TOWARDS THE INTERACTIVE SYNTHESIS OF ASSERTIONS*

by

Mark S. Moriconi

October 1974 ATP-20

* The work reported herein is a revision and extension of an earlier report by the author [17].

TOWARDS THE INTERACTIVE SYNTHESIS OF ASSERTIONS*

Mark S. Moriconi

The University of Texas at Austin

ABSTRACT. A new technique is presented for the semiautomatic derivation of inductive assertions. The method combines information obtained from known assertions with knowledge extracted directly from the program body. A program is transformed into a structure that allows the automatic generation of inductive assertions to occur without having to generalize program statements to n iterations. Heuristics are discussed that in some cases suffice by themselves to generate inductive assertions; but more often, for non-trivial programs, are interfaced with the alternate program representation to immediately yield the desired assertion. When the mechanical derivation fails, inherent in the approach is an interactive capability which can be easily exploited by the human.

* This work was supported in part by NSF Grant DCR 74-12886.

1. INTRODUCTION

Much recent attention has been directed toward developing techniques for proving the correctness of programs. The assertion approach for verifying programs, which was formalized by Floyd [8] and Naur [18], has been the focus of many efforts in the field. However, previous work has indicated various practical limitations. It has been pointed out by King [13] and Elspas, et al. [6,7] among others that one of the single most important factors limiting these efforts at program verification is the difficulty of inventing Floyd assertions. The difficulty appears to be not so much a problem of syntax or the assertion language, but one of correctly understanding the program. Therefore, we initially concern ourselves with the development of a medium to facilitate greater program understanding and then use it for the semiautomatic generation of assertions.

Previous attempts at finding inductive assertions for programs exhibit essentially two approaches. The first is basically heuristic using the output assertion and test predicate in trying to form a particular inductive assertion. This approach is demonstrated by Wegbreit [21] and the top-down method of Katz and Manna [12]. The second attempts to be somewhat more formal and tries to extract information directly from the body of the program. The basis for this approach is a suggestion by Green [9] that the problem can be viewed as that of finding the solution to simultaneous sets of difference equations. The bottom-up method of Katz and Manna [12] is similar. Greif and Waldinger [10] describe a technique which uses the output assertion, the test predicate, and the body of the loop at the same time. They observe, however, that their method

breaks down in complex situations.

Our approach for finding a practical solution to the problem is significantly different from most others. The method we present has the ability to combine in a new fashion information obtained by the heuristic manipulation of known assertions and test predicates with the knowledge we extract from the actual program body. To do this we develop an alternate representation of a program which eliminates all local program variables and non-essential sequencing information resulting in a purely functional representation of a program. Extracting invariants from this alternate representation is often quite easy and is done without generalizing the program statements to n iterations as done in several other techniques. As a result the simplifications, substitutions, and so on that we require seem to be much less complicated than those for most other methods. Another novel feature is that when the mechanical derivation of an inductive assertion fails, the human is provided with a representation of the computation performed within the loop under consideration that in many cases has eliminated the possible obscurity existing in the original program. The human is thus allowed to utilize his ingenuity to great benefit in forming totally unspecified assertions or completing partially specified ones.

The representation of a program we employ, known as a case description, was informally introduced by Pratt [19]. Sections 2-5 give a fairly lengthy formalization of his notion of a case description, extending it to the domain of asserted programs, and ultimately establishing the functional equivalence of a program and its associated case description. This development is essential in order for us to view both as simply different representations

of the same (partial) function. The method we use for synthesizing inductive assertions is predicated upon this assumption. We point out, however, that a cursory reading of the formal analysis in Sections 2-5 is sufficient for proceeding to Sections 6 and 7 which deal directly with assertion synthesis.

Having established the necessary formal results, Section 6 describes the general heuristics of the method. Section 7 presents examples illustrating the use of the heuristics and details the specifics of the technique. A comparison to some other possible approaches is given.

2. PRELIMINARIES

In this section we begin the formal development of this paper by presenting a simple formulation of the first-order predicate calculus. We consider in detail only the concepts which might possibly be confused with other standard usages and also specify notation to be used throughout. For further discussion see Church [4], Mendelson [16], and Shoenfield [20].

The basic alphabet consists of commas; parentheses; the logical symbols $\forall$, $\wedge$, $\sim$; individual variables $x_1, x_2, \ldots, x_n, \ldots$; individual constants $c_1, c_2, \ldots, c_n, \ldots$; function letters $f_1^1, f_1^2, \ldots, f_k^\ell, \ldots$; and predicate letters $p_1^1, p_1^2, \ldots, p_k^\ell, \ldots$. The superscript of a function or predicate letter represents the number of arguments, whereas the subscript is simply an index number to distinguish different function or predicate letters with the same number of arguments.

Terms, atomic formulas, and wffs are formed in the usual manner.

An interpretation[1] $\mathcal{I}$ consists of a non-empty set D, called the domain of $\mathcal{I}$, and an assignment to each predicate letter p_i^n an n-ary relation R_i^n in D, to each function letter f_i^n an n-ary total function F_i^n from D^n into D, and to each individual constant c_i some fixed element d_{c_i} of D.

Given an interpretation $\mathcal{I}$ with domain D, let $\sum$ be the set of denumerable sequences of elements of D. Let $\bar{\sigma} = (a_1, a_2, \ldots)$ be a sequence in $\sum$. We define a function ρ of one argument having terms as arguments and values in D.[2] An <u>assignment</u> is determined by the function ρ depending on $\bar{\sigma}$, denoted as $\rho_{\bar{\sigma}}$, in the following manner:

1. $\rho_{\bar{\sigma}}(x_i) = a_i$.
2. $\rho_{\bar{\sigma}}(c_i) = d_{c_i}$.
3. $\rho_{\bar{\sigma}}(f_i^n(t_1, \ldots, t_n)) = F_i^n(\rho_{\bar{\sigma}}(t_1), \ldots, \rho_{\bar{\sigma}}(t_n))$.

The intuitive notions of satisfiability, truth, and validity can now be defined inductively via the function $\rho_{\bar{\sigma}}$.

---

[1] In Shoenfield [20] what we refer to as an interpretation is called a <u>structure</u>. We use this terminology to be consistent with subsequent definitions.

[2] The ρ function is used extensively in Section 3.3 and Section 5.

3. ABSTRACT PROGRAMS AND RELATED CONCEPTS

In this section we present a typical abstract model of a computer program (called a flowchart or abstract schemata) and then consider a particular (unique) computation (execution sequence) resulting from a specific interpretation and assignment. Models similar to ours have been studied by many authors, e.g., Kaplan [11], Luckham, Park, and Paterson [14], and Manna [15].

3.1. Abstract Programs. An abstract program AP consists of:

1. (a) A finite set of individual input variables $\bar{x} = \{x_1, \ldots, x_n\}$ with $n \geq 0$, and

(b) a finite set of individual program variables $\bar{y} = \{y_1, \ldots, y_n\}$ with $n \geq 1$.

2. A finite, directed, labeled graph which we define by the triple $\langle \mathcal{N}, \Delta, \mathcal{L} \rangle$ with a finite set of nodes $\mathcal{N} = \mathcal{N}' \cup \mathcal{H}$ such that there is

(a) a unique entry node named and labeled START with START $\in \mathcal{N}'$, and

(b) a set $\mathcal{H}$ composed of at least one exit node named and labeled HALT with $\mathcal{H} \cap \mathcal{N}' = \emptyset$.

3. $\Delta: \mathcal{N}' \to \mathcal{N} \cup \bigcup_{i=1}^{2} \mathcal{N}$ defines the flow of control for AP.

4. $\mathcal{A}$ and $\mathcal{B}$ are sets composed of assignment and branch statements, respectively, which are defined as follows:

(a) An assignment statement is an expression of the form $\alpha \leftarrow \tau(\bar{x},\bar{y})$ where $\tau(\bar{x},\bar{y})$ is a term with no variables other than x_i and y_i and α is a program variable. In addition START is considered to be in $\mathcal{A}$.

(b) A branch statement is a quantifier-free wff $\beta(\bar{x},\bar{y})$ with no variables other than x_i and y_i.

5. $\mathcal{L}: \mathcal{N}' \to \mathcal{A} \cup \mathcal{B}$ gives a labeling for the nodes of AP from sets $\mathcal{A}$ and $\mathcal{B}$. This labeling is restricted such that for all $n \in \mathcal{N}'$,

$$\mathcal{L}(n) \in \mathcal{A} \iff \Delta(n) \in \mathcal{N} \tag{1}$$

and

$$\mathcal{L}(n) \in \mathcal{B} \iff \Delta(n) \in \mathcal{N}^2 . \tag{2}$$

To AP we add an input predicate (or input assertion), denoted $\varphi(\bar{x})$, which is a wff with no free individual variables other than $\bar{x}$. $\varphi(\bar{x})$ usually specifies the domains of the input variables and any constraints on the joint occurrence of values of input variables. We denote an abstract program with an input assertion as (AP,φ).

We require that (AP,φ) begin (first node after START) with a sequence of assignment statements $y_1 \leftarrow \tau_1(\bar{x}),\ldots,y_\ell \leftarrow \tau_\ell(\bar{x})$, where $\tau_j(\bar{x})$ with $1 \le j \le \ell$ is a term whose only variables are x_i. We make this requirement because it is an easy way to insure that each program variable used on the right side of an assignment statement has been previously initialized; and if it is not used within the program but designated as an output variable, a previous assignment to it must occur for us to define the value

of an execution sequence in Section 3.3. In order to make this initialization we require that the abstract program contain at least one input variable or constant.

3.2. Programs. Let $D_{\bar{x}}$ and $D_{\bar{y}}$ be non-empty domains for $\bar{x}$ and $\bar{y}$, respectively, such that $D_{\bar{x}} \subseteq D_{\bar{y}}$. Given an abstract program with the characteristics just discussed, an interpretation $\mathcal{I}$ for (AP,φ) can be defined over the appropriate domains as shown in Section 2.

The abstract program (AP,φ) together with an interpretation $\mathcal{I}$ forms what is called a program and is denoted by $(P,\mathcal{I},\varphi)$.

3.3. Interpreted Programs. Let $(P,\mathcal{I},\varphi)$ be a program and $\bar{\xi} \in D_{\bar{x}}$ be an input assignment for $\bar{x}$. This defines the interpreted program $(P,\mathcal{I},\varphi,\bar{\xi})$. An interpreted program can be executed defining what is called an execution sequence that may be finite or infinite. Before defining this concept we make the following notational conveniences to be used throughout.

Notation. Let $\langle d_i \rangle$ denote the vector[3] of constants $\langle d_1, \ldots, d_n \rangle$ and $\langle t_i^k \rangle = \langle t_1^k, \ldots, t_n^k \rangle$ such that t_i^k denotes the i-th term of the k-th sequence. We also refer to $\tau(\bar{x},\bar{y})$ and $\beta(\bar{x},\bar{y})$ as simply τ and β, respectively.

[3] $\langle d_i \rangle$ can be viewed as simply an element $(a_{\sigma_1}, \ldots, a_{\sigma_n})$ of Σ from Section 2.

Definition 1. We define $\mathscr{S}$ inductively by

$$\mathscr{S}(1) = (\langle t_i^1 \rangle, \mathscr{C}_1, n_1)$$

when

$$\bar{\xi} \text{ satisfies } \varphi(\bar{x}) ,$$

where

$$n_1 = \Delta(\text{START}) , \quad \mathscr{C}_1 = \{\varphi(\bar{x})\} ,$$

and

$$t_i^1 = x_\nu \text{ or } t_i^1 = c_\nu , \quad \nu \text{ fixed}$$

if there are or are not input variables, respectively.

The initialization of t_i^1 is necessary in order to define the next step in the sequence, viz. $\mathscr{S}(k+1)$.

Given $$\mathscr{S}(k) = (\langle t_i^k \rangle , \mathscr{C}_k , n_k) .$$

If $\mathscr{L}(n_k) \neq \text{HALT}$, we define

$$\mathscr{S}(k+1) = (\langle t_i^{k+1} \rangle , \mathscr{C}_{k+1} , n_{k+1}) ,$$

depending upon two cases, i.e., whether

$$\Delta(n_k) \in \mathcal{N} \quad \text{or} \quad \Delta(n_k) \in \mathcal{N}^2 \quad .$$

Case (i): If $\Delta(n_k) \in \mathcal{N}$ then let $\mathcal{L}(n_k)$ be $y_p \leftarrow \tau$. Thus

$$n_{k+1} = \Delta(n_k) \quad , \qquad \text{by (1)}$$

$$\mathcal{C}_{k+1} = \mathcal{C}_k \quad ,$$

and

$$t_i^{k+1} = \begin{cases} t_i^k \ , & i \neq p \\ \tau\{t_1^k/y_1, \ldots, t_n^k/y_n\}^4 \ , & i = p \ . \end{cases}$$

Case (ii): If $\Delta(n_k) \in \mathcal{N}^2$ then let $\mathcal{L}(n_k)$ be β. Since (2) gives $\Delta(n_k) = \langle y, z \rangle$, we have

$$n_{k+1} = \begin{cases} y \ , & \text{when } \bar{\xi} \text{ satisfies } \beta\{t_1^k/y_1, \ldots, t_n^k/y_n\}^5 \\ z & \text{otherwise} \ , \end{cases}$$

$$\mathcal{C}_{k+1} = \begin{cases} \mathcal{C}_k \cup \{\beta\{t_1^k/y_1, \ldots, t_n^k/y_n\}\} \ , & \text{if } n_{k+1} = y \\ \mathcal{C}_k \cup \{\sim\beta\{t_1^k/y_1, \ldots, t_n^k/y_n\}\} & \text{otherwise} \ , \end{cases}$$

and

$$t_i^{k+1} = t_i^k \quad .$$

---

[4] $\tau\{t_1^k/y_1, \ldots, t_n^k/y_n\}$ means that each y_i in τ is to be replaced by the corresponding t_i^k of $\langle t_i^k \rangle$. Similarly for $\beta\{t_1^k/y_1, \ldots, t_n^k/y_n\}$.

[5] It is true that $\bar{\xi}$ either satisfies or does not satisfy $\beta\{t_1^k/y_1, \ldots, t_n^k/y_n\}$.

The function $\mathscr{E}$ defines an execution sequence

$$\langle P, \mathscr{I}, \varphi, \bar{\xi} \rangle$$

for $(P, \mathscr{I}, \varphi, \bar{\xi})$ with the <u>value of the computation</u>, denoted

$$\mathrm{Val}(\langle P, \mathscr{I}, \varphi, \bar{\xi} \rangle) \quad ,$$

equal to

$$\langle \rho_{\bar{\xi}}(t_i^{\Omega}) \rangle \quad ,$$

where

$$\Omega = \max(\mathrm{domain}(\mathscr{E})) \quad ,$$

when $\mathscr{E}$ is finite (i.e., the program terminates); otherwise $\mathrm{Val}(\langle P, \mathscr{I}, \varphi, \bar{\xi} \rangle)$ is undefined.

We remark that $\rho_{\bar{\xi}}$ applied to t_i^{Ω} yields a specific vector of constants, i.e.,

$$\langle \rho_{\bar{\xi}}(t_i^{\Omega}) \rangle = \langle d_i \rangle \quad .$$

4. ABSTRACT CASE DESCRIPTIONS AND RELATED CONCEPTS

In the next two sections we extend and formalize some ideas discussed by Pratt in [19] in the direction of developing a formal yet practical basis for generating assertions for programs.

4.1. Descriptors. For the abstract program (AP,φ) let δ_j represent a possible sequence of nodes, i.e., a path, $n_{j_1}, n_{j_2},\ldots,n_{j_\eta}$ from (AP,φ) having the following properties:

1. $\delta_j(1) = \Delta(\text{START})$.
2. Given $\delta_j(k)$:
   (a) If $\mathcal{L}(\delta_j(k)) \in \mathcal{A}$ then $\delta_j(k+1) = \Delta(\delta_j(k))$.
   (b) If $\mathcal{L}(\delta_j(k)) \in \mathcal{B}$ and $\Delta(\delta_j(k)) = \langle y, z\rangle$ then $\delta_j(k+1) = y$ or $\delta_j(k+1) = z$.
3. There is a maximum η such that $\delta_j(\eta) = \text{HALT}$

This defines a path $\delta_j \in \{\delta_1,\delta_2,\ldots,\delta_n,\ldots\}$, the set of all possible sequences of nodes (paths) through (AP,φ).

For each δ_j we record in sequence $\varphi(\bar{x})$ followed by

$$\mathcal{L}(n_{j_\alpha}), \qquad \alpha = 1,\ldots,\eta .$$

If δ_j contains an n_{j_α} such that

$$\mathcal{L}(n_{j_\alpha}) \in \mathcal{B}$$

with $\Delta(n_{j_\alpha}) = \langle y, z \rangle$, we record

$$\mathcal{L}(n_{j_\alpha}) \quad \text{or} \quad \sim\mathcal{L}(n_{j_\alpha})$$

depending on whether

$$n_{j_{\alpha+1}} = y \quad \text{or} \quad n_{j_{\alpha+1}} = z \ ,$$

respectively. We call this an <u>abstract</u> or <u>unreduced descriptor</u> for (AP, φ).

We now make a definition which is very similar to that of an execution sequence in Section 3.3, but we do not assume an input assignment and thus follow an arbitrary path through the program. This algorithm turns out to be the basis for our assertion generation procedure shown in Sections 6 and 7.

<u>Definition 2</u>. We define $\mathfrak{D}(\delta_j)$ inductively as follows:

$$\mathfrak{D}(\delta_j)(1) = (\langle t_i^1 \rangle \ , \ \mathscr{E}_1^j \ , \ \delta_j(1)) \ ,$$

where

$$\mathscr{E}_1^j = \{\varphi(\overline{x})\} \ ,$$

and if there are or are not input variables we have

$$t_i^1 = x_\nu \quad \text{or} \quad t_i^1 = c_\nu \ , \qquad \nu \text{ fixed}$$

respectively.

Given

$$\mathcal{D}(\delta_j)(k) = (\langle t_i^k \rangle , \ \mathscr{C}_k^j , \ \delta_j(k)) \ .$$

For $\mathcal{L}(\delta_j(k)) \neq \text{HALT}$ we have

$$\mathcal{D}(\delta_j)(k+1) = (\langle t_i^{k+1} \rangle , \ \mathscr{C}_{k+1}^j , \ \delta_j(k+1)) \ ,$$

where either

(i) $\mathcal{L}(\delta_j(k)) \equiv y_p \leftarrow \tau$, in which case we have

$$\mathscr{C}_{k+1}^j = \mathscr{C}_k^j \ ,$$

and

$$t_i^{k+1} = \begin{cases} t_i^k & , \quad i \neq p \\ \tau\{t_1^k/y_1, \ldots, t_n^k/y_n\} & , \quad i = p \end{cases}$$

or

(ii) $\mathcal{L}(\delta_i(k)) \equiv \beta$, then for $\Delta(\delta_j(k)) = \langle y, z \rangle$ set

$$\mathscr{C}_{k+1}^j = \begin{cases} \mathscr{C}_k^j \cup \{\beta\{t_1^k/y_1, \ldots, t_n^k/y_n\}\} & , \text{ if } \ \delta_j(k+1) = y \\ \text{else} \ \ \mathscr{C}_k^j \cup \{\sim\beta\{t_1^k/y_1, \ldots, t_n^k/y_n\}\} & , \end{cases}$$

and

$$t_i^{k+1} = t_i^k \ .$$

For each δ_j, $\mathcal{D}(\delta_j)$ produces what is called a reduced descriptor or simply descriptor, denoted as

$$(D,\varphi)_{\delta_j} \ ,$$

with control set and kernel set $\mathcal{K}$ equal to

$$\mathcal{C}_{\omega}^{j} \quad \text{and} \quad \langle t_i^{\omega} \rangle \ ,$$

respectively, where

$$\omega = \max(\text{domain } \mathcal{D}(\delta_j)) \ . \tag{3}$$

We observe that the main difference between the original path δ_j and its corresponding descriptor is that the descriptor contains no local variables or extraneous sequencing information. A descriptor specifies the sequence of operations only where necessary, i.e., in function composition and variables in argument expressions. This means that we now have a representation for a program path which will in many cases exhibit quite clearly the information necessary to understand what a program (or part of a program) computes if that particular path (or subpath) is followed.

4.2. Case Descriptions. We define an abstract case description, denoted (AC,φ), to be

$$\bigcup_{j \geq 1} (D,\varphi)_{\delta_j} ,$$

i.e., we generate a descriptor for each (finite) path δ_j, in (AP,φ). Of course (AC,φ) is countably infinite unless (AP,φ) contains no loops.

(AC,φ) together with an interpretation $\mathcal{I}$ forms the <u>case description</u> $(C,\mathcal{I},\varphi)$.

An <u>interpreted case description</u> is a case description with an assignment $\bar{\xi} \in D_{\underline{x}}$ and is written as $(C,\mathcal{I},\varphi,\bar{\xi})$.

The control sets of the descriptors in the interpreted case description are composed of ground instances of their predicates which can now be evaluated in the usual manner. To find the unique descriptor corresponding to a particular execution sequence we simply search the list of descriptors for the descriptor all of whose control set predicates are satisfied. This descriptor's kernel set specifies the values computed for that execution sequence. However, from a well-known undecidability result we know that it is impossible in general to determine whether an arbitrary program terminates for all inputs. Therefore, we may never find a descriptor in the interpreted case description having all predicates in its control set satisfied, thus continuing to test indefinitely. We formalize this intuitive notion of functional equivalence in the Section 5.

4.3. <u>Examples</u>. In the domain of program correctness an <u>output predicate</u> (or <u>output assertion</u>) is attached to the program under consideration. We designate this predicate as Ψ. It usually specifies the desired relation between

the input and output variables. To distinguish for the reader which variables are assigned values by the program we add a set of individual output variables $\bar{z} = \{z_1, \ldots, z_n\}$ when convenient.

The following examples illustrate the derivation of case descriptions forming the descriptors as specified in Definition 2. In addition we show a possible use of case descriptions for either forming a particular ψ or "checking" one that was given a priori. If an inaccurate ψ is provided serious difficulties can arise in an attempt to prove parital correctness (for discussion see [7]). In addition if an inaccurate ψ is used in the partial or total derivation of an inductive assertion, as in Wegbreit [21] and Katz and Manna [12], it is highly unlikely that the loop invariants will ever be found.

In Figure 1 we see a simple factorial program with range 0 to 50 as specified by the input assertion φ. Table 1 contains descriptors of its case description for some of the shorter paths. Each descriptor is shown in its unreduced, reduced, and interpreted (and simplified) form. Note that the number of unreduced and reduced descriptors is countably infinite due to the loop in the abstract program. However, by viewing the interpreted schemata we see that only a finite number of descriptors are possible. Paths which cannot be followed irrespective of the initial assignment yield a contradiction in the control set (denoted by $\square$) as shown by interpreted descriptor 52. The importance of this will become magnified in more complex programs.

To simplify the interpreted descriptor we can employ a limited theory

of types and specialized routines for algebraic simplification and solving linear inequalities. Since in the control set, with possible the exception of $\varphi(\bar{x})$, we are dealing only with skolem constants we expect very efficient analysis. Some of the specific techniques we will use are discussed by Bledsoe, et al. in [1,2].

Looking again at Table 1, by viewing only a few descriptors we can easily see that ψ is $\{Z = N!\}$.

At this point we are merely trying to show how to form (partial) case descriptions. However, we must mention that the technique we develop in Sections 6 and 7 for generating inductive assertions can, with minor modifications, be used in many cases to derive ψ mechanically as well.

Notation. Let $\delta(j_1,\ldots,j_\eta)$ denote a path (or subpath) through the flowchart over arcs $j_1,\ldots,j_\eta$. We henceforth omit the unreduced descriptor adding instead the statement $\delta_j = \delta(j_1,\ldots,j_\eta)$ to the descriptor.

Figure 2 (taken from [7]) computes the fractional quotient P/Q to within tolerance E.[6] In Table 2 we see the (partial) case description for Figure 2. It is easy to see from the interpreted descriptors that ψ is $\{P/Q - E < Z \leq P/Q\}$.

The flowchart of Figure 3 (taken from [13]) multiplies two numbers, accepts signed inputs, and all additive operations are restricted to incrementing and decrementing by one. From Table 3 it is easily seen that ψ is $\{Y = DA * B\}$. As in Table 1 we again uncovered a path which cannot

[6] In this example the domain is the reals.

be followed, viz. δ_4. However, we note that in this instance it was not entirely obvious that δ_4 could not be executed. The knowledge that a particular path cannot be followed, even if $\varphi(\overline{x})$ is satisfied, will often prove to be quite valuable information in many areas of program analysis, especially in program verification. We also point out that in this example the variables A,Y, and XB have been eliminated.

We have hopefully demonstrated the conceptual advantage in separating a program into its control and kernel sets. Let us now continue our formal development.

5. FUNCTIONAL EQUIVALENCE

We begin by making the following definitions.

Definition 3. $\mathcal{F}$ and $\overline{\xi} \in D_{\overline{x}}$ satisfies (AC,φ) means that there is a descriptor

$$(D,\varphi)_{\delta_j} \in (AC,\varphi)$$

such that $\mathcal{F}$ and $\overline{\xi}$ satisfy

$$\mathcal{C}^j_\omega \ ,$$

where ω is defined by (3).

Definition 4. (AC,φ) is said to be consistent if and only if for every interpretation $\mathcal{I}$ and input assignment $\bar{\xi} \in D_{\underline{x}}$ there is at most one descriptor

$$(D,\varphi)_{\delta_j} \in (AC,\varphi)$$

such that all of its control set predicates are satisfied.

Definition 5. If (AC,φ) is consistent then the value of $(C,\mathcal{I},\varphi,\bar{\xi})$, written

$$Val((C,\mathcal{I},\varphi,\bar{\xi})) \quad ,$$

is determined as follows:

1. Let $(D,\varphi)_{\delta_j}$ be the unique descriptor in (AC,φ) having all of its control set predicates satisfied by $\mathcal{I}$ and $\bar{\xi}$. In this case

$$Val((C,\mathcal{I},\varphi,\bar{\xi})) = \mathcal{K}((D,\varphi)_{\delta_j}) \quad ,$$

the kernel set of $(D,\varphi)_{\delta_j}$ under $\mathcal{I}$ and $\bar{\xi}$.

2. If no such $(D,\varphi)_{\delta_j}$ exists then $Val((C,\mathcal{I},\varphi,\bar{\xi}))$ is undefined.

It is crucial in our development to know that if there is a descriptor in the abstract case description whose control set is satisfied by $\mathcal{I}$ and $\bar{\xi}$ then it is unique. We are now in a position to establish this by proving

Lemma 1. (Consistency Lemma) If (AC,φ) is an abstract case description then (AC,φ) is consistent.

Proof. Suppose $\mathcal{F}$ and $\bar{\xi}$ satisfy (AC,φ) such that there are two distinct descriptors

$$(D,\varphi)_{\delta_i} \quad \text{and} \quad (D,\varphi)_{\delta_j}$$

with control sets $\mathcal{C}^i_\omega$ and $\mathcal{C}^j_\omega$, respectively, satisfied by $\mathcal{F}$ and $\bar{\xi}$. Let

$$\mu = \min\{\gamma \mid \delta_i(\gamma) \neq \delta_j(\gamma)\} \quad . \tag{4}$$

Therefore

$$\delta_i(\mu-1) = \delta_j(\mu-1) \quad . \tag{5}$$

Now consider the case when

$$\delta_i(\mu-1) \in \mathcal{Q} \quad . \tag{6}$$

From the definition of path and (5) we have

$$\delta_i(\mu) = \Delta(\delta_i(\mu-1)) = \Delta(\delta_j(\mu-1)) = \delta_j(\mu)$$

which by (4) cannot be the case. Thus (6) does not hold and $\delta_i(\mu\text{-}1)$ must be a branch node.

Since

$$\delta_i(\mu\text{-}1) \in \mathcal{B}$$

as is the right side of (5) and

$$\delta_i(\mu) \neq \delta_j(\mu) \quad ,$$

without loss of generality we may assume that

$$\delta_i(\mu) = y \quad \text{and} \quad \delta_j(\mu) = z$$

for

$$\Delta(\delta_i(\mu\text{-}1)) = \Delta(\delta_j(\mu\text{-}1)) = \langle y, z \rangle \quad .$$

Thus

$$\mathcal{L}(\delta_i(\mu\text{-}1))\{t_1^{\mu-1}/y_1, \ldots, t_n^{\mu-1}/y_n\} \in \mathcal{C}_\mu^i \tag{7}$$

and

$$\sim\mathcal{L}(\delta_i(\mu\text{-}1))\{t_1^{\mu-1}/y_1, \ldots, t_n^{\mu-1}/y_n\} \in \mathcal{C}_\mu^j \quad . \tag{8}$$

But now

$$\mathscr{C}^{i}_{\mu} \subseteq \mathscr{C}^{i}_{\omega} \quad \text{and} \quad \mathscr{C}^{j}_{\mu} \subseteq \mathscr{C}^{j}_{\omega} \quad . \tag{9}$$

Moreover since

$$\mathscr{C}^{i}_{\omega} \quad \text{and} \quad \mathscr{C}^{j}_{\omega}$$

are satisfied by $\mathscr{F}$ and $\overline{\xi}$ by assumption, it now follows from (9) that $\mathscr{F}$ and $\overline{\xi}$ satisfy

$$\mathscr{C}^{i}_{\mu} \quad \text{and} \quad \mathscr{C}^{j}_{\mu}$$

which contradicts (7) and (8). Q.E.D.

Lemma 2. $\mathscr{F}$ and $\overline{\xi}$ satisfy (AC,φ) if and only if $\langle P,\mathscr{F},\varphi,\overline{\xi}\rangle$ is defined.

We omit the proof of this result as the necessary part follows directly from Lemma 1, Definition 2, and a simple induction proof that for some path δ_j, $\mathscr{C}_k = \mathscr{C}^{j}_{k}$ for $k = 1,\ldots,\omega = \Omega$. The sufficient part also can be obtained in a straightforward manner from Definition 1. (For details see Moriconi [17].)

We now have

Corollary 1. $\mathscr{F}$ and $\overline{\xi}$ do not satisfy (AC,φ) if and only if

$\langle ?, \mathcal{I}, \varphi, \bar{\xi} \rangle$ is undefined.

We are now in a position to prove the main results.

Theorem 1. If $\mathcal{I}$ and $\bar{\xi}$ satisfy (AC,φ) then

$$Val((C,\mathcal{I},\varphi,\bar{\xi})) = Val(\langle P,\mathcal{I},\varphi,\bar{\xi}\rangle) \quad .$$

Proof. By hypothesis and the Consistency Lemma we know that there is a unique descriptor

$$(D,\varphi)_{\delta_j} \in (AC,\varphi)$$

such that its control set $\mathcal{C}^j_\omega$ is satisfied by $\mathcal{I}$ and $\bar{\xi}$. From Lemma 2 $\langle P,\mathcal{I},\varphi,\bar{\xi}\rangle$ is defined and furthermore must follow δ_j. Since it can easily be shown that

$$\langle t^\omega_i \rangle = \langle t^\Omega_i \rangle \quad ,$$

we must have

$$\langle \rho_{\underline{\xi}}(t^\omega_i) \rangle = \langle \rho_{\underline{\xi}}(t^\Omega_i) \rangle \quad .$$

Therefore,

$$Val((C,\mathcal{I},\varphi,\bar{\xi})) = Val(\langle P,\mathcal{I},\varphi,\bar{\xi}\rangle) \quad . \qquad \text{Q.E.D.}$$

By a similar argument we get

Theorem 2. If $\langle P,\mathcal{I},\varphi,\bar{\xi}\rangle$ is defined then

$$Val((C,\mathcal{I},\varphi,\bar{\xi})) = Val(\langle P,\mathcal{I},\varphi,\bar{\xi}\rangle) \quad .$$

Theorems 1 and 2 simply state the intuitive fact that whenever the hypotheses of either theorem is satisfied, i.e., whenever either there is a descriptor in (AC,φ) with its control set $\mathcal{C}^{j}_{\omega}$ satisfied for $\mathcal{I}$ and $\bar{\xi}$ or the execution sequence $\langle P,\mathcal{I},\varphi,\bar{\xi}\rangle$ is defined, we have

$$\langle \rho_{\bar{\xi}}(t_i^{\omega})\rangle = Val((C,\mathcal{I},\varphi,\bar{\xi}))$$
$$= Val(\langle P,\mathcal{I},\varphi,\bar{\xi}\rangle) = \langle \rho_{\bar{\xi}}(t_i^{\Omega})\rangle = \langle d_i\rangle$$

just as we would expect.

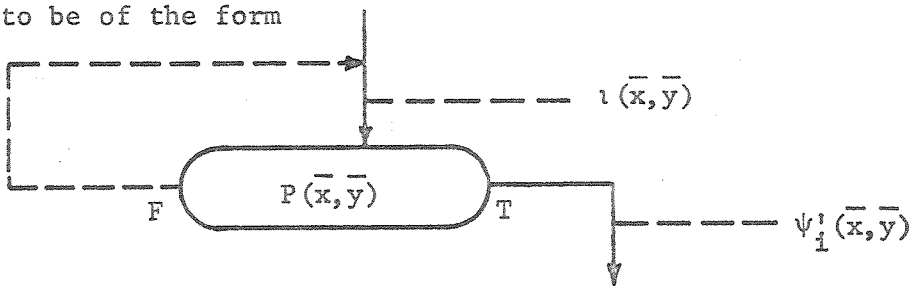
Combining Corollary 1 with Theorems 1 and 2 establishes the functional equivalence of an interpreted program and its corresponding interpreted case description.

6. GENERAL HEURISTICS

We now describe some of the general heuristics we employ which in many cases suffice by themselves to generate inductive assertions; but more often, they are used in concert with a (partial) case description.

The specifics of this interrelationship are shown in the next section. The portions of the rules we describe that manipulate the output assertion and test predicate are similar in content to those of Wegbreit [21] and Katz and Manna [12]. A significant difference, however, is that we develop only a few heuristic rules, of which normally only one applies, such that when they are employed (usually in conjunction with a (partial) case description) a consistent assertion is usually formed on the first try. Hence, we generally avoid fruitless attempts at generating and trying to prove verification conditions for inconsistent predicates. Furthermore, having shown in Section 4.3 that a (partial) case description is a simple and easy to understand representation of a program, when the heuristics fail the human can in many instances recognize "patterns", i.e., invariants, in successive descriptors with relative ease.

For simplicity in the statement of the heuristic rules, we view an exit from a loop to be of the form



where the cutpoint for the loop immediately preceeds the exit test and is marked by $\iota(\bar{x},\bar{y})$ representing the inductive assertion for the loop, $P(\bar{x},\bar{y})$ is the exit test, and $\psi_i'(\bar{x},\bar{y})$ is some conjunct of a predicate $\psi'(\bar{x},\bar{y})$ known to be true when $P(\bar{x},\bar{y})$ is satisfied. We observe that $\psi'(\bar{x},\bar{y})$ is usually different from $\psi(\bar{x},\bar{y})$. However, in the simplest case $\psi'(\bar{x},\bar{y})$ is merely $\psi(\bar{x},\bar{y})$, i.e., there are no statements between

$\psi(\bar{x},\bar{y})$ and the branch statement under consideration.

We now consider the following heuristics which are applied in the specified order to formulas in their "natural" form.

1. Convert $\psi'(\bar{x},\bar{y})$ to conjunctive form, i.e., $\psi'(\bar{x},\bar{y}) = \psi'_1(x,y) \wedge \ldots \wedge \psi'_n(\bar{x},\bar{y})$, and consider each $\psi'_i(\bar{x},\bar{y})$ separately with one goal being to form for each $\psi'_i(\bar{x},\bar{y})$ an appropriate $\iota_k(\bar{x},\bar{y})$ that satisfies $\iota_k(\bar{x},\bar{y}) \Rightarrow (P(\bar{x},\bar{y}) \Rightarrow \psi'_i(\bar{x},\bar{y}))$ where $\iota(\bar{x},\bar{y}) = \iota_1(\bar{x},\bar{y}) \wedge \ldots \wedge \iota_\ell(\bar{x},\bar{y})$.

2. (Transitivity) If $\psi'_i(\bar{x},\bar{y})$ is of the form $t_1 R_1 t_2$ and $P(\bar{x},\bar{y})$ is of the form $t_3 R_2 t_4$, where t_1, t_2, t_3, t_4 are terms and R_1, R_2 are inequality relations, then to find the appropriate $\iota_k(\bar{x},\bar{y})$ we normally employ the usual transitive closure properties with the domain being the integers unless otherwise specified. If $\psi'_i(\bar{x},\bar{y})$ contains quantification we apply a transitivity axiom directly to $P(\bar{x},\bar{y}) \Rightarrow \psi'_i(\bar{x},\bar{y})$ and if necessary make the appropriate subscript changes when arrays occur within the quantified expression. If no transitive rule applies to R_1 and R_2 we simply choose $\iota_k(\bar{x},\bar{y}) = \psi'_i(\bar{x},\bar{y})$.

Rules 3,4, and 5 will normally be considered as yielding trial inductive assertions for which we generate a (partial) case description for the loop under consideration to verify the correctness of or to generate an entirely new $\iota_k(\bar{x},\bar{y})$ before attempting to prove any verification conditions.

3. (Equality) The reason for using this rule almost exclusively in conjunction with a (partial) case description is to avoid making erroneous

assertions which in the case of an equality branch can happen quite easily. We view a few specific cases.

(a) If $P(\bar{x},\bar{y})$ is an equality branch with $P(\bar{x},\bar{y}) \neq \psi_i'(\bar{x},\bar{y})$ and $P(\bar{x},\bar{y})$ is of the form $t_i = 0$, compute a (partial) case description for the loop. We then use this (partial) case description to determine where the t_i occurs in $\iota_k(\bar{x},\bar{y})$. For example, suppose we have $A=0 \Rightarrow X=Y*Z$. There are numerous plausible $\iota_k(\bar{x},\bar{y})$ such as $X+A = Y*Z$, $X*C\uparrow A = Y*Z$, $X*(C*A)=Y*Z$, $X*C\uparrow A=Y*Z*C\uparrow A$ and so on where C is a constant. We avoid this proliferation of possible inductive assertions by generating the (partial) case description from which in many cases we can generate a consistent inductive assertion without human intervention.

(b) If $P(\bar{x},\bar{y})$ is an equality branch with $P(\bar{x},\bar{y}) \neq \psi_i'(\bar{x},\bar{y})$ and is of the form $t_i = t_j$ where $t_i \neq t_j \neq 0$, then one possibility is to try $\iota_k(\bar{x},\bar{y}) = \psi_i''(\bar{x},\bar{y})$ where $\psi_i''(\bar{x},\bar{y})$ is obtained from $\psi'(\bar{x},\bar{y})$ by replacing all occurences of t_i by t_j, and then if that fails, try replacing t_j by t_i. This appears to be sufficient for many applications, but if it is not we again revert to a (partial) case description to guide further substitutions.

4. If $P(\bar{x},\bar{y}) \neq \psi_i'(\bar{x},\bar{y})$, let $\iota_k(\bar{x},\bar{y}) = \psi_i'(\bar{x},\bar{y})$.

5. Let $\iota_k(\bar{x},\bar{y})$ be $P(\bar{x},\bar{y}) \Rightarrow \psi_i'(\bar{x},\bar{y})$.

6. If $P(\bar{x},\bar{y})$ and $\psi_i'(\bar{x},\bar{y})$ have been used by a previous rule and additional invariants are still needed, i.e., invariants for which

$\psi_i'(\bar{x},\bar{y})$ is no help in finding, we form a (partial) case description for the loop under consideration and attempt to extract the required invariants directly from it.

We reiterate at this point that these rules simply exhibit the general flavor of the approach rather than a detailed analysis of all heuristics which are employed. Let us now proceed to discuss the specifics of the method.

7. EXAMPLES AND DETAILS OF THE METHOD

We now want a descriptor to reflect only what is computed for a particular part of the program, i.e., we want to know what is computed in the loops requiring inductive assertions. As a result we must slightly modify the algorithm given in Definition 2 so that it reflects this change. This can be done by simply considering as the output variable set the local variables of the loop occurring on the left side of an assignment statement. Rather than terminating the process at the HALT node we necessarily stop immediately after exiting the loop.

We present three types of examples. The first illustrates the generation of an inductive assertion by using only the test predicate and output assertion. The second example shows the separate use of output-assertion-test-predicate analysis and a (partial) case description, obtaining parts of the inductive assertion from each. In the third example we discuss in detail how to combine the output assertion, test predicate, and a (partial)

case description to mechanically generate inductive assertions.

Notation. We assume that by now the reader is familiar with the derivation of a case description. As a result we henceforth present only interpreted (and simplified) descriptors in the tables. We also refer to the input, inductive, and output assertions as simply φ, ι, and ψ (respectively) since their arguments will be obvious from context.

Looking at Figure 4 (taken from [13]) we see a simple exchange sort program with $\psi = \forall M(2 \leq M \leq N \Rightarrow A(M-1) \leq A(M))$. This first example can be done with relative ease by some other methods as well. We consider it to illustrate how we process formulas in their "natural" form thus eliminating some intermediate manipulations.

By taking ψ backwards over $\delta(3,10,12)$ we get

$$I > N \Rightarrow (J = 0 \Rightarrow \forall M(2 \leq M \leq N \Rightarrow A(M-1) \leq A(M))).$$

Applying the rewrite rules of IMPLY [3], which is a natural-deduction-type system that processes formulas in their "natural" form, immediately yields

$$I > N \wedge J = 0 \Rightarrow \forall M(2 \leq M \leq N \Rightarrow A(M-1) \leq A(M)) \quad .$$

Rule 2 now gives the invariant

$$J = 0 \Rightarrow \forall M(2 \leq M < I \Rightarrow A(M-1) \leq A(M)) \quad .$$

We now proceed to examine some other more prominent characteristics of our method by considering the next two examples.

For the Wensley Division Algorithm of Figure 2 with $\psi = \{P/Q - E < Z \leq P/Q\}$ we try to find the appropriate ι at arc 7. We begin by extracting as much of the inductive assertion as possible from the output predicate ψ. Using Rule 1 we split ψ into $\psi_1 = P/Q - E < Z$ and $\psi_2 = Z \leq P/Q$. Dragging ψ_1 backwards to arc 7 we get

$$D < E \Rightarrow P/Q - E < Y \quad . \tag{10}$$

Doing the same with ψ_2 gives

$$D < E \Rightarrow Y \leq P/Q \quad . \tag{11}$$

Applying Rule 2 to (10) and (11), respectively, we get

$$P/Q - D \leq Y \wedge Y \leq P/Q \quad . \tag{12}$$

Thus far the derivation of ι is fairly standard.

We now observe local loop variables A and B which are not in (12). To establish additional loop invariants containing A and B, Rule 6 suggests we form a (partial) case description for the loop as seen in Col. 1 of Table 4. Looking again at Figure 2 we see that B and D are on one path through the loop and A and Y on the other.

We, therefore, attempt to relate them in this manner via the descriptors in Table 4.

Before considering our approach for relating these variables let us first look at another standard method for doing it. The method basically entails expressing A and Y after n iterations and then eliminating a factor common to both equations thus yielding an invariant. In the example under consideration, such expressions for A and Y would be

$$A^{(n)} = A^{(0)} + \sum_{i=1}^{n} B^{(i-1)} \quad , \tag{13}$$

$$Y^{(n)} = Y^{(0)} + \sum_{i=1}^{n} \frac{D^{(i-1)}}{2} \quad , \tag{14}$$

where the superscripts indicate the iteration count with a (0) superscript designating the initial value for the particular variable. We observe that (13) and (14) have no elements in common. So at this point we have no way to relate them; however, possibly we can relate B and D. Expressions for B and D are

$$B^{(n)} = B^{(0)} \cdot \prod_{i=1}^{n} \frac{1}{2} \quad , \tag{15}$$

$$D^{(n)} = D^{(0)} \cdot \prod_{i=1}^{n} \frac{1}{2} \quad . \tag{16}$$

Simplifying (15) and (16) and solving for the like term we have

$$\frac{D^{(n)}}{D^{(0)}} = \frac{1}{2^n} = \frac{B^{(n)}}{B^{(0)}}$$

which gives loop invariant

$$2B = QD \quad . \tag{17}$$

At this point we might be able to substitute the invariant (17) into (13) or (14) to possibly introduce a common variable. Suppose we choose to substitute $\frac{QD}{2}$ for B in (13). This would result in (13) being written as

$$A^{(n)} = A^{(0)} + Q^{(0)} \cdot \sum_{i=1}^{n} \frac{D^{(i-1)}}{2} \quad . \tag{18}$$

We can now eliminate the common term $\sum_{i=1}^{n} \frac{D^{(i-1)}}{2}$ from (14) and (18) to get

$$\frac{A^{(n)} - A^{(0)}}{Q} = Y^{(n)} - Y^{(0)} \quad . \tag{19}$$

Plugging the appropriate values into (19) yields the invariant

$$Y = A/Q \quad . \tag{20}$$

The reason for this lengthy description of an alternate method is to illustrate its dependence on several things. First of all it is often essential to find the invariants in the "proper" order to avoid unnecessary

attempts at simplification and elimination. In this example attempting to relate (13) and (14) was unnecessary since (17) needed to be found before (20). This problem could become greatly magnified by more complex programs. Secondly, in order to apply this technique one needs to be able to generalize each computational statement to the n-th interation. One also must have at his disposal a powerful simplifier to be able to manipulate the resultant expressions. A third deficiency in this approach is pointed out by the next example.

Our technique exhibits a very different approach. We, in a sense, try to eliminate the induction variable without generalizing and actually finding an expression for it. More specifically, we seek to eliminate what we will call induction constants. We use this term for the entities to be eliminated because they are "constants" that represent a combination of a constant and a particular value of the induction variable. For example, rather than generalizing to say n/10 (n being the usual induction variable) we might have 1/5. We work directly with the 1/5 ignoring the possibility of generalizing to the n-th level. We henceforth prime all induction constants to distinguish them from actual constants (unprimed).

The marking of induction constants is normally quite trivial and can be done during the descriptor generation process. While isolating the induction constants, we necessarily "factor out" of each expression the initial variable assignment in a manner similar to that done in say (13)-(16). For example, we would replace

$$X \leftarrow X/C \qquad \text{by} \quad X = X^{(0)} \cdot \zeta \ , \quad \text{and}$$

$$X \leftarrow X \pm C \qquad \text{by} \quad X = X^{(0)} \pm \zeta \ ,$$

where $X, C,$ and ζ denote a variable, constant, and induction constant respectively. These simple replacements can easily be extended to other forms of statements. Suppose we have a statement of the form $X \leftarrow X+Y$. Initially we would write $X \leftarrow X^{(0)} + Y^{(0)}$. However, subsequent iterations would depend on the form of Y. If Y is of the form $Y \leftarrow Y/C$, then following the initial iteration $X \leftarrow X+Y$ would be of the form $X \leftarrow X^{(0)} + Y^{(0)} \cdot \zeta$. These simple transformations suffice for the examples we consider here. Of course, additional replacement rules need to be added in general. However, it appears that this can normally be done with relative ease as circumstances demand. This "factoring out" of the initial assignment is reflected in Col. II of Table 4.

We again observe that path δ_1 does not include the assignments to A and Y but does cover the assignments to B and D. Using Col. II we therefore try to relate B and D but ignore A and Y. Our goal is to eliminate the induction constants between

$$B = (Q/2) \cdot \tfrac{1}{2}' \quad \text{and} \quad D = (1) \cdot \tfrac{1}{2}' \ . \tag{21}$$

Eliminating the $1/2'$ is trivial giving

$$B = (Q/2) * D \ . \tag{22}$$

δ_2 traverses both pairs of statements. From Col. II we get (21) then (22) as above and next consider $A = Q/2$ and $Y = 1/2$. Since this is the initial time through these statements we have no induction constant, i.e., $A \leftarrow A^{(0)} + B^{(0)}$ and $Y \leftarrow Y^{(0)} + D^{(0)}/2$. So in this initial step only, we can eliminate a constant between the two expressions. In this case we can easily eliminate the 1/2 getting

$$A = Q * Y \quad . \tag{23}$$

We note that normally we would go on to the next descriptor rather than attempt this type of substitution as we do not have an induction constant to guide the substitution.

Suppose we do go on to δ_3. This is almost the same as δ_1 so we can again relate B and D by eliminating the 1/4' to get (22).

Looking at δ_4 we see that it traverses both sets of statements as does δ_2. Eliminating the 1/4' between B and D again gives (22). We now want to eliminate the induction constants 1/2' and 1/4' from $A = (Q/2) \cdot 1/2'$ and $Y = 1/4'$, i.e., we want to find a k such that

$$c_1' = k \cdot c_2' \ ,$$

where k is a <u>constant</u> and c_1' and c_2' are the <u>induction constants</u> 1/2' and 1/4'. We easily solve $1/2' = k \cdot 1/4'$ to get $k = 2$. We now multiply $Y = 1/4'$ by 2 to get $2Y = 1/2'$. We can now eliminate the 1/2' to get (23), i.e.,

$$A = (Q/2)\cdot 2Y = Q * Y \quad .$$

Having established the same invariants for several paths, we now have a high degree of confidence that they are correct, i.e., that the inductive assertion is

$$(12) \wedge (22) \wedge (23) \quad .$$

An alternate approach would have been to simply start with say δ_4 (a path through both sets of statements) and use its result as the trial inductive assertion. We admit that this approach is not as universal as some other techniques, but when it works it is quite efficient and straight-forward, and when it does not we again point out that the human is left with the possibility of recognizing "patterns" in successive descriptors of the loop's (partial) case description.

As our next example we reconsider the multiplication program in Figure 3 with $\psi = Y = B * DA$. This example has a few subtle features that make the generation of its inductive assertions somewhat difficult. We look at two distinct approaches to finding an assertion at arc 2 before viewing ours.

The first is similar to the method of Wegbreit [21] and Katz and Manna's top-down method [12]. We begin by backing ψ up to arc 2 getting

$$A = 0 \Rightarrow Y = B * DA \quad . \qquad (24)$$

This suggests that there was an A in the expression Y = B * DA before exiting the loop that "disappeared" upon exit. Thus there are many possible inductive assertions, e.g., Y + A = B * DA, Y - A = B * DA, Y + A = B * DA + A, etc. The above methods suggest trying some of these (i.e., generate and try to prove the verification conditions) and see if they work. This is clearly undesirable.

The second approach is the one already discussed and is similar in content to the difference equations of Green [9] and the bottom-up method of Katz and Manna [12]. The difficulty with this approach is (i) the order-dependence problem and (ii) what we call the loop-dependence problem. Subpaths δ (9,10) and δ (8,11) (also δ (22,23) and δ (21,24)) both contain assignments to Y and XB. When this occurs it is necessary to know which branch was followed for each loop traversal to be able to analyze the computation. To do this an additional variable is added to keep the proper branch history. Furthermore, the statement A ← A-1 on arc δ (13,12) (also A ← A+1 on δ (26,25)) creates what we call a loop-dependence problem. For example, to represent the n-th iteration for the statements on subpath δ (7,8,11,6,12,13) one would tend to write

$$Y^{(n)} = Y^{(0)} + \sum_{i=1}^{n} 1 \quad ,$$

$$XB^{(n)} = XB^{(0)} - \sum_{i=1}^{n} 1 \quad , \quad A^{(n)} = A^{(0)} - \sum_{i=1}^{n} 1 \quad .$$

However, this cannot be the case since A ← A-1 is usually not traversed the same number of times as the other two statements. We must therefore

introduce another induction variable. The end result is numerous induction variables causing a potentially difficult generalization and (usually unsolvable) elimination procedure. The authors referred to make no claim to solve this problem. This should simply point out the difficulty in applying their techniques directly.

We are now in a position to demonstrate a way of finding the inductive assertions at arcs 2, 6, and 19 of Figure 3. The approach we take combines the use of the output assertion and the body of the loop to guide the search.

We begin by getting (24) and applying Rule 3 which says to generate a (partial) case description for the loop as seen in Table 5. It is important to note that in generating Table 5 we traverse δ (6,12) getting XB = 0. If we have an XB in the kernel set for this outer loop we set it to 0 since XB is "unknown" outside of its loop, viz. at arc 2.

Looking at Col. II, we know from (24) that for δ_1 we want to relate the elements of $\mathcal{X}'$ such that

$$\{Y = 1' \wedge B = 1' \wedge A = DA\text{-}1'\} \wedge A = 0 \Rightarrow Y = B * DA. \qquad (25)$$

Once in this form there are several straight-forward ways to eliminate the induction constants such that (25) is true. One is to simply substitute 0/A from the hypothesis giving

$$\{Y = 1' \wedge B = 1'' \wedge DA = 1'''\} \Rightarrow Y = B * DA \quad . \qquad (26)$$

To distinguish among the induction constants we mark each with a different number of primes. We do likewise in (25). Substituting all hypotheses of (26) into its conclusion we get

$$'T' \Rightarrow 1' = 1'' * 1''' \qquad (27)$$

which is true. We now back substitute from (25) into (27), replacing each differently marked induction constant by the appropriate expression i.e., Y/1', B/1", and DA-A/1'", to get the invariant

$$Y = B * (DA-A) \quad . \qquad (28)$$

The identical result can be found in a similar manner for δ_2, δ_3, ...

We now attempt to find the inductive assertion ι_6 at arc 6. This again would be difficult to find using some of the other standard techniques.

Since we have already found (28) to be an invariant at arc 2, we use it as a new φ thus beginning the algorithm in Definition 2 at arc 2 with φ set to (28). We do this since we know that we will have to prove the verification condition

$$(28) \wedge \delta(2,3,4,5,6) \Rightarrow \iota_6 \quad ,$$

the hypothesis being the result obtained by moving (28) forwards over the path specified by δ. In other words, by viewing (28) as a new φ

we seek to use this fact as much as possible in finding ι_6.

Since we know that (28) is an invariant at arc 2 we back it up to arc 6 over path $\delta(2,14,13,12,6)$ to obtain

$$XB = 0 \Rightarrow Y = B * (DA - (A-1)) \quad . \tag{29}$$

Again Rule 3 specifies the generation of a (partial) case description to avoid the proliferation of possible assertions at arc 6. Viewing (28) as a new φ in the generation process we get Table 6. Similar to (25) we consider

$$\{Y = B * (DA-A) + 1' \wedge XB = B - 1'\} \wedge XB = 0 \Rightarrow Y = B * (DA - (A-1)) \quad . \tag{30}$$

By eliminating the $1'$ we immediately get the loop invariant

$$Y = B * (DA-A) + (B-XB) \quad .$$

The same invariant results for $\delta_2, \ldots$; the inductive assertion at arc 19 also follows easily by the same method.

We make the observation at this point that in more complex examples we might have many more possibilities for the substitution of inductive constants. However, the information obtained directly from the program in the form of a (partial) case description plus the use of known assertions as done above often quickly guide us to the proper result. If not, we can

exploit the technique's interactive qualities.

8. CONCLUSION

We have now shown that the technique is capable of handling fairly complex programs. We have used the notion of a (partial) case description as a tool for identifying eraoneous program paths, "checking" output assertions, generating inductive assertions, and as the basis for man-machine interaction.

At present we expect to develop an interactive system allowing the human to recognize "patterns" in the descriptors. As pointed out, this human intervention would normally occur only when our heuristic rules do not apply or when the appropriate assertion becomes obvious to the user thus making it expeditious to terminate the generation process. It is clear, however, that more and better heuristic rules are needed. We anticipate new rules which interface smoothly with (partial) case descriptions to surface as we gain experience with the system.

The theorem-proving required is within the scope of some present automatic theorem proving systems which are adept at simplification and proving verification conditions (see, e.g., Bledsoe [1,3] or Deutsch [5]).

We emphasize that we do not foresee this or any other proposed technique bringing to fruition the practical verification of large, complex programs which now exist and were written in an arbitrary fashion. However, we do feel that many of the ideas developed here will carry over almost directly into an environment in which the programmer writes

hierarchically well-structured programs creating the Floyd assertions along with the program construction rather than ex post facto. We anticipate that in this case our tools could provide a practical basis for the verification of large, complicated programs.

ACKNOWLEDGMENTS

I am indebted to my supervisor Professor W.W. Bledsoe for his help and continual encouragement. Thanks is also due to Dr. Dallas Lankford for his numerous suggestions throughout this research.

REFERENCES

1. Bledsoe, W.W.; "Program correctness." Departments of mathematics and computer sciences report, ATP-15, The Univ. of Texas, Austin, Tx., Jan. 1974.

2. Bledsoe, W.W.; R.S. Boyer; and W.H. Henneman; "Computer proofs of limit theorems." Artificial intelligence, 3(1972), 27-60.

3. Bledsoe, W.W.; and P. Bruell; "A man-machine theorem-proving system." IJCAI-3 (Aug. 1973), 56-65; also in Artificial intelligence, 5(1974), 51-72.

4. Church, A.; Introduction to mathematical logic. Vol. 1, Princeton Univ. Press, N.J., 1956.

5. Deutsch, L.P.; "An interactive program verifier." Ph.D. Dissertation, Univ. of California, Calif., June 1973.

6. Elspas, B.; K.N. Levitt; and R.J. Waldinger; "An interactive system for the verification of computer programs." SRI Project 1891, Stanford Research Institute, Menlo Park, Calif., Sept. 1973.

7. Elspas, B.; K.N. Levitt; R. Waldinger; and A. Waksman; "An assessment of techniques for proving program correctness." Computing surveys, 4, 2 (June 1972), 97-147.

8. Floyd, R.W.; "Assigning meanings to programs." Mathematical aspects of computer science, J.T. Schwartz, ed., Vol. 19 (American mathematical society, Providence, R.I. (1967)).

9. Green, M.W.; "The use of difference equations as an aid to specifying assertions." "Research in interactive program proving techniques." SRI Report 8398-II, Stanford Research Institute, Menlo Park, Calif., May 1972.

10. Grief, I.; and R.J. Waldinger; "A more mechanical heuristic approach to program verification." Proceedings, Colloque sur la Programmation, Institut de Programmation, Paris (April 1974).

11. Kaplan, D.M.; "Regular expressions and the equivalence of programs." J. comp. and sys. sciences 3, 4 (Nov. 1969), 361-386.

12. Katz, S.M.; and Z. Manna; "A heuristic approach to program verification." IJCAI-3 (Aug. 1973), 500-512.

13. King, J.C.; "A program verifier." Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1969.

14. Luckham, D.C.; D.M.R. Park; and M.S. Paterson; "On formalized computer programs." J. comp. and sys. sciences 4, 3 (June 1970), 220-249.

15. Manna, Z.; "Properties of programs and the first-order predicate calculus." J. ACM 16, 2 (April 1970), 244-255.

16. Mendelson, E.; Introduction to mathematical logic. Van Nostrand Co., Princeton Univ. Press, Princeton, N.J., 1964.

17. Moriconi, M.S.; "Semiautomatic synthesis of inductive predicates." Departments of mathematics and computer sciences report, ATP-16, The Univ. of Texas, Austin, Tx., June 1974.

18. Naur, P.; "Proof of algorithms by general snapshots." BIT 6, 4 (1966), 310-316.

19. Pratt, T.W.; "Case descriptions of programs: an informal introduction." The Univ. of Texas C.S. report TSN-32, Austin, Tx., Oct. 1972.

20. Shoenfield, J.; Mathematical logic. Addison-Wesley, 1967.

21. Wegbreit, B.; "The synthesis of loop predicates." Comm. ACM 17, 2 (Feb. 1974), 102-112.

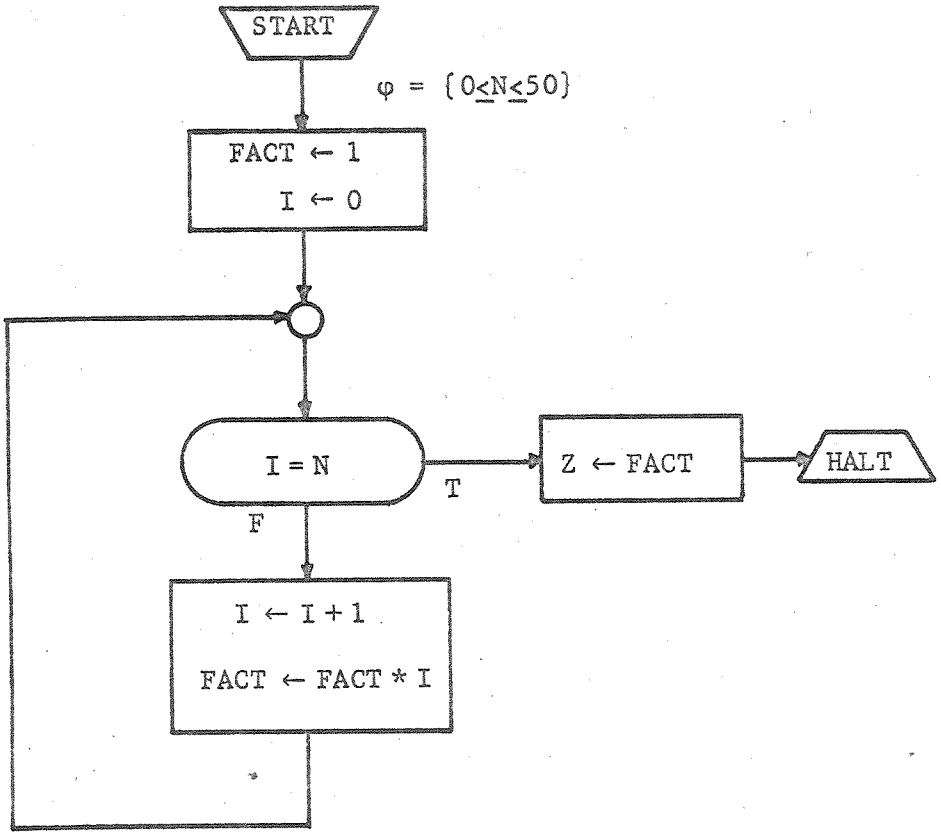


FIGURE 1. FACTORIAL PROGRAM

	Unreduced Descriptor	Descriptor	Interpreted Descriptor
1	(0≤N≤50); FACT←1; I←0; (I=N); Z←FACT	$\mathscr{C}$ = {0≤N≤50, 0=N} $\mathscr{K}$ = {Z←1}	$\mathscr{C}$ = {N=0} $\mathscr{K}$ = {Z←1}
2	(0≤N≤50); FACT←1; I←0; ∼(I=N); I←I+1; FACT←FACT*I; (I = N); Z←FACT	$\mathscr{C}$ = {0≤N≤50, ∼(0=N), (0 + 1) = N} $\mathscr{K}$ = {Z←1*(0+1)}	$\mathscr{C}$ = {N=1} $\mathscr{K}$ = {Z←1}
3	(0≤N≤50); FACT←1; I←0; ∼(I=N); I←I+1; FACT←FACT*I; ∼(I=N); I←I+1; FACT←FACT*I; (I=N); Z←FACT	$\mathscr{C}$ = {0≤N≤50, ∼(0=N), ∼((0 + 1) = N), (0 + 1) + 1 = N} $\mathscr{K}$ = Z ← (1*(0+1))* ((0+1) + 1)}	$\mathscr{C}$ = {N=2} $\mathscr{K}$ = {Z←2}
	. . .	. . .	. . .
52	(0≤N≤50): FACT←1; I←0; [∼(I=N); I←I+1; FACT←FACT*I]^{51}; (I=N); Z←FACT	$\mathscr{C}$ = {0≤N≤50, ∼(N=0), ..., ∼(N=50), N=51} $\mathscr{K}$ = {Z←1*1*2*...*51}	□

Table 1. (Partial) Case Description for Factorial Program of Figure 1.

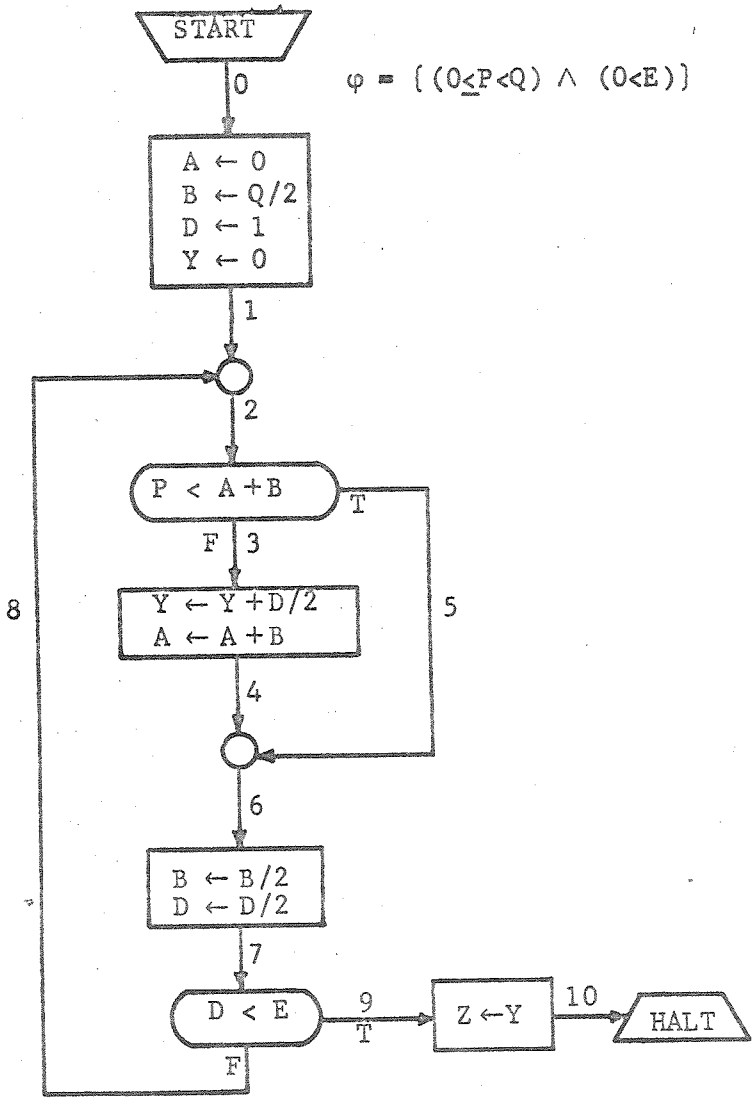


FIGURE 2. WENSLEY'S QUOTIENT ALGORITHM

Descriptor	Interpreted Descriptor
$\delta_1 = \delta(0,1,2,5,6,7,9,10)$ $\mathscr{C} = \{0 \leq P < Q,\ 0 < E,\ P < 0 + Q/2,$ $1/2 < E\}$ $\mathscr{K} = \{Z \leftarrow 0\}$	$\mathscr{C} = \{P/Q - E < 0 \leq P/Q < 1/2\}$ $\mathscr{K} = \{Z \leftarrow 0\}$
$\delta_2 = (0,\ldots,7,9,10)$ $\mathscr{C} = \{0 \leq P < Q,\ 0 < E,\ \sim(P < 0 + Q/2),$ $1/2 < E\}$ $\mathscr{K} = \{Z \leftarrow (0 + 1/2)\}$	$\mathscr{C} = \{P/Q - E < 1/2 \leq P/Q < 1$ $\mathscr{K} = \{Z \leftarrow 1/2\}$
$\delta_3 = (0,1,2,5,6,7,8,2,5,6,7,9,10)$ $\mathscr{C} = \{0 \leq P < Q,\ 0 < E,\ P < 0 + Q/2,$ $\sim(1/2 < E),\ P < 0 + ((Q/2)/2),$ $((1/2)/2) < E\}$ $\mathscr{K} = \{Z \leftarrow 0\}$	$\mathscr{C} = \{P/Q - E < 0 \leq P/Q < 1/4$ $\mathscr{K} = \{Z \leftarrow 0\}$
. . .	. . .

Table 2. (Partial) Case Description for Wensley Algorithm of Figure 2.

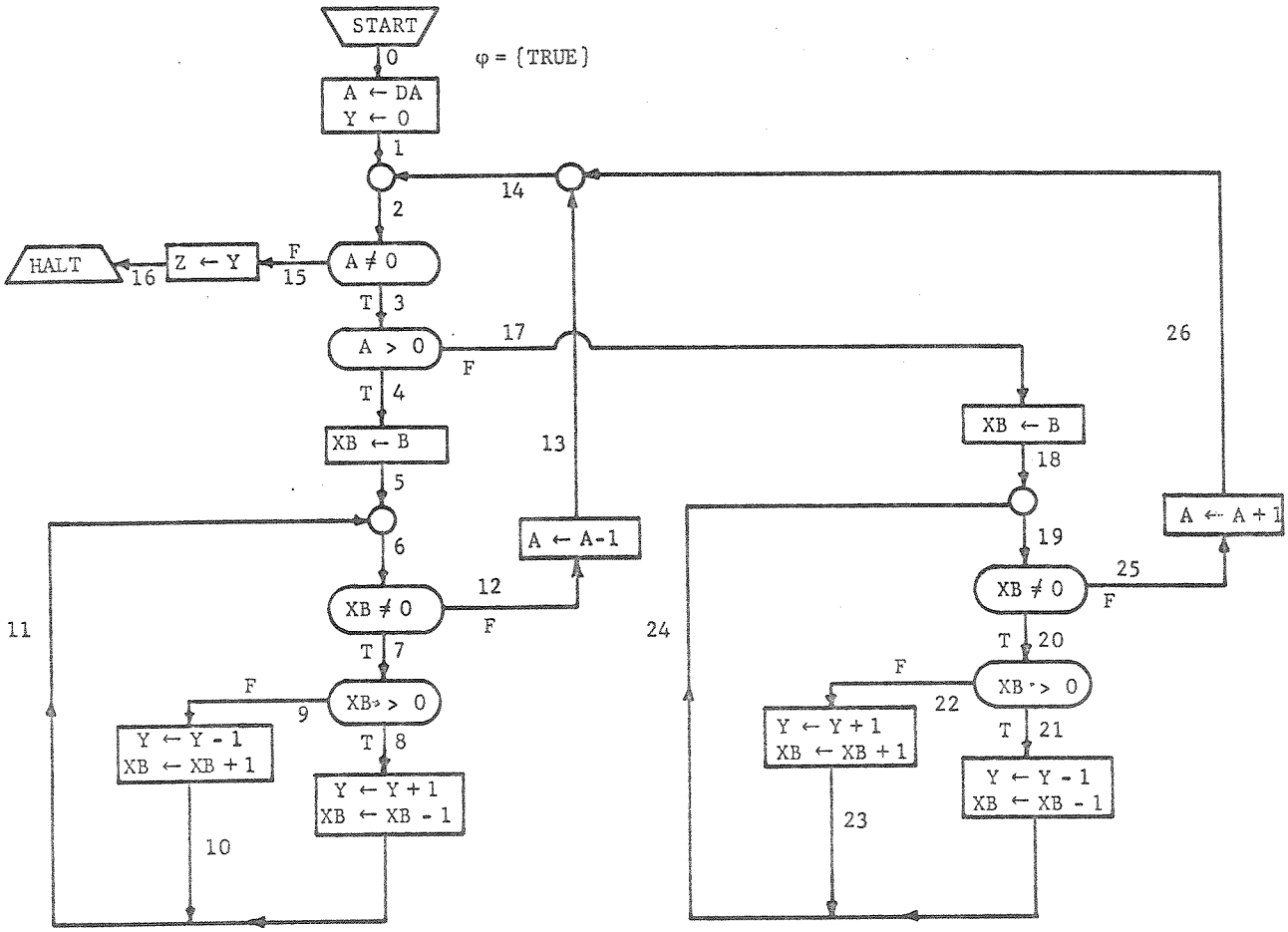


FIGURE 3. MULTIPLICATION PROGRAM

Descriptor	Interpreted Descriptor
$\delta_1 = \delta(0,\ldots,6,12,13,14,2,15,16)$ $\mathscr{C} = \{DA\neq 0,\ DA>0,\ \sim(B\neq 0),\ \sim(DA-1\neq 0)\}$ $\mathscr{K} = \{Z\leftarrow 0\}$	$\mathscr{C} = \{B=0,\ DA=1\}$ $\mathscr{K} = \{Z\leftarrow 0\}$
$\delta_2 = \delta(0,\ldots,8,11,6,12,13,14,2,15,16)$ $\mathscr{C} = \{DA\neq 0,\ DA>0,\ B\neq 0,\ B>0,\ \sim(B-1\neq 0),$ $\sim(DA-1\neq 0)\}$ $\mathscr{K} = \{Z\leftarrow(0+1)\}$	$\mathscr{C} = \{B=1,\ DA=1\}$ $\mathscr{K} = \{Z\leftarrow 1\}$
$\delta_3 = \delta(0,\ldots,3,17,\ldots,21,24,19,25,26,$ $14,2,15,16)$ $\mathscr{C} = \{DA\neq 0,\ \sim(DA>0),\ B\neq 0,\ B>0,\ \sim(B-1\neq 0),$ $\sim(DA+1\neq 0)\}$ $\mathscr{K} = \{Z\leftarrow(0-1)\}$	$\mathscr{C} = \{B=1,\ DA=-1\}$ $\mathscr{K} = \{Z\leftarrow -1\}$
$\delta_4 = \delta(0,\ldots,8,11,6,7,9,10,11,6,12,13,$ $14,2,15,16)$ $\mathscr{C} = \{DA\neq 0,\ DA>0,\ B\neq 0,\ B>0,\ B-1\neq 0,$ $\sim(B-1>0),\ \sim((B-1)+1\neq 0),$ $\sim(DA-1\neq 0)\}$ $\mathscr{K} = \{Z\leftarrow((0+1)-1)\}$	$\square$
$\delta_5 = \delta(0,\ldots,8,11,6,7,8,11,6,12,13,14,2,$ $15,16)$ $\mathscr{C} = \{DA\neq 0,\ DA>0,\ B\neq 0,\ B>0,\ B-1\neq 0$ $\sim((B-1)-1\neq 0),\ \sim(DA-1\neq 0)\}$ $\mathscr{K} = \{Z\leftarrow(0+1)+1\}$	$\mathscr{C} = \{B=2,\ DA=1\}$ $\mathscr{K} = \{Z\leftarrow 2\}$
. . .	. . .

Table 3. (Partial) Case Description for Multiplication Program of Figure 3.

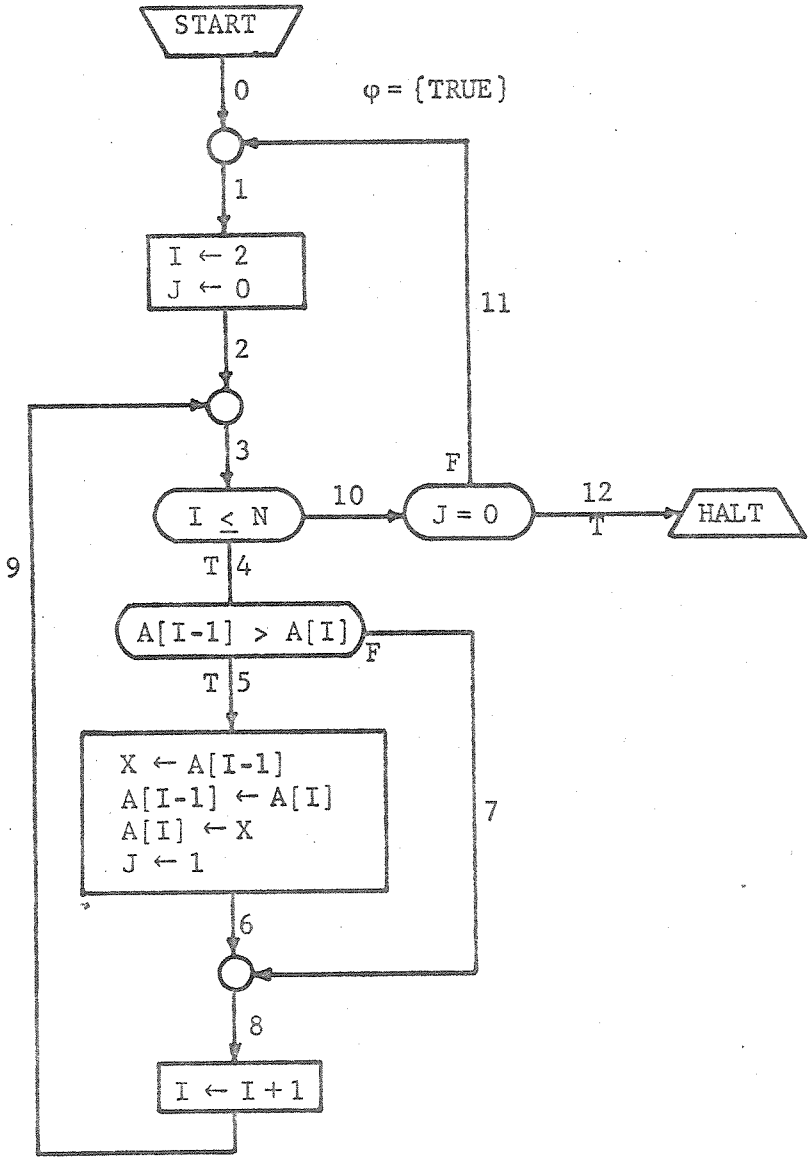


FIGURE 4. SIMPLE EXCHANGE SORT

I	II
$\delta_1 = \delta(0,1,2,5,6,7,9)$ $\mathscr{C} = \{0 \leq P < Q,\ E > 1/2\}$ $\mathscr{K} = \{A \leftarrow 0,\ B \leftarrow Q/4,\ D \leftarrow 1/2,\ Y \leftarrow 0\}$	$\mathscr{K}' = \{A = 0,\ B = (Q/2)\cdot\frac{1}{2}',$ $D = (1)\cdot\frac{1}{2}',\ Y = 0\}$
$\delta_2 = \delta(0,\ldots,4,6,7,9)$ $\mathscr{C} = \{Q/2 \leq P < Q,\ E > 1/2\}$ $\mathscr{K} = \{A \leftarrow Q/2,\ B \leftarrow Q/4,\ D \leftarrow 1/2,\ Y \leftarrow 1/2\}$	$\mathscr{K}' = \{A = Q/2,\ B = (Q/2)\cdot\frac{1}{2}',$ $D = (1)\cdot\frac{1}{2}',\ Y = \frac{1}{2}\}$
$\delta_3 = \delta(0,1,2,5,6,7,8,2,5,6,7,9)$ $\mathscr{C} = \{0 \leq P < Q/2,\ 1/4 < E \leq 1/2\}$ $\mathscr{K} = \{A \leftarrow 0,\ B \leftarrow Q/8,\ D \leftarrow 1/4,\ Y \leftarrow 0\}$	$\mathscr{K}' = \{A = 0,\ B = (Q/2)\cdot\frac{1}{4}',$ $D = (1)\cdot\frac{1}{4}',\ Y = 0\}$
$\delta_4 = \delta(0,1,2,5,\ldots,8,3,4,6,7,9)$ $\mathscr{C} = \{Q/4 \leq P < Q/2,\ 1/4 < E \leq 1/2\}$ $\mathscr{K} = \{A \leftarrow Q/4,\ B \leftarrow Q/8,\ D \leftarrow 1/4,\ Y \leftarrow 1/4\}$	$\mathscr{K}' = \{A = (Q/2)\cdot\frac{1}{2}',\ B = (Q/2)\cdot\frac{1}{4}',$ $D = (1)\cdot\frac{1}{4}',\ Y = \frac{1}{4}'\}$
. . .	. . .

Table 4. From Loop in Wensley Algorithm of Figure 2.

I	II
$\delta_1 = \delta(0,\ldots,8,11,6,12,13,14,2,15)$ $\mathscr{C} = \{DA = 1,\ B = 1\}$ $\mathscr{K} = \{Y \leftarrow 1,\ B \leftarrow 1,\ A \leftarrow DA\text{-}1\}$	$\mathscr{K}' = \{Y = 1',\ B = 1',\ A = DA\text{-}1'\}$
$\delta_2 = \delta(0,\ldots,8,11,6,7,8,11,6,12,13,14,2,15)$ $\mathscr{C} = \{DA = 1,\ B = 2\}$ $\mathscr{K} = \{Y \leftarrow 2,\ B \leftarrow 2,\ A \leftarrow DA\text{-}1\}$	$\mathscr{K}' = \{Y = 2',\ B = 2',\ A = DA\text{-}1'\}$
. . .	. . .

Table 5.

I	II
$\delta_1 = \delta(2,\ldots,8,11,6,12)$ $\mathscr{C} = \{A \geq 1,\ B = -1\}$ $\mathscr{K} = \{Y \leftarrow B^*(DA\text{-}A) + 1,\ XB \leftarrow B\text{-}1\}$	$\mathscr{K}' = \{Y = B * (DA\text{-}A) + 1',\ XB = B\text{-}1'\}$
$\delta_2 = \delta(2,\ldots,7,9,10,11,6,12)$ $\mathscr{C} = \{A \geq 1,\ B = -1\}$ $\mathscr{K} = \{Y \leftarrow B^*(DA\text{-}A)\text{-}1,\ XB \leftarrow B + 1\}$	$\mathscr{K}' = \{Y = B * (DA\text{-}A)\text{-}1',\ XB = B + 1'\}$
. . .	. . .

Table 6.

Tables 5 and 6 are from the Multiplication Program of Figure 3.