STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM - 191
STAN-CS-73-341

A HEURISTIC APPROACH TO PROGRAM VERIFICATION

BY

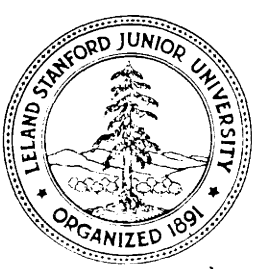
SHMUEL M. KATZ AND ZOHAR MANNA
WEIZMANN INSTITUTE OF SCIENCE

SUPPORTED BY

ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

MARCH, 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-191

March 1973

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS-341

A HEURISTIC APPROACH TO PROGRAM VERIFICATION

by

Shmuel M. Katz and Zohar Manna
Applied Mathematics Department
Weizmann Institute of Science

Abstract: We present various heuristic techniques for use in proving the correctness of computer programs. The techniques are designed to obtain automatically the "inductive assertions" attached to the loops of the program which previously required human "understanding" of the program's performance. We distinguish between two general approaches: one in which we obtain the inductive assertion by analyzing predicates which are known to be true at the entrances and exits of the loop (top-down approach), and another in which we generate the inductive assertion directly from the statements of the loop (bottom-up approach).

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract SD-183.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151.

May, 1973

A HEURISTIC APPROACH TO PROGRAM VERIFICATION

by

SHMUEL M. KATZ and ZOHAR MANNA
Applied Mathematics Department
Weizmann Institute of Science
Rehovot, Israel.

Abstract. We present various heuristic techniques for use in proving the correctness of computer programs. The techniques are designed to obtain automatically the "inductive assertions" attached to the loops of the program which previously required human "understanding" of the program's performance. We distinguish between two general approaches: one in which we obtain the inductive assertion by analyzing predicates which are known to be true at the entrances and exits of the loop (top-down approach), and another in which we generate the inductive assertion directly from the statements of the loop (bottom-up approach).

I. Introduction

The desirability of proving that a given program is correcthas been noted repeatedly in the computer literature. Floyd [1967] has provided a proof method for showing partial correctness of iterative (flowchart) programs, that is, it shows that if the program terminates, a given input -output relation is satisfied. The method involves cutting each loop

of the program, attaching to each cutpoint an "inductive assertion" (which is a predicate in first-order predicate calculus), and constructing verification conditions for each path from one assertion to another (or back to itself). The program is partially correct if all the verification conditions are valid. Elements of these techniques have been shown amenable to mechanization. King [1969], for example, has actually written a 'verifier' program which, given the proper inductive assertions for programs written in a simplified Algol-like language, can prove partial correctness. Thus, it is fairly clear that the parts of this method which involve generating verification conditions from inductive assertions and then proving or disproving their validity is a difficult but programmable problem. However, as King puts it, finding a set of assertions to 'cut' each loop of the program "depends on our deep understanding of the program's performance and requires some sophisticated intellectual endeavor".

In this paper we show some general heuristic techniques for automatically finding a set of inductive assertions which will allow proving partial correctness of a given program. More precisely, we are given a flowchart program with input variables $\bar{x}$ (which are not changed during execution), program variables $\bar{y}$ (used as temporary storage during the execution of the program), and output variables $\bar{z}$ (which are assigned values only at the end of the execution). In addition, we are given "input predicate" $\phi(\bar{x})$, which puts restrictions on the

input variables, and "output predice" $\psi(\bar{x},\bar{z})$, which indicates the desired relation between the input and output variables. Given a set of cutpoints which cut all the loops, our task is to attach an appropriate inductive assertion Q_i to each cutpoint i.

We distinguish between two general approaches:

(a) top-down approach in which we obtain the inductive assertion inside a loop by analyzing the predicates which are known to be true at the entrances and exits of the loop, and

(b) bottom-up approach in which we generate the inductive assertion of a loop directly from the statements of the loop.

For "toy" examples, having only a single loop, it is generally clear that the top-down approach is the natural method to use. However, this is definitely not the case for real (non-trivial) programs with more complex loop structure. In this case some bottom-up techniques were found indispensible. Most commonly we have found it necessary to combine the two techniques, with the bottom-up methods dominant.

Preliminary attempts to attack the problem of finding assertions have been made by Floyd [private communication], and Cooper [1971]. Heuristic rules basically similar to some of our top-down rules have been discovered independently by Wegbreit [1973]. Elspas, et.al. [1972], used "difference

equations" derived from the program's statements which is, in essence, a bottom-up approach.

We handle programs with arrays separately, since generating assertions involving quantification over the indices of arrays requires special treatment. Thus in Section II we discuss heuristic techniques for flowchart programs without arrays, while in Section III we extend the treatment to programs with arrays. In Section IV (conclusion) we discuss open problems and possible implications of our techniques. Related problems where these approaches seem applicable include proving termination of programs, and discovering the input and output assertions of a program.

Our emphasis in this paper is on the exposition of the rules themselves and we are purposely somewhat vague on other problems, such as correctly locating the cutpoints or ordering the application of the rules. Though we do not enter into details, we assume that whenever possible we conduct immediate tests on the consistency (with known information) of a new component for an assertion as soon as it is generated, and that algebraic simplifications and manipulations are done whenever necessary.

II. Heuristics for Programs without Arrays

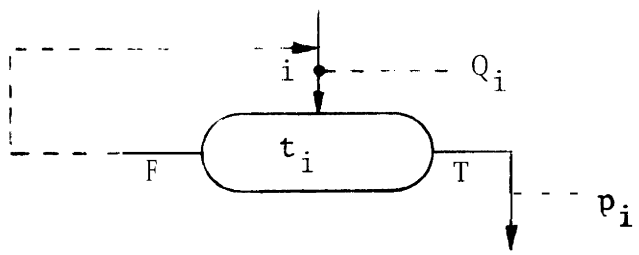
A. Top-down approach. We begin by listing the top-down rules, which may be divided into two classes: entry rules and exit rules.

1. Entrv rules. These rules are intuitively obvious, but provide valuable information in a surprising number of cases.

rule Enl. Any conjunct* in the input predicate $\phi(\bar{x})$ may be added to any Q . It need not be proven since the input variables are not changed inside the program.

rule En2. Any predicate known to be true upon first reaching a cutpoint i should be tried in Qi .

2. Exit rules. For simplicity in the statement of these rules, we assume that a cutpoint is attached to the arc immediately before an exit test of the loop. Thus we may consider an exit from a loop to be of the form



where t_1 is the exit test, p_i is some conjunct of a predicate known to be true when the exit test first holds, i is the cutpoint on the arc leading into the exit test, and Q_i is the assertion which we wish to discover. We attempt to extract

---

* If a predicate is expressed as a conjunction $A_1 \wedge A_2 \wedge \ldots \wedge A_n$, then each A_i is a conjunct of the predicate.

information from p_i and t_i in order to find an assertion for the cutpoint i. . The exit rules will lead to a predicate R which is guaranteed only to satisfy

$$t_i \wedge R \supset p_i \quad ;$$

we then must show that the R obtained is indeed a valid assertion.

rule Exl . If p_1 is not identical to t_i , let R be p_i itself .

rule Ex2. (transitivity) Although this rule could be applicable to a wider class of operators and relations, we restrict the treatment to inequalities. Suppose p_i has the form $a_1 A a_2$ and t_i has the form $b_1 B b_2$, where a_j and b_j are any terms and A,B are equality or inequality relations. If one of the a_j 's is identical to one of the b_j 's , try to find an appropriate inequality or equality relation R so that $t_i \wedge R \supset p_i$ becomes true. For example, if ti is $x < y_2$ and p_1 is $x < (y_1+1)^2$, then we let R be $y_2 \leq (y_1+1)^2$ since $x < y_2 \wedge y_2 \leq (y_1+1)^2 \supset x < (y_1+1)^2$ is true.

We may extend rule Ex2 and use in our search for R any conjunct attached to cutpoint i which has somehow been previously verified (i.e., it is true upon entry to the loop, and is invariant going around the loop, but does not yet imply the exit predicate p_i). For example, if the conjunct $y_2 = y_3 \cdot x_2$

has been previously verified at cutpoint i, and t_i is the test $y_3 = 1$, while p_i is $y_1 < x_2$, then we may try R being $y_1 < y_2$, since $y_2 = y_3 \cdot x_2 \wedge y_3 = 1 \wedge y_1 < y_2 \supset y_1 < x_2$.

Another possible extension of rule Ex2 is to search for additional information on the variables in the exit test. We seek information which along with t_i would imply stronger restrictions on the exit values of those variables. For example, suppose t_i is $y_1 \geq x$, we know that $y_1 < x$ upon first reaching i (i.e., the loop is executed at least once), and y_1 is incremented by 1 at each pass through the loop. Then we let t^* be $y_1 = x$ since $y_1 \geq x \wedge y_1 - 1 < x \supset y_1 = x$ in the integers. Thus, rather than looking for R satisfying $y_1 \geq x \wedge R \supset p_i$, it suffices to find an R satisfying $y_1 = x \wedge R \supset p_i$.

rule Ex3. If rule Ex1 fails, a natural "weaker" attempt could be to let R be $t_i \supset p_i$. This rule is sometimes of practical use; however, it says very little about the computation taking place in the loop. Our strategy would give this rule a low priority, trying other rules with stronger resultant claims first.

It is possible to continue and design rules for obtaining R for specific forms of p_i, but since our aim is to explain the general tone of these techniques, we will not go into further details in this direction.

B. Bottom-up Approach.

All of the rules given above have in common that they expect to be provided with some information on either what conditions were true upon entering the loop or what conditions were expected to hold upon completing the loop (or both). However, it is possible to produce conjuncts of the assertion Q without considering predicates already established elsewhere in the program, In order to accomplish this goal we shall look for a predicate which is an invariant of the loop L , i.e., it remains true upon repeated executions of the loop.

Clearly, any conjunct in the inductive assertion of a loop must be an invariant of the loop. However, in the top-down rules this is usually the last fact which is established about a perspective assertion. In the pure bottom-up approach, assertions which arise "naturally" from the computations in the loop are directly generated -- and only afterward checked for relevance to the overall proof.

Most invariants may be traced back to the fact that at any stage of the computation, those assignment statements which are on the same paths through the loop have been executed an identical number of times, and this is a 'constant' which may be used to relate the variables iterated.

For an assignment statement $y_i \leftarrow f(\bar{x},\bar{y})$ we let $v_i^{(n)}$ denote the value of y_i after n executions of the statement,

while $y_i^{(0)}$ indicates the "initial" value of y_i upon first reaching a given cutpoint of the loop.

Our technique for finding invariants involves constructing an "operator table" in which we record useful information for each operator. Among the entries for an operator are its definition (using "weaker" operators), a description of a general computation after n iterations, and other common identities which facilitate simplications. For example, for + our table will include the fact that for an assignment statement of the form yi $\leftarrow y_i+k$, in general $y_i^{(n)} = y_i^{(0)} + \sum_{j=1}^{n} k^{(j-1)}$ where $k^{(j-1)}$ is the value of k before the j-th iteration of the assignment statement. Important identities are also noted including that for a constant c, $\sum_{i=1}^{n} c = c \cdot n$, and that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Rules for producing invariants linking variables which receive assignments on different paths through the loop are presently being developed. Here we present rules only for the simple case of variables changed only on the same paths through the loop.

rule Il. (invariant) To construct an invariant, given a group of assignments* $(y_1, \ldots, y_\ell) \leftarrow (f_1(\bar{x},\bar{y}), \ldots, f_\ell(\bar{x},\bar{y}))$, we consider

---

* The above notation implies that the value of $f_i(\bar{x},\bar{y})$ is assigned to y_i for all i 's simultaneously.

those variables y_j, $1 \leq j \leq \ell$, which are not changed elsewhere in the loop. Using the operator table we express the value of each y_j after n iterations, i.e. $y_j^{(n)}$. We then attempt to find a factor common to two expressions in order to obtain a usable relationship between $y_i^{(n)}$ and $y_k^{(n)}$. The relation obtained after substituting the initial values of y_i and y_k at cutpoint A for $y_i^{(0)}$ and $y_k^{(0)}$, respectively, and removing the superscript (n) is an invariant of the loop. It also holds for the initial values of the variables at A and thus may be added to Q_A.

For example, if y_1 and y_2 are changed only in the assignments $(y_1, y_2) \leftarrow (y_1 + x \cdot y_3, y_2 + 5 \cdot y_3)$ inside a loop, then $y_1^{(n)} = y_1^{(0)} + x \cdot \sum_{i=1}^{n} y_3^{(i-1)}$ and $y_2^{(n)} = y_2^{(0)} + 5 \cdot \sum_{i=1}^{n} y_3^{(i-1)}$.

Therefore, for $x \neq 0$, $\dfrac{y_1^{(n)} - y_1^{(0)}}{x} = \sum_{i=1}^{n} y_3^{(i-1)} = \dfrac{y_2^{(n)} - y_2^{(0)}}{5}$.

Assuming we know that the initial values of y_1 and y_2 upon first reaching the cutpoint are $y_1^{(0)} = 1$ and $y_2^{(0)} = 0$, we obtain the invariant $5(y_1 - 1) = x \cdot y_2$. If the assignments were $(y_1, y_2) \leftarrow (2 \cdot y_1, y_2/2)$ then $y_1^{(n)} = y_1^{(0)} \cdot \prod_{i=1}^{n} 2$ and $y_2^{(n)} = y_2^{(0)} \cdot \prod_{i=1}^{n} (\frac{1}{2})$. Simplifying, we obtain $y_1^{(n)} = y_1^{(0)} \cdot 2^n$ and $y_2^{(n)} = y_2^{(0)} \cdot \dfrac{1}{2^n}$, therefore

$\frac{y_1^{(n)}}{y_1^{(0)}} = 2^n = \frac{y_2^{(0)}}{y_2^{(n)}}$. Thus given that $y_1^{(0)} = 1$ and $y_2^{(0)} = x$ $(x \neq 0)$ we get that $y_1 \cdot y2 = x$ is an invariant.

rule 12. Whenever $y_i^{(n)}$ may be expressed in terms of only $y_i^{(0)}$ and n, i.e., $y_i^{(n)} = f(y_i^{(0)}, n)$, and the value of $y_i^{(0)}$ at the cutpoint A is known to be m, then replacing $y_i^{(0)}$ by its value and removing superscript (n), we may obtain the invariant $\exists n[n \geq 0 \wedge yi = f(m,n)]$. Variables iterated simultaneously may be quantified by the same n. For example, in the second example of 11, $\exists n[n \geq 0 \wedge y_1 = 2^n \wedge y_2 = x/2^n]$ is an invariant of the loop.

Our heuristic rules are all relevant to programs having an arbitrary number of loops, and an arbitrary complex 'topology', although, of course, they will yield valid inductive assertions more often and more immediately in a simple program.

One of the problems in applying the rules is deciding what order is preferable. In particular, it has been found that many terms of the assertion may be obtained both by the bottom-up rules and by repeated use of the top-down rules. However, usually one method will yield the result immediately, while considerable effort is expended if the other method is applied first. Experience shows that there is a need for interaction between the top-down and bottom-up approaches

For example, we may use established invariants to deduce the relation R in the top-down rule Ex2, and on the other hand, we may direct the search for particular invariants based on variables or operators which appear in p_i.

C. Examples. We demonstrate the rules listed so far on a few examples.

Example 1: Integer square root. The program in Figure 1 computes $z = \lfloor \sqrt{x} \rfloor$ for every natural number x. That is, the final value of z is the largest integer k such that $k \leq \sqrt{x}$. We show partial correctness for $\phi(x) : x \geq 0$ and $\psi(x,z)$: $z^2 \leq x \wedge x < (z+1)^2$. Clearly, $y_1^2 \leq x \wedge x < (y_1+1)^2$ is required to be true after exit from the loop. We first try the top-down approach. By rule Ex1 we attempt adding the conjunct $y_1^2 \leq x$ to Q. The verification condition $x \geq 0 \supset y_1^2 \leq x$ is, in fact, true for the initial value of y_1 at the cutpoint, i.e., $y_1 = 0$. For the moment we do not attempt to verify that it is an invariant of the loop. Considering the second conjunct of the predicate, $x < (y_1+1)^2$, an attempt to apply Ex1 fails because this relation is not true for the values of the variables when the cutpoint is first reached. Since the exit test $y_2 > x$ and the predicate $x < (y_1+1)^2$ both contain x, we apply rule Ex2. We find that $y_2 \leq (y_1+1)^2$ is the desired relation since $y_2 > x \wedge y_2 \leq (y_1+1)^2 \supset x < (y_1+1)^2$ is a valid statement. $y_2 \leq (y_1+1)^2$ is satisfied for the initial values of the variables.

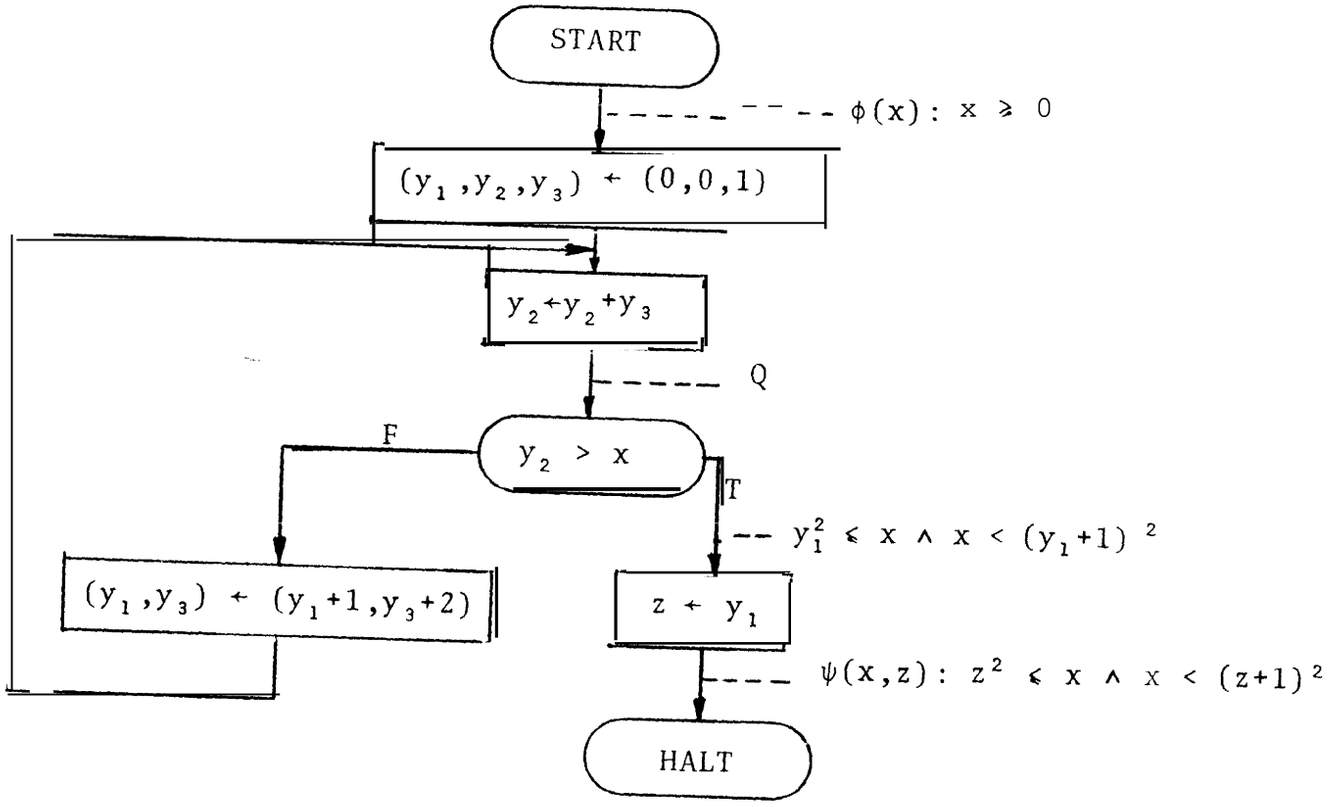


Figure 1. Integer Square-Root Program.

However, an attempt to prove the validity of

$Q : y_1^2 \leq x \wedge y_2 \leq (y_1+1)^2$ does not yet succeed.

At this point we try to use the bottom-up approach, i.e. try to find invariants. We note that the assignments along one pass of the loop may be combined into the single group of assignments $(y_1, y_2, y_3) \leftarrow (y_1+1, y_2+y_3+2, y_3+2)$. From the operator table we obtain the equations

$$(1) \qquad y_1^{(n)} = y_1^{(0)} + \sum_{i=0}^{n} 1 = 0+n = n$$

$$(2) \qquad y_2^{(n)} = y_2^{(0)} + \sum_{i=1}^{n} (y_3^{(i-1)}+2) = 1+2n + \sum_{i=1}^{n} y_3^{(i-1)}$$

$$(3) \qquad y_3^{(n)} = y_3^{(0)} + \sum_{i=1}^{n} 2 = 1+2n \quad .$$

We may use equation (3) to substitute for $y_3^{(i-1)}$ in equation (2), and simplify to

$$(2') \qquad y_2^{(n)} = 1+2n + \sum_{i=1}^{n} [1+2(i-1)] = 1+2n+n + \frac{2(n-1)n}{2} =$$

$$= 1+2n+n^2 = (1+n)^2 \quad .$$

In the simplification above known facts about the summation operator (obtained from the operator table) are used.

Since $y_1^{(n)} = n$, we obtain

$$y_3^{(n)} = 1+2y_1^{(n)} \wedge y_2^{(n)} = (1+y_1^{(n)})^2$$

for every n, i.e., $y_3 = 1+2y_1 \wedge y_2 = (1+y_1)^2$ are invariants of the loop and should be added to the trial Q. Q becomes $y_1^2 \leq x \wedge y_3 = 2y_1+1 \wedge y_2 = (y_1+1)^2$, which will prove the partial correctness of the program.

Example 2: Division within tolerance. The program of Figure 2 divides x_1 by x_2 within tolerance x_3. We try first to find invariants. Considering the assignments $(Y, ,y_3) \leftarrow (y_2/2, y_3/2)$, we obtain the equations $y_2^{(n)} = y_2^{(0)} \cdot \frac{1}{2^n}$ and $y_3^{(n)} = y_3^{(0)} \cdot \frac{1}{2^n}$. Therefore $\frac{y_2^{(n)}}{y_2^{(0)}} = \frac{1}{2^n} = \frac{y_3^{(n)}}{y_3^{(0)}}$. Since $y_2^{(0)} = \frac{x_2}{4}$ and $y_3^{(0)} = \frac{1}{2}$ at the cutpoint, we obtain $2y_2^{(n)} = x_2 \cdot y_3^{(n)}$. Thus $2y_2 = x_2 \cdot y_3$ is the first conjunct in the trial Q. Next we consider the assignments $(y_1, y_4) \leftarrow (y_1+y_2, y_4+y_3/2)$. In order to be able to find a common factor in the equations for $y_1^{(n)}$ and $y_4^{(n)}$ we first eliminate y_2 by using the already established invariant $y_2 = x_2 \cdot y_3/2$, obtaining $(y_1, y_4) \leftarrow (y_1+x_2 \cdot y_3/2, y_4+y_3/2)$. Now we get $y_1^{(n)} = y_1^{(0)} + \sum_{i=1}^{n} \frac{y_3^{(i-1)}}{2}$ and $y_4^{(n)} = y_4^{(0)} + \sum_{i=1}^{n} \frac{y_3^{(i-1)}}{2}$. Eliminating the common term $\sum_{i=1}^{n} \frac{y_3^{(i-1)}}{2}$, the result is

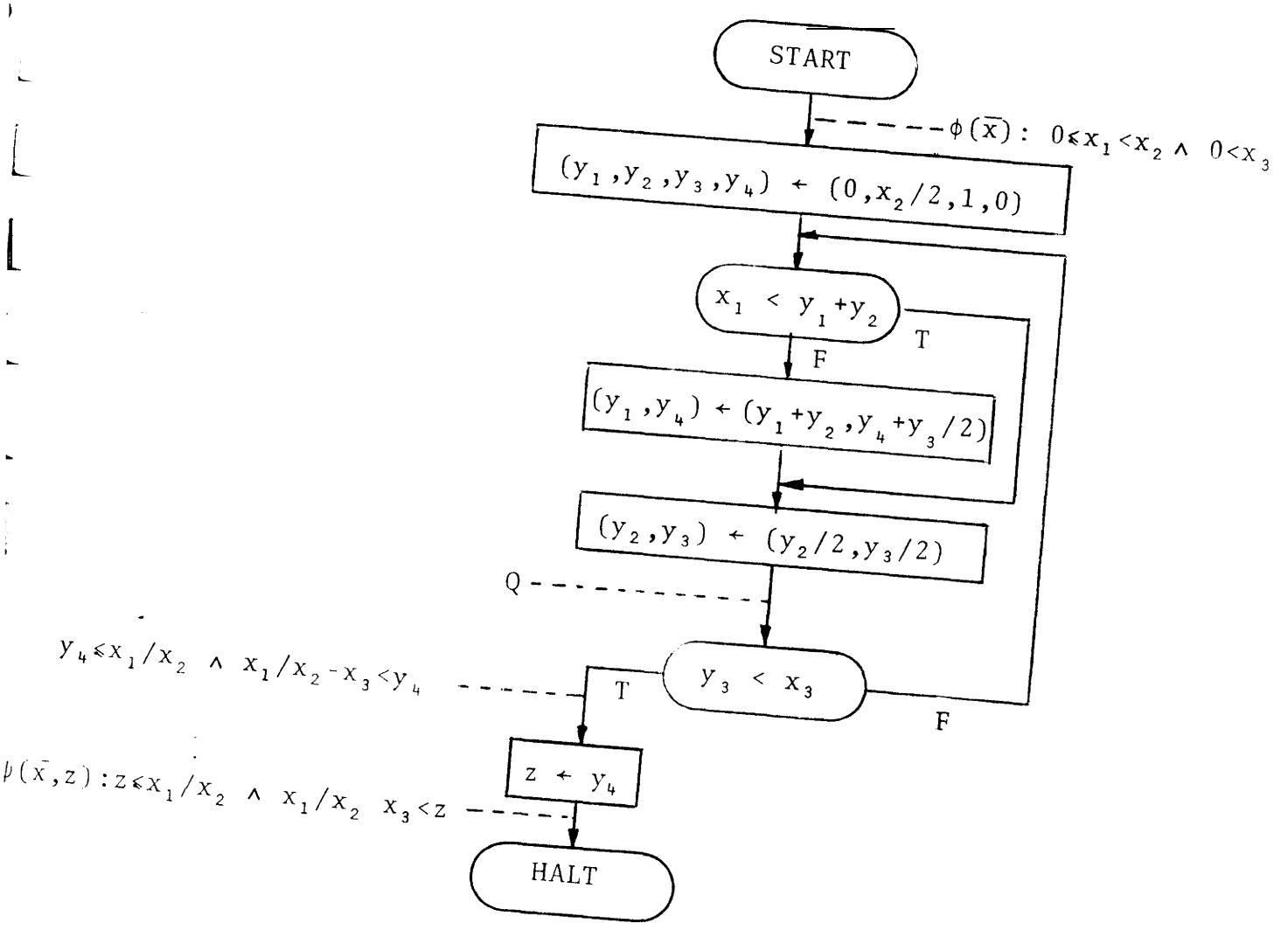


Figure 2. Real Division within Tolerance Program

$\frac{y_1^{(n)} - y_1^{(0)}}{x}$ $y_4^{(n)}$ $y_4^{(0)}$ Since there are two possible paths to initially reach the cutpoint, the pair $(y_1^{(0)}, y_4^{(0)})$ may be either $(0,0)$ or $(\frac{x_2}{2}, \frac{1}{2})$. In either case, the simplified expression becomes $y_4^{(n)} = y_1^{(n)}/x_2$. Therefore $y_4 = y_1/x_2$ is added as a conjunct to the trial Q.

Since no further information can be gained from the invariant rules, we turn to the top-down rules. We have $y_4 \leq x_1/x_2 \wedge x_1/x_2 - x_3 < y_4$ true upon exit from the loop. Trying Ex1 on $y_4 \leq x_1/x_2$, the conjunct can be seen to hold initially at the cutpoint by cases, since if $x_1 < x_2/2$ then y_4 is initially 0 at the cutpoint and by ϕ we have $0 \leq x_1/x_2$, while if $x_1 \geq x_2/2$, then $1/2 \leq x_1/x_2$ and y_4 is 1/2 at the cutpoint. Thus by Ex1 we may add $y_4 \leq x_1/x_2$ to Q. The second conjunct, $x_1/x_2 - x_3 < y_4$, on the other hand, does not hold initially so we try Ex2. The necessary 'transitive' relation is found to be $x_1/x_2 - y_3 \leq y_4$ since $y_3 < x_3 \wedge x_1/x_2 - y_3 \leq y_4 \supset x_1/x_2 - x_3 < y_4$. We note that $x_1/x_2 - y_3 \leq y_4$ holds for the initial values at the cutpoint so we add it to Q. Q is now $y_2 = x_2 \cdot y_3/2 \wedge y_4 = y_1/x_2 \wedge y_4 \leq x_1/x_2 \wedge x_1/x_2 - y_3 \leq y_4$ which will prove the program partially correct.

Example 3. Hardware (integer) division. The program of Figure 3 is a simulation of how integer division might be carried

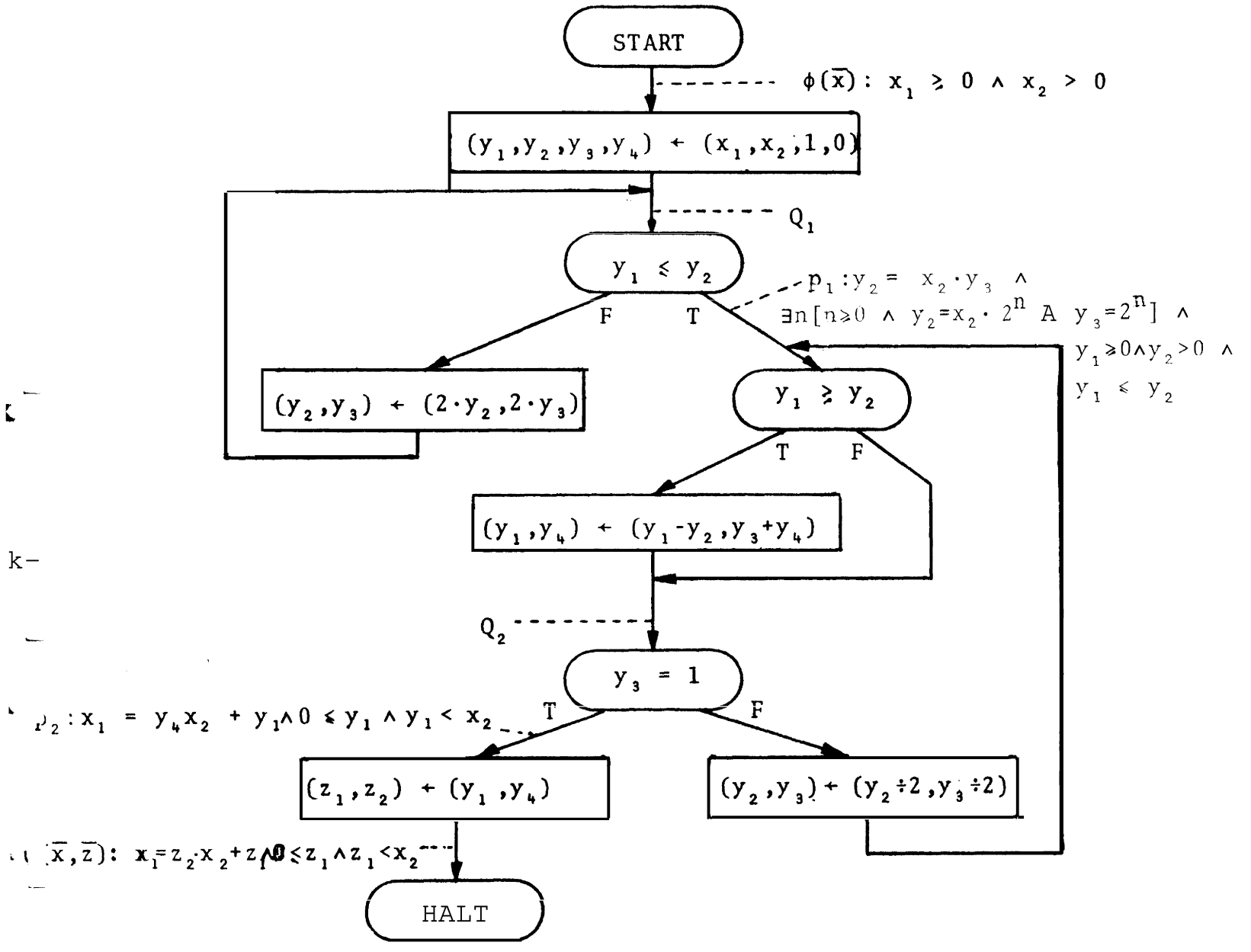


Figure 3. Hardware (Integer) Division Program.

out by a computer. The 'division by 2' represents a 'shift-right,, and the 'multiplication by 2' a 'shift-left'. Although the second loop of this example is similar to the program of Example 2, we bring it in order to illustrate how programs with more than one loop may be handled, and how complications which could arise from integer division may be solved with the aid of the invariant rule. Our strategy is to obtain a maximum amount of information from the first loop, which will be true upon entrance to the second loop. Then top-down rules can be used conveniently for the second loop.

In the-first loop we attempt to link y_2 and $Y3$, obtaining $y_2^{(n)} = x_2 \cdot 2^n$ and $y_3^{(n)} = 1 \cdot 2^n$ which leads to the invariant $y_2 = x_2 \cdot y_3$ by rule I1. By rule 12 we also have the conjunct $\exists n[n \geq 0 \wedge y_2 = x_2 \cdot 2^n \wedge y_3 = 2^n]$. We now consider top-down rules. Since $y_1 \geq 0 \wedge y_2 > 0$ holds initially, it is added by rule En2 to Q_1, which thus becomes the valid invariant $y_2 = x_2 \cdot y_3 \wedge \exists n[n \geq 0 \wedge y_2 = x_2 \cdot 2^n \wedge y_3 = 2^n] \wedge y_1 \geq 0 \wedge y_2 > 0$. All this information, as well as $y_1 \leq y_2$ is a predicate p_1 true upon first reaching the second loop. Recall that for the entrance rules we consider the predicates true upon first reaching the cutpoint i. Thus the information in p_1 must be 'moved' along the paths to cutpoint 2.

$y_2 > 0 \wedge y_2 = x_2 \cdot y_3 \wedge \exists n[n \geq 0 \wedge y_2 = x_2 \cdot 2^n \wedge y_3 = 2^n]$ are unchanged by either path to 2, while y_1 might be changed but $y_1 \geq 0$ can be seen to remain true by inspection. If the

right path is taken, $y_1 \leq y_2$ is strengthened to $y_1 < y_2$, while the left path may be used only if $y_1 = y_2$. In this case y_1 is set to zero, and since $y_2 > 0$ is known, in either case $y_1 < y_2$ at cutpoint 2. At this point, all the necessary assertions for handling the second loop are already given explicitly in the entry and exit predicates. Using rule En2 we obtain $Q_2 : y_2 = x_2 \cdot y_3 \wedge \exists n[n \geq 0 \wedge y_2 = x_2 \cdot 2^n \wedge y_3 = 2^n] \wedge y_1 \geq 0 \wedge y_2 > 0 \wedge y_1 < y_2$, while from Ex1 we add $x_1 = y_4 \cdot x_2 + y_1$ to Q_2. This Q_2 will be a good inductive assertion.

The rule involving n, obtained by 12, is necessary here in order to guarantee that the conjunct $y_2 = x_2 \cdot y_3$ is valid, because of the 'shift-right' division. We clearly could have obtained some of the conjuncts in Q by other rules. For example, $y_1 < y_2$ could have been obtained by rule Ex2 (because p_2 contains $y_1 < x_2$, $y_2 = x_2 \cdot y_3$ is an invariant, and $y_3 = 1$ is the exit test), or $x_1 = y_4 \cdot x_2 + y_1$ by rule I1.

III. Heuristics for Arrays

The problem of finding assertions involving arrays is quite different from that of finding assertions for simple variables because an array assertion generally will be an entire family of claims. This is the reason most assertions about arrays will involve quantifiers. All rules in Section II continue, of course, to be applicable for those variables not in arrays. In addition, rules En1, En2, Ex1 and Ex3 may be used for assertions with arrays.

Underlying the heuristics which follow is the assumption that arrays in a program are used "properly,', i.e., to treat a large number of variables in a uniform manner, and not just as a collection of unrelated variables fulfilling different roles in the program. The further assumption is usually made that an assertion about an array will be of the form

$$\forall j\,[\langle j\text{-index}\rangle \supset \langle j\text{-array}\rangle] \quad \text{or} \quad \exists j\,[\langle j\text{-index}\rangle \wedge \langle j\text{-array}\rangle] ,$$

where <j-index> is a claim on the indices of the array and <j-array> is the claim which is made about the array elements themselves. We often separate the two problems of finding the <j-index> and of finding the <j-array>.

As in Section II, we distinguish between the top-down and bottom-up approaches.

In order to apply some of the array rules it is convenient to first determine the "one-pass" assertion, i.e., the claim

which can be made about the effect on the arrays of one circuit through the loop. This claim is often not difficult to establish, in particular for loops which do not contain other loops since then the circuit through the loop is a simple sequence of statements. The assertion can be most easily established by the known technique of "backward substitution", moving backwards around the loop past each assignment statement.

A. Top-down rules. As noted above, all previous top-down rules, except for the transitivity rule Ex2 (which involves inequalities), are directly applicable for arrays. In the rules listed below, p denotes an assertion with quantification concerning an array which is true after exit from the loop, while P' is an assertion like p , but true upon entrance to the loop. Q denotes the desired loop assertion. Rules A1, A2, and A4 attempt to either transform or create assertions p and P' having a form which will facilitate generating Q by rule A3.

rule A1. Let p be a claim about a specific element of an array, say S[c] (and thus not necessarily including quantifiers). We rewrite it as $\exists j\ [c \leqslant j \leqslant c \wedge$ <j-array>] , where <j-array> is p with j in place of c . Similarly, if a p' as above is true upon entrance to the loop, we rewrite it as $\forall j\ [c \leqslant j \leqslant c \supset$ <j-array>] .

The underlying principle here is that a claim whose <j-index> is made smaller by the loop probably has an existential

quantifier (we are "looking for something"), while if the <j-index> is extended to cover more elements by the loop, the claim probably has a universal quantifier (we want something to be true for a larger part of the array). Thus we may check the feasibility of the resulting assertion by determining whether the <j-array> is in fact expanded or contracted in the loop. This principle is also used in the bottom-up rules.

rule AZ. Given a p, we examine the definitions of the operators and relations in p and whatever information is known about the array upon first reaching the loop. Using this information we produce the <j-index> for a p' which must be true upon entrance to the loop, and has a <j-array> identical to P. For example, if p is $\exists j[1 \leq j \leq 3 \text{ A } A[j] = \max(A[1],...,A[n])]$, and we know only that A is defined upon entrance to the loop, by the rule we require a <j-index> such that $A[j] = \max(A[1],...,A[n])$ must be true. By the definition of max we can determine that the maximum element must belong to the array. Thus $\exists j[1 \leq j \leq n \text{ A } A[j] = \max(A[1],...,A[n])]$ is the parallel assertion upon entrance to the loop.

In some cases of a predicate p with universal quantifiers, the corresponding initial claim may require a <j-index> which is empty (so that the overall claim is vacuously true). For example, if p is $\forall j\,[1 \leq j < n \supset A[j] \leq A[j+1]]$, and we have not sorted A before the loop, p' might be

$\forall j[1 \leq j < 1 \supset A[j] \leq A[j+1]]$.

Rule A2 is the only example in this paper of a rule which enables us to project "backwards" to find the minimal conditions which must hold upon entrance to a given loop. Such rules should be useful not only to aid in discovering the correct assertion for the loop in question, but also to carry information backwards for loops earlier in the program. Thus further investigation of this general technique is warranted.

rule A3. If $=p$ contains a term r as a boundary of the indices, and we have determined that for some term s , $s = r$ upon exit from the loop (by any of the rules in Section II) , we let Q be the predicate obtained by substituting s for some appearances of r in p .

Similarly, if p' contains a term r , and $s = r$ upon entrance to the loop, let Q be the predicate obtained by substituting s for some appearances of r in p'.

For example, if p is $\forall i[1 \leq i \leq m \supset A[i] \leq A[m]]$, and $\ell = m$ is the exit test of the loop, we could try letting Q be either $\forall i[1 \leq i \leq \ell \supset A[i] \leq A[m]]$,
$\forall i[1 \leq i \leq m \supset A[i] \leq A[\ell]]$ or $\forall i[1 \leq i \leq \ell \supset A[i] \leq A[\ell]]$.

Obviously, if information is known about both p and p', the application of A3 can often be directed by matching the results of various substitutions until the entrance and exit claims are identical. Thus, if there are several possibilities

for substition, we may decide for which appearances of terms in P or p' to substitute.

We would like to be able to also use the transitivity rule Ex2 for an array assertion with quantification (specifically, a p with an inequality as its <j-array>). This requires establishing that for each pair of terms compared, we may find a third term such that there will be two new inequalities, true upon exit from the loop, which will imply the original inequality (as in Ex2).

rule A4. Given a p with universal quantifier and an inequality including arrays as its <j-array>, we use the "one-pass" assertion to find a term which contains the two needed inequalities for a particular value of j (i.e., for a single pair of values from p). Then let each new inequality be the <j-array> for a claim having the <j-index> of p. The other top-down rules may then be used separately on each of the new inequality claims to obtain the loop assertion.

For example, given $p:\forall i[1 \leq i \leq m \supset A[i] \leq B[i]]$, we might discover a C[k] such that $A[k] \leq C[k] \wedge C[k] \leq B[k]$ for some k, and assume $\forall i[1 \leq i \leq m \supset A[i] \leq C[i]] \wedge \forall i[1 \leq i \leq m \supset C[i] \leq B[i]]$ is true upon exit from the loop. Then, if, for example, $\ell = m$ and $j = 1$ upon exit from the loop, A3 used along with other information could result in $\forall i[1 \leq i \leq \ell \supset A[i] \leq C[i]] \wedge \forall i[j \leq i \leq m \supset C[i] \leq B[i]]$ as the loop assertion.

B. Bottom-up approach. In order to identify which heuristics to use, we must differentiate between two methods of computation:
a) If the exit test has the variable i compared with a term which is not changed inside L, and i is incremented

monotonically inside L , then it is assumed to be a counter controlling the loop in an "iterative going up" computation. (b) If the variable i is compared with a term which does not change in the loop, and is decremented monotonically inside L , then i is a counter controlling the loop in an "iterative going down" computation.

In the rules below we assume all loops have the index i , and let i_0 denote the value of i when it first reaches the cutpoint of the loop, while i_1 denotes the value of i upon exit from the loop. As in Section II, we assume that the cutpoint is located immediately before the exit test.

We first list the rules for finding the <j-index> .

rule X1. If i is a counter (incremented by 1) in a "going-up" iteration and is also the variable which appears in the index of array elements receiving assignments, then try assertions of the forms $\forall j [i_0 \leq j < i \supset$ <j-array>] or $\exists j [i < j \leq i_1 \wedge$ <j-array>] in the inductive assertion. These will also be the form of the predicate p which is true after exit from the loop.

If i_0 is known, say $i_0 = c$ upon entrance to the loop, then the c should be substituted for i_0 in Q and p . Similarly, if $i_1 = d$ upon exit from the loop, d should be substituted for i_1 .

rule X2. If i is a counter (decremented by 1) in a "going-down" iteration and is also the variable which appears in the index of array elements. receiving assignments, try assertions of the forms

$\forall j[i < j \leq i_0 \supset$ <j-array>$]$ or $\exists j[i_1 \leq j < i \wedge$ <j-array>$]$.

As in rule Xl, p will also have the above form and i_0 or i_1 should be eliminated if possible.

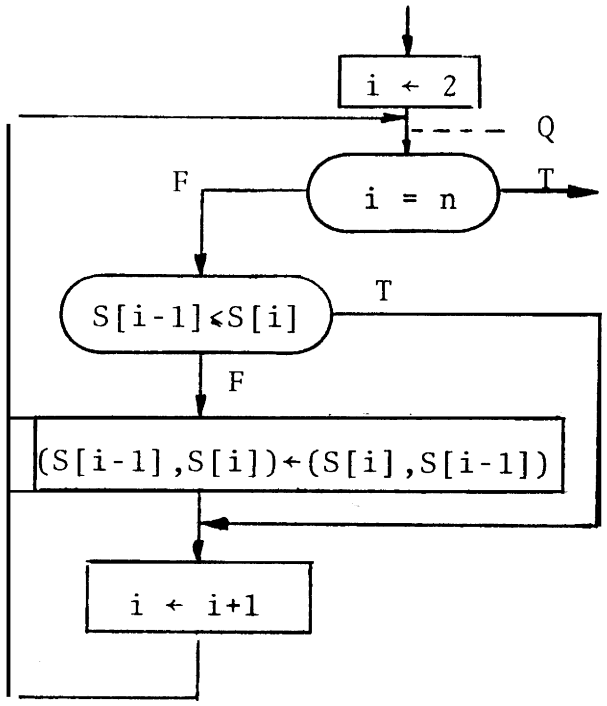
rule X3. Discover whether Xl and X2 fail only because i is assigned a function f(i) rather than merely incremented or decremented by 1 in the loop. If so, try to find the set of elements which i assumes during the loop (using rule 12). The assertion will have the same form as in X1 or X2, except that the <j -index> will include the 12 invariant. For example, if i ← i+7 in the loop, and i is initially zero, then the assertion is

$\forall j\{0 \leq j < i \wedge \exists n[n \geq 0 \wedge j = 7 \cdot n] \supset$ <j-array>$\}$.

The following two conditions are used to decide which of the bottom-up <j -array> rules to apply assuming that the <j -index> has already been determined.

(a) All assignments in the loop are to array elements with indices not specified by the <j-index> before executing the loop. That is, once we have included an element of the array

in the assertion after some circuit, we will make no more assignments to that element in subsequent circuits around the loop.



For example, the program segment at left could be part of a "bubble-sort" program. The "one-pass" assertion is clearly $S[i-1] \leq S[i]$, but if the form of the assertion before executing the loop is

Q : $\forall j[2 \leq j < i \supset$ <j-array>]

the loop violates condition (a) because S[i-1] may receive an assignment and i-1 is already in the domain of the <j-index>.

(b) The "one-pass" assertion can be written as a single conjunct. Furthermore this conjunct is valid for all array elements whose indices are added to the domain of the <j-index> by one circuit through the loop. Thus if the "one-pass" assertion is $S[i] = S[i+1) \wedge S[i+1] \leq S[i+2]$ and i and i+1 are added to the <j-index> by the loop, the condition (b) is not fulfilled because it cannot be expressed by an appropriate single conjunct.

rule R1. If both (a) and (b) are true the "one-pass" assertion itself is taken as the <j-array> . Of course, the quantified variable of the <j-index> must be substituted for the actual array index which appears in the loop. For example, if we have found the assertion to be $\forall j[1 \leq j < i \supset$ <j-array>] and in the loop we have only $A[i] \leftarrow 0$ and then $i \leftarrow i+1$, the "one-pass" assertion is $A[i] = 0$, and (a) and (b) hold. Thus we obtain $\forall j[1 \leq j < i \supset A[j] = 0]$ as the loop assertion.

The following rule is based on the fact that we have already established the desired form of the <j-index> part of the assertion. We want to be able to write one conjunct, say $\forall j[1 \leq j < i \supset$ <j-array>], where the <j-array> will be a statement about (only) the array elements with indices $1 \leq j < i$ and not contain any additional restrictions on the indices.

rule R2. (generalization) If (a) is true, but (b) is not, check whether (b) fails only because the assertion is not a single conjunct. If so, the <j-array> parts of all the conjuncts in the assertion are searched to find the strongest single conjunct which is true for all array elements specified by the known <j-index> . This conjunct becomes the <j-array> . For example, given a one-pass assertion $\forall j[1 \leq j < n \supset A[j-1] < A[j]] \wedge A[n-1] \leq A[n]$ and a required Q of the form $\forall j[1 \leq j < n+1 \supset$ <j-array>], the correct <j-array> by this rule is $A[j-1] \leq A[j]$.

rule R3. If (b) is true, but (a) is not, take the "one-pass" assertion as the <j-array> and consider the effect of an additional pass through the loop on this predicate. Then apply the generalization rule R2 to the result. For example, for the segment of the bubble-sort program introduced above, the one-pass assertion yields

$$\forall j[2 \leq j < i \supset S[j-1] \leq S[j]] \quad .$$

One circuit will change this to:

$$\forall j[2 \leq j < i\text{-}1 \supset S[j-1] \leq S[j]] \wedge S[i-2] \leq S[i] \wedge S[i-1] \leq S[i] \; .$$

Generalizing this predicate by R2 is a relatively difficult step, not yet completely investigated. The generalization procedure would be expected to recognize that no predicate comparing each element with its neighbor is possible, since no information is available about the relation between $S[i-2]$ and $S[i-1]$. Then the transitivity of the inequality would yield that $\forall j[1 \leq j < i \supset S[j] \leq S[i]]$ is the strongest claim which can be made about the entire segment.

Example 4. Minimum of an Array. The program in Figure 4 will find the minimum of an array A using an array S in an unusual way. (The upper half of the array is set to A, and the computation takes place in the lower half, using only comparisons.) For the first loop, top-down rules give no information, so we use bottom-up rules. By X1, we will try the assertion $\forall j[1 \leq j < k \supset \langle j\text{-array}\rangle]$ (because $k_0 = 1$, and we have a "going-up" iteration). The "one-pass" assertion is clearly $S[n+k] = A[k]$, and conditions (a) and (b) are fulfilled. Thus by rule R1 we obtain $Q_1 : \forall j[1 \leq j < k \supset S[n+j] = A[j]]$. Since upon exit from the loop $k = n+2$, we have $p : \forall j[1 \leq j \leq n+1 \supset S[n+j] = A[j]]$. By rule En2, p is added to Q_2. We try rule Ex1 on ψ', but S[1] is undefined on entrance to the loop, so the rule fails.

Using array top-down rules, we first rewrite ψ' as $\exists j[1 \leq j \leq 1 \wedge S[j] = \underline{\min}(A[1],\ldots,A[n])]$ by rule A1. Using A2, we would like to retain the <j-array> part in an assertion true on entrance to the loop. By the definition of $\underline{\min}$ we know that one of the elements is the minimum, and the p we have at the entrance to the loop states that A has been copied to the upper half of S. Thus we obtain $\exists j[n+1 \leq j \leq 2n+1 \wedge S[j] = \underline{\min}(A[1],\ldots,A[n])]$ as the initial assertion which must be true. Since the assignment before the loop implies that $i = n$ upon entrance to the loop, a possible substitution by A3 is

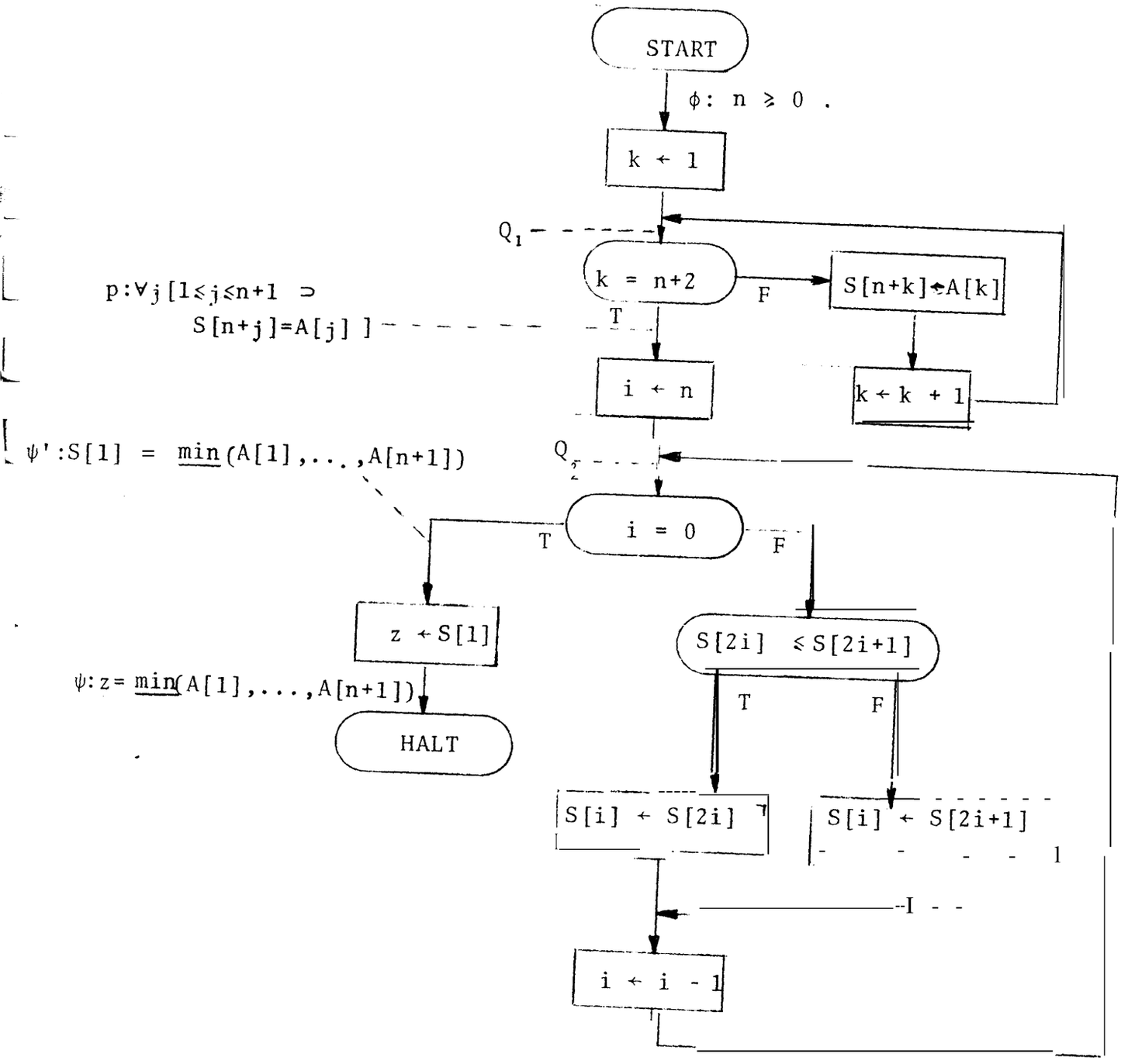


Figure 4. Program for Finding the Minimum of an Array

$q : \exists j[i+1 \leq j \leq 2i+1 \wedge S[j] = \underline{\min}(A[1],\ldots,A[n])]$.

Since $i = 0$ upon exit from the loop, this q becomes identical to ψ'. (Any of the other possible substitutions of i for n will fail to match ψ'.) Thus we let Q_2 be $q \wedge p$. The second conjunct is not needed to prove ψ', but can be retained to provide the additional information that the upper half of S is unchanged by the second loop, and contains A.

Example 5. Partition Program. The program of Figure 5, due to Hoare, will find a partition of the elements of a real array S. We would like to show that it is partially correct with respect to $\phi : n \geq 0$ and $\psi : \forall a \forall b\{0 \leq a < i \wedge j < b \leq n \supset S[a] \leq S[b]\} \wedge j < i$. We use the bottom-up approach, seeking a Q_1 for the large outer loop. Thus we consider one pass through the loop. (It should be noted that the invariants we will find at cutpoints 2 and 3 during the "linear" pass are not necessarily the desired Q_2 or Q_3 for the overall execution of the program.) The first inner loop yields immediately by rules X1 and R1, the invariant $p_1 : \forall k[i_0 \leq k < i \supset S[k] < r]$. Thus upon exit from the first inner loop $p_1 \wedge S[i] \geq r$ is true. Similarly, after the second inner loop, we obtain $p_2 : \forall \ell[j_0 \geq \ell > j \supset S[\ell] > r] \wedge S[j] \leq r$ by X2 and R1. There is no possibility that the second loop could disturb the claim of p_1, because there are no assignment

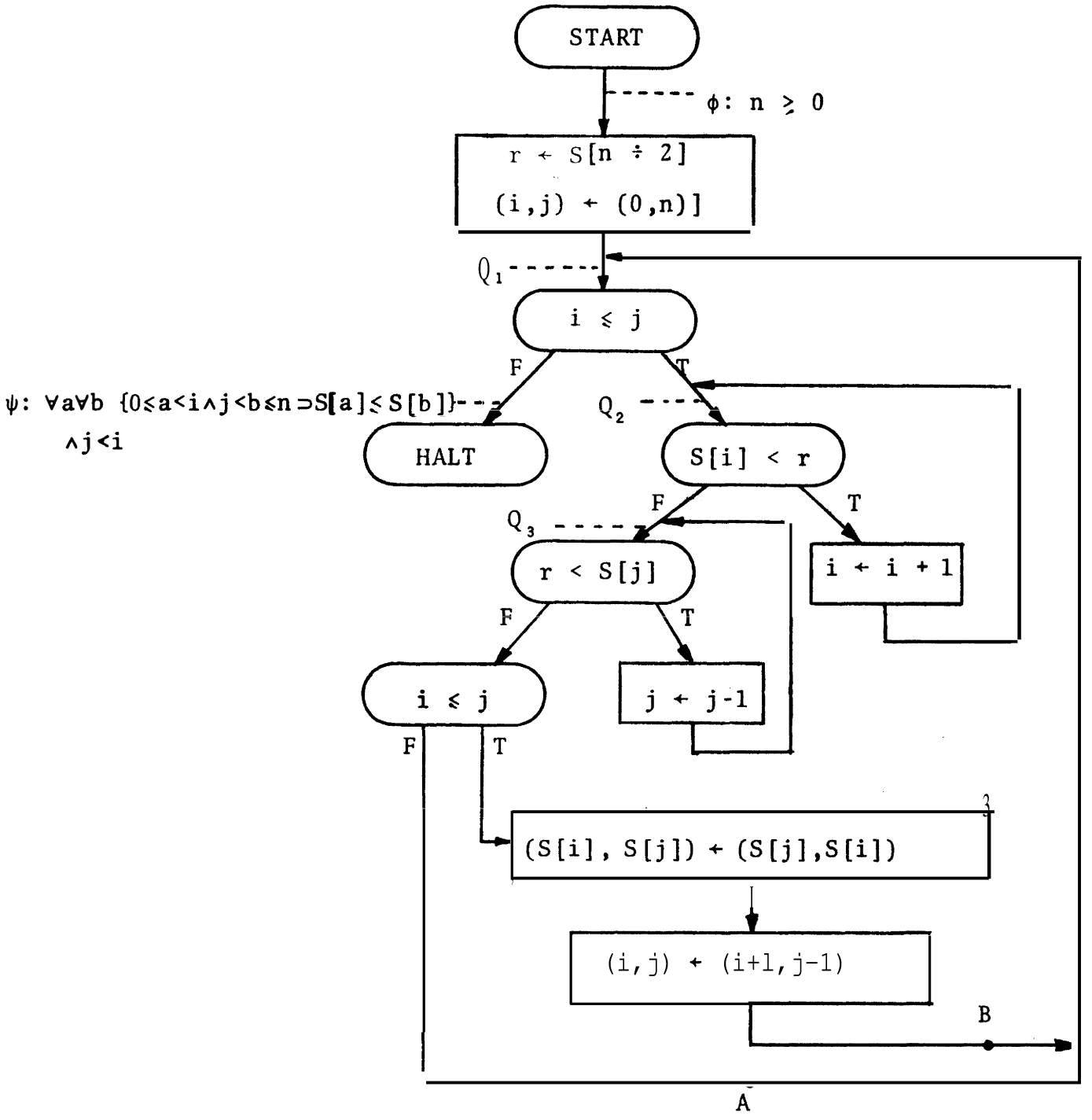


Figure 5. Partition Program

statements to the array in the loop. Moving $p_1 \wedge p_2 \wedge S[i] \geq r \wedge S[j] \leq r$ through the two possibilities for the test $i \leq j$, if we reach point A, the assertion is unchanged while at point B we have

$$p_1' : \forall k[i_0 \leq k < i\text{-}1 \supset S[k] < r] \wedge S[i-1] \leq r \quad \text{and}$$

$$p_2' : \forall \ell[j_0 \geq \ell > j+1 \supset S[\ell] > r] \wedge S[j+1] \geq r .$$

Rules X1 and X2 indicate that we require $p_1^* : \forall k[i_0 \leq k < i \supset \langle k\text{-array}\rangle]$ and $p_2^* : \forall \ell[j_0 \geq \ell > j \supset \langle R\text{-array}\rangle]$. Thus by R2 we seek weaker array assertions about the entire range of k and ℓ which will fulfill these forms. The weakest assertion made about an element in p_1 or p_1' is that $S[i-1] \leq r$. Thus we let p_1^* be $\forall k[i_0 \leq k < i \supset S[k] \leq r]$. Similarly p_2^* is $\forall \ell[j_0 \geq \ell > j \supset S[\ell] \geq r]$. Since i_0 is initially 0, while j_0 is initially n, we assume a Q_1 assertion of the form $\forall k[0 \leq k < i \supset S[k] \leq r] \wedge \forall \ell[n \geq \ell > j \supset S[\ell] \geq r]$. By rule En2, Q_2 and Q_3 will be given the assertion of Q_1, and verifying these assertions will show the program partially correct. We clearly could have used the transitivity rule here, but for this example, the amount of work required is about the same.

IV. Conclusion

Clearly, the rules and examples given in this paper are far from being a general system for finding inductive assertions. More and better rules are needed, particularly, for array assertions, which tend to be complex and unwieldy.

In addition, before the rules can be incorporated into a practical framework, we must order their application. That is, at each step we must provide more exact criteria for deciding which rule to apply and on which cutpoint of the program. The order in which the rules are presented in each subclass does implicitly provide a partial specification. Thus we presently would try to apply Ex1, and only if it failed try Ex2, etc. Moreover, we generally would try to gather information on simple variables using the rules of Section II before attempting to treat array assertions.

The more basic (and open) questions are (a) whether to attempt top-down or bottom-up techniques first for a given loop, and (b) which loop of a program should be treated first. Although we experimented with various orderings in the examples in this paper, we have tentatively formulated a more fixed approach. Our present inclination is to first use top-down rules from the (physical) beginning of the program. (Since in general there is more than one outer loop, usually only entrance rules are applicable.) Then we use bottom-up rules for the same loop,

to create a p true after exit from the first loop containing as much information as possible. We continue with the next outer loop in a similar manner. If, however, we are stymied and unable to find a loop assertion, we start with top-down rules from the end of the program, and try to work backwards towards the beginning.

A more sophisticated approach would require a weighted evaluation function capable of making a very cursory scan of the program. This function would identify loops which seemed 'promising', i.e. likely to yield valuable information rapidly, and apply selected rules first to these loops.

Since some of the rules could continue searching for a possibly non-existent form of assertion almost indefinitely (the transitivity rule, for example), such rules would have a "weak" version and a "strong" version. The "weak" version would be used in the initial attempt to find an assertion, and would "give-up" rapidly if it did not provide an almost immediate 'solution. Then other, possibly more appropriate, rules may be tried on the cutpoint. Only if all rules failed to add relevant information, would the "strong" version be applied. This division is parallel to the human attempt to first find what is "obviously" true in the loop, and only afterwards bring out the fine points.

The overall strategy we have adopted in this paper has been

to find assertions strong enough to prove the partial correctness in as few steps as possible. Thus, in general, we attempt to directly produce a near-exact description of the operation of a loop, without going through numerous intermediate stages where we are unable to show either validity or unsatisfiability. If our heuristic is wrong, this fact will be revealed relatively rapidly by generating an unsatisfiable verification condition. We then may try a weaker alternative claim. We feel that this is the approach which should be taken in order to construct a practical system which could be added to a program verifier.

We believe that the bottom-up approach may also be used to solve other problems. For example, in the partition program (Example 5), the inductive assertion was actually found without using the ψ given by the programmer. In one single step ψ may be generated from Q_1, and thus we have 'discovered' what the program does without the use of additional information. This feature of the bottom-up approach can probably be most useful for strengthening a too-weak assertion, i.e., revealing that the program does more than is claimed in ψ.

Another apparent application is for proving termination using well-founded sets. For termination, predicates Q_i and functions u_i are required, where u_i (a mapping to the well-founded set) has its domain bounded by Q_i and descends each time the loop is executed. Here again the bottom-up approach

is useful since no ψ is provided. We have already begun investigating bottom-up methods for generating both the Q_i's and the u_i's which will ensure termination.

The ultimate goal of automatic assertion generation is almost certainly unattainable ; thus the optimal system would involve man-machine interaction. Whenever it was unable to generate the proper assertion, the machine would supply detailed questions on problematic relations among variables and possible failure points (incorrect loops) of the program. Cl early, a partial specification of the assertions, provided by the programmer, could shorten this entire process.

REFERENCES

COOPER [1971]. D. C. Cooper, "Programs for Mechanical Program Verification", in Machine Intelligence 6, American Elsevier, pp. 43-59 (1971).

ELSPAS et al. [1972]. B. Elspas, M.W. Green, K.N. Levitt and R.J. Waldinger, "Research in Interactive Program-Proving Techniques", SRI, Menlo Park, Cal. (May 1972).

FLOYD [1967]. R. W. Floyd, "Assigning Meanings to Programs", in Proc. of a Symposium in Applied Mathematics, Vol. 19 (J. T. Schwartz - editor), AMS, pp. 19-32 (1967).

KING [1969]. J. King, "A Program Verifier", Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pa. (1969).

WEGBREIT [1973]. B. Wegbreit, "Heuristic Methods for Mechanically Deriving Inductive Assertions", Unpublished memo, Bolt, Beranek and Newman, Inc., Cambridge, Mass., (February 1973).