}, \cdots, v_m$ executes $\{r_i, r_{i+1}, \cdots, r_m\}$. Again, there are two cases.

1) If r_i is assign: $\{x_k, f(x_1, x_2, \cdots, x_n), k_2\}$, then $A_{i+1}(v_{i+1}) = \exists x_k' [A_i(b_1, \cdots, x_k', \cdots, b_n) \wedge b_k = f(b_1, \cdots, x_k', \cdots, b_n)] = true$. Let a_k be some value for x_k' which makes $A_{i+1}(v_{i+1}) = true$ and choose $a_i = b_i$ for $1 \leq i \leq n$, $i \neq k$. Choose k_1 such that $s_{k_1} = r_i$. Then $A_i(v_i) = true$ and v_{i+1} is derived from v_i by execution of r_i ($b_k = f(v_i)$).

2) If r_i is test: $\{p(x_1, x_2, \cdots, x_n), j_1, j_2\}$, then let $a_i = b_i$ for $1 \leq i \leq n$. Choose k_1 such that $s_{k_1} = r_i$. If $k_2 = j_1$, then $A_{i+1}(v_{i+1}) = [p(v_{i+1}) \wedge A_i(v_{i+1})] = [p(v_i) \wedge A_i(v_i)] = true$ so $p(v_i) = A_i(v_i) = true$ and v_{i+1} is derived from v_i by execution of r_i. If $k_2 = j_2$, then $A_{i+1}(v_{i+1}) = [\sim p(v_{i+1}) = A_i(v_{i+1})] = [\sim p(v_i)$

$\wedge A_i(v_i)] = true$ so $\sim p(v_i) = A_i(v_i) = true$ and v_{i+1} is derived from v_i by execution of r_i.

This concludes Case 2 of the proof. The proof of Case 2 exposes an interesting corollary to the proposition: if there is no sequence of state vectors $v_1, v_2, \cdots, v_m$ that executes R such that $A(v_1) = true$, then A_m must be identically *false*.

A statement on a control path defines a transformation of a state vector v_1 into a new state vector v_2. Also rules for statements have been presented above for transforming a predicate A_1 into a new predicate A_2 such that $A_1(v_1) \supset A_2(v_2)$. That is, if one knows A_1 about v_1 before the statement is executed, one can deduce that A_2 is known about the resulting v_2. The completeness of the above proposition also assures us that A_2 represents as much as one could learn about v_2 from only knowing $A_1(v_1)$ and understanding what the execution of the statement means. In this sense, one may view these transformation rules as gving a "semantic definition" for the execution of statements in the programming language [2].

Note that, once a path is determined, the transformations on the predicates and the creation of verification conditions is done strictly by examining the program statements. The arguments discussing particular values of program variables in state vectors are used to prove the correctness of this method but do not occur explicitly in its application. This is important since the results of a single proof may have implications about an infinite class of execution sequences, in the same way that a finite program represents (possibly) an infinite number of different computions.

Verifying A Program

This method for verifying a finite path of a program may be applied directly to constructing a proof of correctness for any loop-free program. The finite number of finite paths beginning at statement 1 and ending with a halt statement can each be processed, associating I (the program's initial predicate) with the beginnings of the paths and F (the program's final predicate) with the ends. If all the verification conditions are true, the program is correct with respect to I and F. If one or more of the verification conditions is not true, there is an execution sequence for which the program is not correct.

Can this same method be applied to an arbitrary program which is not necessarily loop free? Such a program may have an infinite number of control paths. Directly applying the method used for loop-free programs will not work, but the problem is overcome by introducing "inductive predicates." These predicates are over the variables of the program and each is associated with a distinct statement in the program, "tagging" that statement. The initial predicate I tags statement 1 and the final predicate F tags all halt statements. The statements of the program must be tagged in such a way that there is no control path which has two occurrences of the same untagged statement that are not separated by at least one tagged statement. This is always possible since the property would trivially hold if one tagged every statement in the program. These tagged statements essentially "cut" all loops in the program.

Consider all "tagged paths" in the program which take the form $R = \{r_1, r_2, \cdots, r_k\}$ where statement r_1 is tagged by predicate A, r_k is tagged by predicate B, and $r_2, \cdots, r_{k-1}$ are untagged statements. Using the method developed earlier, one verifies R with respect to A and B, and does this for all tagged paths in the program. There are only a finite number of these since the tagging was done so that each of $r_2, \cdots,$ and r_{k-1} are distinct. Each verification produces a verification condition to be proved. If they are all proved *true* (i.e., the verifications were successful), then the program is correct with respect to the given I and F. This is easily seen by looking at any finite execution sequence $\{v_1, v_2, \cdots, v_m\}$ and its associated control path $R = \{r_1, r_2, \cdots, r_m\}$. If $I(v_1) = false$, the program is correct for v_1. Suppose $I(v_1) = true$. Let r_1 be the next tagged statement (r_m is *halt* and is tagged by F). Name that tag A. Since all control paths starting and ending with tagged statements (i.e., all tagged paths) have been verified, $I(v_1) \supset A(v_i)$ and consequently $A(v_i) = true$. The next tagged path of R has A as its initial tag and we apply the same argument to this path, etc., eventually concluding that $F(v_m) = true$. But then $I(v_1) \supset F(v_m)$, and the program is correct for v_1.

The method may fail when not all verification conditions are *true*. This does not necessarily mean that the program is not correct with respect to I and F. If the inductive predicates are not chosen properly, a correct program may not be verifiable. The strongest statement that can be made in this respect is that, if the program is correct, there exist inductive predicates that will yield a proof by this technique. (For a proof of this, see [1, Theorem 1].)

An Example

An example of the complete process is now given using the program P' previously presented. A flowchart corresponding to P' is given as Fig. 1. Example control paths of P' are

$$cp_1 = \{s_3, s_5\}$$
$$cp_2 = \{s_2, s_3, s_4\}$$
$$cp_3 = \{s_1, s_2, s_3, s_5, s_6, s_3\}.$$

Path cp_3 shows that P' is not loop free since s_3 occurs twice. An infinite number of paths as well as an infinite path can be created from the cycle occurring in cp_3: $\{s_3, s_5, s_6, s_3\}$.

P' is correct with respect to $I(x_1, x_2, x_3, x_4) = true$ and $F(x_1, x_2, x_3, x_4) = (x_3 = x_1 * x_2)$. To show this, associate the inductive predicate $A(x_1, x_2, x_3, x_4) = (x_3 + x_4 * x_1 = x_1 * x_2)$ with statement s_3. Then the tagged paths to be verified are

$$tp_1 = \{s_1, s_2, s_3\} \qquad tp_2 = \{s_3, s_4\} \qquad tp_3 = \{s_3, s_5, s_6, s_3\}.$$

To verify tp_1, begin with I and using statement s_1 form the new predicate $\exists x_4' \, [true \wedge (x_4 = x_2)]$ or more simply $(x_4 = x_2)$. Using this and s_2 get $\exists x_3' \, [(x_4 = x_2) \wedge x_3 = 0]$ or $[(x_4 = x_2) \wedge (x_3 = 0)]$. Next form the verification condition

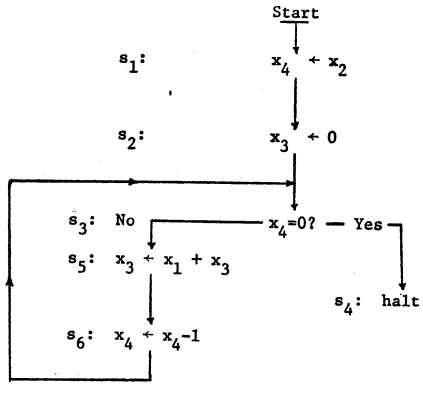

Fig. 1. Flow diagram of program P'.

$$vc_1 = \{[(x_4 = x_2) \wedge (x_3 = 0)] \supset (x_3 + x_4{*}x_1 = x_1{*}x_2)\}.$$

To verify tp_2, begin with A and using statement s_3 form $\{(x_3+x_4{*}x_1=x_1{*}x_2)\wedge(x_4=0)\}$. The second verification condition is then

$$vc_2 = \{[(x_3 + x_4{*}x_1 = x_1{*}x_2) \wedge (x_4 = 0)] \supset (x_3 = x_1{*}x_2)\}.$$

To verify tp_3, begin again with A and using statement s_3 form $\{(x_3+x_4{*}x_1=x_1{*}x_2)\wedge(x_4\neq 0)\}$. Using this and s_5 get $\exists x_3'\{(x_3'+x_4{*}x_1=x_1{*}x_2)\wedge(x_4\neq 0)\wedge(x_3=x_1+x_3')\}$. Then from s_6 get $\exists x_4'\exists x_3'\{(x_3'+x_4'{*}x_1=x_1{*}x_2)\wedge(x_4'\neq 0)\wedge(x_3=x_1+x_3')\wedge(x_4=x_4'-1)\}$. The final verification condition is

$$\begin{aligned} vc_3 = \{&\exists x_4' \exists x_3' \{(x_3' + x_4'{*}x_1 = x_1{*}x_2) \wedge (x_4' \neq 0) \\ &\wedge (x_3 = x_1 + x_3') \wedge (x_4 = x_4' - 1)\} \\ &\supset (x_3 + x_4{*}x_1 = x_1{*}x_2)\}. \end{aligned}$$

All three verification conditions can be proved to be *true*, and so P is correct with respect to I and F.

Note that if an equation of the form $x_k = f(x_1, \cdots, x_k', \cdots, x_n)$ can be solved for x_k', i.e., rewritten as $x_k' = g(x_1, \cdots, x_k, \cdots, x_n)$, then in this case (the equation being a conjunct of the predicate) the expression $g(x_1, \cdots, x_k, \cdots, x_n)$ can be substituted throughout the predicate. This eliminates x_k' and its associated existential quantifier. So when verifying tp_3 one could have simplified

$$\exists x_3'[(x_3' + x_4{*}x_1 = x_1{*}x_2) \wedge (x_4 \neq 0) \wedge (x_3 = x_1 + x_3')],$$

using $x_3' = x_3 - x_1$, to get $(x_3-x_1+x_4{*}x_1=x_1{*}x_2)\wedge(x_4\neq 0)$. In this way, vc_3 can be simplified to

$$\begin{aligned} vc_3 = \{&[(x_3 - x_1 + (x_4 + 1){*}x_1 = x_1{*}x_2) \wedge (x_4 + 1 \neq 0)] \\ &\supset (x_3 + x_4{*}x_1 = x_1{*}x_2)\}. \end{aligned}$$

The inductive predicate A for P' is an *invariant* statement about the variables of P' with respect to executions of the cycle $\{s_3, s_5, s_6, s_3\}$. That is, verification condition vc_3 essentially proves that if A is *true* of the values of the variables immediately before executing s_3 and if the statements s_3, s_5, and s_6 are then executed, A will again be *true* for the new values. This is a property of inductive predicates. The following observation also lends some intuitive feeling to the inductive predicates: the predicate A in the above example shows the relationship among what has been computed (x_3), the desired result ($x_1{*}x_2$), and what has yet to be computed ($x_4{*}x_1$) (i.e., $x_3+x_4{*}x_1=x_1{*}x_2$).

In the example, the inductive predicate could have been associated with either statement s_5 or s_6 as well as s_3, as was done. Of course, these would not necessarily be the same predicates in each case. For example, let $A'(x_1, x_2, x_3, x_4) = (x_3+(x_4-1){*}x_1=x_1{*}x_2)$ be the inductive predicate associated with statement s_6. Then the tagged paths would be

$$\begin{aligned} tp_1' &= \{s_1, s_2, s_3, s_4\} \\ tp_2' &= \{s_1, s_2, s_3, s_5, s_6\} \\ tp_3' &= \{s_6, s_3, s_5, s_6\} \\ tp_4' &= \{s_6, s_3, s_4\}. \end{aligned}$$

This time, four verification conditions would be generated and proved.

One may also "over kill" and use both predicates A and A' at the same time. This would result in the tagged paths

$$\begin{aligned} tp_1'' &= \{s_1, s_2, s_3\} \\ tp_2'' &= \{s_3, s_4\} \\ tp_3'' &= \{s_3, s_5, s_6\} \\ tp_4'' &= \{s_6, s_3\}. \end{aligned}$$

For this simple example, there appears to be no value for introducing more than a minimum number of predicates. However, there are two good reasons one may wish to do so, particularly on larger programs. First, extra predicates may be used to reduce the number of tagged paths and hence the number of verification conditions. To see this, consider the extreme example of the loop-free program having $2m+1$ statements:

$$\begin{aligned} s_i &= \text{test: } \{p_i, i+1, i+2\}, \\ &\qquad \text{for } i = \{1, 3, 5, 7, \cdots, 2m-1\}, \\ s_i &= \text{assign: } \{x_k, f, i+1\}, \\ &\qquad \text{for } i = \{2, 4, 6, \cdots, 2m\}, \end{aligned}$$

and

$$s_{2m+1} = \text{halt}.$$

The program consists of m repetitions of the form shown in Fig. 2. Considering only the initial and final predicate tags, the program has 2^m distinct tagged paths. If one associates a predicate A_i with each s_i, $i=1, 3, 5, \cdots, 2m-1$, the number of tagged paths drops to $2m$.

The second reason one may wish to introduce extraneous predicates is to reduce one large, possibly difficult verification condition to two or more simpler ones. In the example program P', one can associate tp_1 with tp_1'', tp_2 with tp_2'', and consider tp_3'' and tp_4'' as two components of tp_3. The original verification of tp_3 involved working over $\{s_3, s_5, s_6, s_3\}$

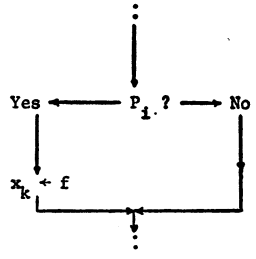

Fig. 2. Flow diagram of repetitive program segment.

whereas in verifying tp_3'', one deals with $\{s_3, s_5, s_6\}$ and then, in verifying tp_4'', continues with $\{s_6, s_3\}$.

Remarks

The sequences of predicates associated with program paths were derived by moving "forward" in the same direction as the flow of control in the program execution. The derivation can, in fact, be done in the opposite direction to the flow ("backward") or even in the general form of both ends of the path toward the middle. A verification condition is generated by each "collision" of predicates.

Extensions to this method can be defined for sophisticated programming languages providing for such constructs as subroutines (procedures), statements with "side effects," recursive procedures, block structure, etc. Most such constructs simply provide more complicated ways of associating variables and values, more complicated assignments of values, and more complicated flow of control. The careful mapping of these properties onto the derivation of the predicate sequences is, in theory, all that is required.

These methods have, in fact, been used to construct a computer program which is capable of proving the correctness of many simple, yet nontrivial programs written in an Algol-like programming language over integers [4].

Acknowledgment

The author wishes to thank Prof. R. W. Floyd for first presenting the ideas explained here and for advising the author on his thesis from which this paper is drawn.

References

[1] D. C. Cooper, "Program scheme equivalences and second order logic," presented at the 4th Annu. Mach. Intelligence Workshop, Univ. Edinburgh, Edinburgh, Scotland, Aug. 1969.
[2] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Appl. Math., Amer. Math. Soc.*, vol. 19, pp. 19–32, 1967.
[3] ——, "The verifying compiler," *Comput. Sci. Res. Rev.*, Carnegie-Mellon Univ., Pittsburgh, Pa., Dec. 1967, pp. 18–19.
[4] J. C. King, "A program verifier," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., Sept. 1969.
[5] Z. Manna, "The correctness of programs," *J. Comput. Syst. Sci.*, May 1969.
[6] J. von Neumann and H. H. Goldstine, "Planning and coding problems for an electronic computing instrument," in *Collected Works of John von Neumann*, vol. 5, A. H. Taub, Ed. New York: Pergamon, 1961, pp. 80–235.