AN ALGEBRAIC PROGRAM VERIFICATION METHOD APPLIED TO MICROPROGRAMS

A. Blikle*

Computation Center
Polish Academy of Sciences
00-901 Warsaw, Poland

S. Budkowski**

Dept. of Computer Science
Warsaw Technical University
00-655 Warsaw, Poland

Research Report CS-76-31

June 1976

* Part of this work was done when the author was visiting the University of Waterloo. This part was supported by the National Research Council of Canada under Grant A-1617.

** Part of this work was done when the author was visiting the Department of Computer Science, University of Maryland, College Park, MD.20742.

Abstract

This program-verification method provides a mathematical framework for proving input-output properties (such as partial and total correctness) of iterative programs. Technically it uses a calculus of binary relations extended with fixed-point equations. The method has been tested on several microprograms of a computer's arithmetical unit. One example of such a microprogram and its correctness proof is discussed in the paper.

1. Introduction

In the current literature of the subject one can distinguish two different trends of attacking the problem of the mathematical verification of microprograms. In one approach (A. Birman, B. Leeman and W. Carter [2],[8],[10],[11]) the analyzed microprogram and its expected meaning are described by two abstract machines one of which is defined on the hardware level and the other on the level of architecture. The correctness proofs consist of showing that one of these machines simulates the other in the sense defined by R. Milner [14]. In the other approach the expected meaning of a microprogram is described by more mathematical (or less operational) terms either by verification conditions (D.A. Patterson [15]), which involves Floyd's method, or by regular expressions (T. Ito [9]) which involves the algebra of events together with the related fixed-point equations. In the latter two cases one refers to some standard mathematical methods of software-program verification.

This paper presents another software-program verification method (A. Blikle and A. Mazurkiewicz [3],[4],[5],[7],[13]) applied to micro-programs [6]. The general idea of this method is the following: Given a program Π its meaning is assumed to be a binary input-output relation R (I-O relation) which describes the mapping of the initial values of the vector of variables into the terminal values of this vector. To establish R explicitly we split the program Π into a finite number of modules $\Pi_1,\ldots,\Pi_n$ (e.g. assignment statements and tests) which must be simple enough for their I-O relations $R_1,\ldots,R_n$ to be obvious. Since the program

Π is a combination of $\Pi_1,\ldots,\Pi_n$ the relation R must be a combination of $R_1,\ldots,R_n$:

$$R = \Psi(R_1,\ldots,R_n) \tag{1.1}$$

The function Ψ is defined in the algebra of relations (Sec.2) and describes the control structure of Π. To find the function Ψ we use an algebraic method which consists of writing and solving a set of fixed-point equations in our algebra. Once Ψ has been found, we use (1.1) in the further analysis of Π. This analysis is carried out in the algebra of relations and permits proofs of partial as well as total correctness of programs.

The general method above was applied by the authors to several arithmetical microprograms of a floating-point arithmetical unit designed at the Warsaw Technical University. This application has raised a problem which usually is neglected in the consideration of software programs. Namely, the arithmetical microprograms involve computer arithmetics which is fairly different from the usual (Peano's) arithmetics. E.g. in the computer arithmetics the law of the distributivity of multiplication does not hold. The lack of this property makes the calculations which appear in the verification of the program practically impossible. To solve this problem we consider two programs Π_1 and Π_2, where Π_1 is the "real" microprogram and Π_2 is an abstract program resulting from Π_1 by the replacement of the machine operations by the corresponding arithmetical ones. We verify the program Π_2 and then we show that Π_2 simulates Π_1 in the following sense: Let F_1 and F_2 denote the I-O functions of Π_1 and Π_2

respectively, let D_1 and D_2 denote the input domains of Π_1 and Π_2 (i.e. the domains of F_1 and F_2) and let $D_1 \subseteq D_2$. There exists a function $T:D_2 \to D_1$ such that $T(d) = d$ for all $d \in D_1$ and $F_1(T(d)) = T(F_2(d))$ for all $d \in D_2$. Now all the I-O properties of Π_2 can easily be "translated" into the I-O properties of Π_1. This concept of simulation coincides, of course, with the algebraic simulation of R. Milner [14] which apparently makes our approach similar to that of A. Birman, B. Leeman and W. Carter. As a matter of fact, however, our program verification is not restricted to the proof of simulation but also provides the proof of the total correctness of Π_2. The latter is carried out in the algebra of binary relations extended with fixed-point methods.

Regarding the general philosophy of the approach our attempt was to make this approach as close as possible to the style of calculations performed by engineers. For instances, designing electrical circuits one also "proves" - in a sense - their properties, but the "proofs" consist of solving Kirhoff's - or others - equations and in analysing the solutions obtained. In our approach we deal with programs but we proceed in a very similar way.

The organization of the paper is the following: First we describe the general Blikle-Mazurkiewicz method which is slightly modified here in regard to the form of fixed-point equations. This frees the reader of looking with the references which are not readily available. Next we show a detailed example of the verification of a software program Π_3. This program performs the Booth fixed-point multiplication algorithm and is a simplified version of an abstract program Π_2 which in

turn simulates the real microprogram Π_1. We describe the modifications required in Π_3 to get Π_2 from it and the modifications of Π_2 which result Π_1. Referring to the analysis of Π_3 we describe briefly the analysis of Π_2 and show the function of simulation T between Π_1 and Π_2.

2. The algebra of binary relations

The algebra of binary relations is a common mathematical tool used in the mathematical theory of programs. Below we describe some principles of this algebra together with the notation used in the sequel.

Let D be an arbitrary nonempty set called the domain and interpreted as the set of all possible states of the vector of variables in a program. By Rel(D) we denote the set of all binary relations in D, i.e.

$$\mathrm{Rel}(D) = \{R \mid R \subseteq D \times D\}$$

For any a,b in D and R in Rel(D) we shall write aRb for $(a,b) \in R$. By ϕ we shall denote the empty relation, and by I the identity relation, i.e. $I = \{(a,a) \mid a \in D\}$.

Basic operations in the set Rel(D), which we shall use in the sequel, are defined below. Let $R_1, R_2 \in \mathrm{Rel}(D)$.

$$R_1 \cup R_2 = \{(a,b) \mid aR_1b \vee aR_2b\} \quad \text{- union}$$

$$R_1 \circ R_2 = \{(a,b) \mid (\exists c)aR_1c \;\&\; cR_2b\} \quad \text{- composition}$$

$$R_1^0 = I \quad \text{- 0-th power}$$

$$R_1^n = R_1^{n-1} \circ R_1 \quad \text{- n-th power}$$

$$R_1^* = I \cup R_1 \cup R_1 \circ R_1 \cup R_1 \circ R_1 \circ R_1 \cup \ldots = \bigcup_{n=0}^{\infty} R_1^n \quad \text{- *-iteration}$$

$$R_1^+ = R_1 \cup R_1 \circ R_1 \cup R_1 \circ R_1 \circ R_1 \cup \ldots = \bigcup_{n=1}^{\infty} R_1^n \quad \text{- +-iteration}$$

Interpretation. The operations $\cup$, $\circ$, $*$ are used in the descriptions of the I-O relations of programs; precisely speaking they are used to describe explicitly the function Ψ in (1.1). Fig.1 shows the interpretation of the operations defined in this section.

□

Below we list the most important properties of these operations. Here and in the sequel we shall omit the symbol "$\circ$" of composition and write R_iR_j instead of $R_i \circ R_j$.

1) $R_1(R_2R_3) = (R_1R_2)R_3$ - associativity

2) $R_1(R_2 \cup R_3) = R_1R_2 \cup R_1R_3$ - finite distributivity

$(R_2 \cup R_3)R_1 = R_2R_1 \cup R_3R_1$

3) $R_0(\bigcup_{i=1}^{\infty} R_i) = \bigcup_{i=1}^{\infty} R_0R_i$ - infinite distributivity

$(\bigcup_{i=1}^{\infty} R_i)R_0 = \bigcup_{i=1}^{\infty} R_iR_0$

4) $RI = IR = R$ - the unit property of I

5) $R\phi = \phi R = \phi$ - the zero property of ϕ

6) $R^* = I \cup R^+$

$R^+ = RR^* = R^*R$

To deal with concrete programs (and microprograms) and to carry out their analysis we shall need an explicit notation to specify the I-O relations. Since we are going to deal only with deterministic programs, we can restrict ourselves to the case of partial functions and use the notation introduced by A. Mazurkiewicz [7]. A relation $R \subseteq D \times D$ is a partial function if for any $d_1 \in D$ there is at most one $d_2 \in D$ such that d_1Rd_2.

Let $f:D \to D$ be an arbitrary partial function and let $p:D \to \{true,false\}$ be an arbitrary predicate such that if $p(d) = true$ then $f(d)$ is defined. We denote by

$$[p(x)|\, x := f(x)] = \{(d_1,d_2)|\, p(d_1) = true \;\&\; d_2 = f(d_1)\} \qquad (2.1)$$

Of course, $[p(x)|x := f(x)]$ is a partial function whose domain is $\{d|\, d \in D \;\&\; p(d) = true\}$. For the sake of simplicity we shall also write

$$[x := f(x)] \quad \text{for} \quad [true\ x|:= f(x)]$$

and

$$[p(x)] \quad \text{for} \quad [p(x)|x := x]$$

Of course $[p(x)][x := f(x)] = [p(x)|x := f(x)]$. In the sequel we shall use the following equivalences which can be proved easily from (2.1):

$$\begin{aligned}
&1) \quad [p(x)|x := f(x)][q(x)|x := g(x)] = \\
&\qquad = [p(x) \;\&\; q(f(x))|x := g(f(x))] \\
&2) \quad [p(x)|x := f(x)] \cup [q(x)|x := f(x)] = \\
&\qquad = [p(x) \vee q(x)|x := f(x)] \qquad (2.2) \\
&3) \quad [p(x)][q(x)] = [q(x)][p(x)] = [p(x) \;\&\; q(x)] \\
&4) \quad \text{If } p(x) \Rightarrow q(f(x)) \text{ then } [p(x)|x := f(x)][q(x)] = \\
&\qquad = [p(x)|x := f(x)] \,.
\end{aligned}$$

3. The mathematical models of programs

In order to define in a rigorous way the concept of the I-O relation, we need here a rigorous concept of a program. To this effect, we shall use the notion of an algorithm introduced by A. Mazurkiewicz [7].

By an algorithm we shall mean any system $A = (D,V,\alpha_1,\mathcal{J})$ where

D is an arbitrary nonempty set called the domain of the algorithm and is interpreted as in Sec.2,

$V = \{\alpha_1,\ldots,\alpha_n\}$ is a finite nonempty set of elements called labels of the algorithm,

α_1 is a distinguished element of V called the initial label of the algorithm,

$\mathcal{J} = \{(\alpha_i,R_{ij},\alpha_j) | R_{ij} \in Rel(D);\ i,j \leq n\}$ is a set of $p = n^2$ triples called instructions. It is implicit in the notation above that for any α_i and α_j there is exactly one R_{ij} such that $(\alpha_i,R_{ij},\alpha_j) \in \mathcal{J}$. Usually many of the R_{ij} relations will be empty. An instruction with $R_{ij} = \phi$ describes the fact that there is no direct trespassing between α_i and α_j in the program. Given an instruction, the corresponding α_i, R_{ij} and α_j are called the entrance label, the action and the exit label.

Interpretation. The algorithms of Mazurkiewicz are used to describe (formalize) flowchart programs. The α_i's are the control states and the R_{ij}'s define the meaning of "boxes". E.g. the flowchart of Fig.2 corresponds to the set of two instructions:

$\mathcal{J} = \{(\alpha_1,[p(x)|x := f(x)],\alpha_2);(\alpha_1,[\sim p(x)|x := g(x)],\alpha_3)\}$ □

As mentioned in Sec.2 we are going to apply our theory to deterministic programs only. Nevertheless, the theory itself will be developed in the general nondeterministic case which makes its presentation much simpler.

Consider an arbitrary algorithm $A = (D,V,\alpha_1,\mathfrak{I})$ where $V = \{\alpha_1,\ldots,\alpha_n\}$. For any $\alpha_i,\alpha_j \in V$, by an (α_i,α_j)-run we shall mean any sequence of instructions of $\mathfrak{I}$:

$$(\alpha_{i_1},R_1,\alpha_{j_1}); \ \ldots \ ; (\alpha_{i_k},R_k,\alpha_{j_k}) \tag{3.1}$$

such that $\alpha_{i_1} = \alpha_i$; $\alpha_{j_k} = \alpha_j$ and $\alpha_{j_p} = \alpha_{i_{p+1}}$ for $p \leq k-1$. Of course, an (α_i,α_j)-run is simply a path in the graph of A. The corresponding sequence of actions $(R_1,\ldots,R_k)$ will be called an (α_i,α_j)-symbolic execution (abbr. s. execution). Let $Exec(\alpha_i,\alpha_j)$ denote the set of all the (α_i,α_j)-s. executions in A. The (α_i,α_j)-resulting relation is defined as follows:

$$Res(\alpha_i,\alpha_j) = U\{R_1\circ\ldots\circ R_k \mid (R_1,\ldots,R_k) \in Exec(\alpha_i,\alpha_j)\} \tag{3.2}$$

This is of course the I-O relation of A under the assumption that α_i is the input label and α_j is the output label. Indeed, $d_1 Res(\alpha_i,\alpha_j) d_2$ iff there exists an (α_i,α_j)-s. execution $(R_1,\ldots,R_k)$ such that $d_1 R_1\circ\ldots\circ R_k d_2$. Observe that in any (α_i,α_j)-run the control of the algorithm may pass through α_i and α_j many times.

By the definition of A the label α_1 is assumed to be initial and therefore we shall be interested mostly in the rleations $R(\alpha_1,\alpha_j)$ for $j = 1,\ldots,n$. Moreover, among these n relations we shall select usually some number of $k \leq n$ relation that correspond to the actual outputs of the program. The particular one-output case corresponds to $k = 1$, but in some applications we may want to consider programs with more than one output (e.g. the successful termination, the overflow and the underflow).

Now suppose we are considering an algorithm A where α_n has been chosen to be the terminal label and suppose that we have proved

$$Res(\alpha_1,\alpha_n) = [p(x)|x := f(x)]. \tag{3.3}$$

By the definition of $Res(\alpha_i,\alpha_j)$ this implies the following about A:

1) for every initial $d \in D$ the algorithm terminates (stops) if and only if p(d) is satisfied,

2) for every initial $d \in D$ if the algorithm terminates, then the terminal value is f(d).

Of course 2) is a partial-correctness property of A and 1) defines exactly the domain of termination. Consequently, (3.3) is the strongest total-correctness property of A, since p(x) is not only sufficient but is also a necessary condition of termination (for the concepts of partial and total correctness see [12]).

4. Fixed-point equations and programs

Dealing with concrete programs we shall attempt to express their resulting relations $Res(\alpha_1,\alpha_j)$ in terms of the actions of instructions R_{ij} and the operations defined in Sec.2. The definition (3.2) does not indicate, however, how to do it. Below we present a method of fixed-point equations which permits to achieve this goal.

Let $A = (D,V,\alpha_1,\mathcal{J})$ be an arbitrary algorithm with $V = (\alpha_1,\ldots,\alpha_n)$. By the canonical set of equation (CSE) of A we mean the set:

$$\begin{aligned} X_1 &= X_1R_{11} \cup \ldots \cup X_nR_{n1} \cup R_{11} \\ &\ldots \\ X_n &= X_1R_{n} \cup \ldots \cup X_nR_{nn} \cup R_{1n} \end{aligned} \tag{4.1}$$

where every R_{ij} is, of course, the action of the instruction $(\alpha_i, R_{ij}, \alpha_j)$ in $\mathfrak{I}$. The unknowns X_i of (4.1) range, of course, over the set Rel(D) of relations.

Any vector $(P_1,\ldots,P_n)$ of relations which satisfies (4.1) is called a solution of this set. In the general case (4.1) has more than one solution. The solution $(P_1,\ldots,P_n)$ is said to be the least solution if for any other solution $(Q_1,\ldots,Q_n)$ we have

$$P_i \subseteq Q_i \qquad \text{for } i = 1,\ldots,n.$$

It is a well-known fact that the least solution, if any, is unique. In our case we can prove the following:

Theorem 1. For any algorithm A the vector of relations $(\mathrm{Res}(\alpha_1,\alpha_1),\ldots,\mathrm{Res}(\alpha_1,\alpha_n))$ is the least solution of the corresponding CSE. □

The proof is in Sec.8. Here we shall show an effective method of solving (4.1) in such a way that we get the least solution whose components are expressed by R_{ij}'s and the operations of union, composition and iteration. The method consists in the application of two variable-elimination transformations:

1) Substitution: the substitution of $X_1R_{1i} \cup \ldots \cup X_nR_{ni} \cup R_{1i}$ for an arbitrary occurrence of X_i on the right side of (4.1).

2) Iteration: the replacement of the equation

$$X_i = X_1R_{1i} \cup \ldots \cup X_iR_{ii} \cup \ldots \cup X_nR_{ni} \cup R_{1i}$$

by the equation

$$X_i = (X_1R_{1i} \cup \ldots \cup X_{i-1}R_{i-1i} \cup X_{i+1}R_{i+1i} \cup \ldots \cup X_nR_{ni} \cup R_{1i})R^*_{ii}.$$

Each of these transformations is applicable to any set of equations like (4.1) and yields another set of equations of the same form. As can be proved (see [5] for the references) the new set of equations has exactly the same least solution as the former. To solve a given CSE we keep applying our transformations as long as there are some unknowns (variables) on the right side. This method of solving equations is very similar to the analogous method in the algebra of regular expressions. It must be emphasised, however, (see also Sec.7) that the algebra of relations is different from the algebra of regular expressions. Consequently, the equation-solving methods in one of them are not straightforward consequences of the analogous methods in the other.

The set of equations (4.1) can be written in a concise form as $\underline{X} = \Phi(\underline{X})$, where $\underline{X}$ ranges over the n-dimensional vectors of relation. Since solutions of such an equation are called in mathematics fixed points of Φ, the set (4.1) can be referred to as a fixed-point set of equations. It should be pointed out here that the fixed-point equations as used everywhere except this paper (see [1],[3-7],[9],[12] and also the references given there) are of a form symmetrical to (4.1), namely

$$X_i = R_{i1}X_1 \cup \ldots \cup R_{in-1}X_{n-1} \cup R_{in} \qquad \text{for } i = 1,\ldots,n-1.$$

The least solution is in such a case the vector $(\mathrm{Res}(\alpha_1,\alpha_n),\ldots,\mathrm{Res}(\alpha_{n-1},\alpha_n))$ which contains all the I-O relations corresponding to the output label α_n

with the input label varying from α_1 to α_{n-1}. In our case the situation is converse. All the I-O relation appearing in the least solution of (4.1) correspond to the same input α_1 with output varying from α_1 to α_n. This approach permits one to simplify calculations in dealing with programs which have more than one output. It also gives some advantages in proving local properties of programs, e.g. such that at a given label α_i a given Floyd's assertion is satisfied.

5. An example of a software-program verification

As mentioned in Sec.1 the method described in this paper has been tested on several arithmetical microprograms of a floating-point arithmetical unit designed at the Warsaw Technical University. One of these programs, call it Π_1, performed Booth's algorithm of the fixed-point multiplication of mantissas. We analysed Π_1 by introducing and verifying an abstract program Π_2 and by proving that Π_2 simulates Π_1 (see Sec.1). Here we shall investigate a simplified version of Π_2 which differs from it in neglecting overflows. In spite of this simplification the example still provides an adequate flavour of the method. In Sec.6 we show how to extend these calculations to deal with the real cases of Π_2 and Π_1.

In our program we shall deal with numbers from the interval $\langle -1,1)$ represented in the 2's complement code:

$$\alpha = -\alpha[0] + \sum_{i=1}^{n} \alpha[i]*2^{-i} \qquad (5.1)$$

where $\alpha[i] \in \{0,1\}$ for $i = 0,\ldots,n$ and $n \geq 1$ is fixed. We shall also use the equation

$$\alpha = \sum_{i=0}^{n} (\alpha[i+1]-\alpha[i])*2^{-i} \tag{5.2}$$

which follows from (5.1) under the condition that $\alpha[n+1] = 0$. The flowchart of our program is given in Fig.3. This program operates on the following variables:

1) the real variable a which ranges over arbitrary real numbers (this is actually the assumption which neglects the overflow),

2) the real variable d which ranges over $<-1,1)$,

3) the integer variable i which ranges over $\{0,\ldots,n+1\}$,

4) the 0's and 1's array q[0:n+1].

We shall prove that the program performs the multiplication $(-q[0]+\sum_{j=1}^{n} q[j]*2^{-j})*d$ and stores the result in a. In the calculations we shall use an extension of the notation introduced in Sec.2. Namely, for any vector of variables $(x_1,\ldots,x_n)$, the function

$$[(x_1,\ldots,x_n) := (f_1(x_1,\ldots,x_n),\ldots,f_n(x_1,\ldots,x_n))]$$

will be written as

$$\begin{bmatrix} x_1 := f_1(x_1,\ldots,x_n) \\ \ldots \\ x_n := f_n(x_1,\ldots,x_n) \end{bmatrix} . \tag{5.3}$$

Of course all the assignment statements in (5.3) are understood to be performed simultaneously. We assume also to omit in (5.3) all the assignment statements of the form $x_i := x_i$.

We shall prove the following about our program:

$$\mathrm{Res}(\alpha_1,\alpha_6) = \left[\begin{array}{l} a := (-q[0]+\sum_{j=1}^{n} q[i]*2^{-j})*d \\ i := 0 \\ q[n+1] := 0 \end{array}\right] \qquad (5.4)$$

According to the definition of $\mathrm{Res}(\alpha_i,\alpha_j)$ (see also the remarks by the end of Sec.2) this implies that our program terminates everywhere in its domain and that it performs the multiplication of the number whose representation is stored in q[0:n] by the number which is stored in d.

To simplify the calculations observe first that the program of Fig.3 can be reduced to the program of Fig.4 which has exactly the same $\mathrm{Res}(\alpha_1,\alpha_6)$ as the former one. Now compute (using (2.2)):

$$\begin{aligned}
&\mathrm{Res}(\alpha_2,\alpha_3) = [q[i+1] = q[i]] \cup \\
&\cup [q[i+1] \neq q[i]]([q[i+1] = 0 | a := a-d] \cup [q[i+1] = 1 | a := a+d]) = \\
&= [q[i+1] = q[i] \quad | a := a+(q[i+1]-q[i])*d] \cup \\
&\cup [q[i+1] = 0 \ \& \ q[i] = 1 | a := a+(q[i+1]-q[i])*d] \cup \\
&\cup [q[i+1] = 1 \ \& \ q[i] = 0 | a := a+(q[i+1]-q[i])*d] = \\
&= [a := a+(q[i+1]-q[i])*d] \ .
\end{aligned}$$

In the next step we establish the CSE of the program of Fig.4:

$$\begin{aligned}
X_1 &= \phi \\
X_2 &= X_5R_{52} \cup R_{12} \\
X_3 &= X_2\mathrm{Res}(\alpha_2,\alpha_3) \\
X_5 &= X_3R_{35} \\
X_6 &= X_3R_{36}
\end{aligned}$$

where

$$R_{52} = \begin{bmatrix} a := a*2^{-1} \\ i := i-1 \end{bmatrix}; \quad R_{12} = \begin{bmatrix} a := 0 \\ q[n+1] := 0 \\ i := n \end{bmatrix}$$

$$R_{35} = [i \neq 0] \quad ; \quad R_{36} = [i = 0]$$

Solving this set with respect to X_6 we get by a few substitutions (we omit the equations which we do not need any more to compute the solution for X_6):

$$X_3 = X_3R_{35}R_{52}Res(\alpha_2,\alpha_3) \cup R_{12}Res(\alpha_2,\alpha_3)$$
$$X_6 = X_3R_{36}$$

Therefore (by the iteration):

$$X_3 = R_{12}Res(\alpha_2,\alpha_3)(R_{35}R_{52}Res(\alpha_2,\alpha_3))^* =$$
$$= R_{12}(Res(\alpha_2,\alpha_3)R_{35}R_{52})^*Res(\alpha_2,\alpha_3)$$
$$X_6 = X_3R_{36}$$

and finally

$$X_6 = R_{12}(Res(\alpha_2,\alpha_3)R_{35}R_{52})^*Res(\alpha_2,\alpha_3)R_{36}. \tag{5.5}$$

By Theorem 1 we have, of course, $X_6 = Res(\alpha_1,\alpha_6)$. For $1 \leq k \leq n$ denote

$$S_k = \begin{bmatrix} a := \sum_{i=1}^{k} (q[n+1-k+i]-q[n-k+1])*2^{-i}*d \\ i := n-k \\ q[n+1] := 0 \end{bmatrix}$$

and for all $k \geq 1$ denote

$$P_k = R_{12}(Res(\alpha_2,\alpha_3)R_{35}R_{52})^k.$$

We shall prove by induction that for all $1 \le k \le n$ we have $P_k = S_k$.

For k = 1 we have immediately

$$P_1 = R_{12}\mathrm{Res}(\alpha_2,\alpha_3)R_{35}R_{52} =$$

$$= \begin{bmatrix} a := -q[n]*d*2^{-1} \\ i := n-1 \\ q[n+1] := 0 \end{bmatrix} = S_1$$

Let $P_k = S_k$ for some $k < n$. Then

$$P_{k+1} = P_k\mathrm{Res}(\alpha_2,\alpha_3)R_{35}R_{52} =$$

$$= S_k[a := a+(q[i+1]-q[i])*d][i \neq 0] \begin{bmatrix} a := a*2^{-1} \\ i := i-1 \end{bmatrix} =$$

$$= S_k[a := a+(q[n-k+1]-q[n-k]*d] \begin{bmatrix} a := a*2^{-1} \\ i := i-1 \end{bmatrix} =$$

$$= \begin{bmatrix} a := (\sum_{i=1}^{k}(q[n+1-k+i]-q[n-k+i])*2^{-i}*d+(q[n-k+1]-q[n-k])*d)*2^{-1} \\ i := n-k-1 \\ q[n+1] := 0 \end{bmatrix} =$$

$$= \begin{bmatrix} a := (\sum_{i=2}^{k+1}(q[n+1-(k+1)+i]-q[n-(k+1)+i])*2^{-i}*d+(q[n+1-(k+1)+1]- \\ \qquad -q[n-(k+1)+1])*2^{-1}*d \\ i := n-(k+1) \\ q[n+1] := 0 \end{bmatrix} =$$

$$= \begin{bmatrix} a := \sum_{i=1}^{k+1}(q[n+1-(k+1)+i]-q[n-(k+1)+i])*2^{-i}*d \\ i := n-(k+1) \\ q[n+1] := 0 \end{bmatrix} = S_{k+1}$$

This terminates the proof of the equality $P_k = S_k$ for $1 \leq k \leq n$.

As is easy to see $P_n = P_n[i = 0]$, hence

$P_{n+1} = P_n[i=0]Res(\alpha_2,\alpha_3)[i\neq 0]R_{52} = \phi$. Therefore,

$$P_m = \phi \quad \text{for all } m \geq n+1.$$

By a similar argument we can prove that for $0 \leq k < n$

$$P_k Res(\alpha_2,\alpha_3)[i=0] = \phi$$

Both propositions prove that the loop in our program must be performed exactly n times, which formally means (c.f.(5.5)) that

$$Res(\alpha_1,\alpha_6) = S_n Res(\alpha_2,\alpha_3)[i=0]$$

By straightforward calculations we get now

$$Res(\alpha_1,\alpha_6) = \left[\begin{array}{l} a := \sum_{i=0}^{n} (q[i+1]-q[i])*2^{-i}*d \\ i := 0 \\ \\ q[n+1] := 0 \end{array}\right]$$

which by (5.2) gives us the required equation (5.4). This equation provides the complete description of the I-O relation of our program and therefore terminates the proof of the total correctness of the program.

It is worth observing that using our method one can also prove some local properties of the program. For instance, by similar calculations as above, one can easily show

$$Res(\alpha_1,\alpha_2) = \bigcup_{k=0}^{n} S_k$$

and

$$Res(\alpha_1,\alpha_3) = \bigcup_{k=0}^{n} S_k Res(\alpha_2,\alpha_3)$$

where we let $S_0 = R_{12}$. Using these equations we can prove the equations

$$Res(\alpha_1,\alpha_2) = Res(\alpha_1,\alpha_2)[-1 \leq a < 1]$$

and

$$Res(\alpha_1,\alpha_3) = Res(\alpha_1,\alpha_2)[-2 \leq a < 2]$$

This gives us the estimation of the current value of a at the label α_2 or α_3 respectively. This estimation shows that the overload problem, which appears in the real program, is not too cumbersome and can be solved easily by well-known tricks.

6. An example of a microprogram verification

The program investigated in Sec.5 - call it Π_3 - was a simplification of the abstract program Π_2 which simulated the real microprogram Π_1. The program Π_2 differs from Π_3 in the following:

1) n = 23 and d ranges over numbers representable by (5.1)

2) the assignments a := a±d are replaced respectively by:

   a := a±d - 2*Ovf(a±d);

   z := |Ovf(a±d)|;

where

$$Ovf(x) = \begin{cases} -1 & ; x < -1 \\ 0 & ; -1 \le x < 1 \\ 1 & ; 1 \le x \end{cases}$$

3) the assignment $a := a*2^{-1}$ is replaced by:

$\underline{if}\ z = 0\ \underline{then}\ a := a*2^{-1}\ \underline{else}\ a := a*2^{-1} + sgn(a);$

$z := 0;$

where

$$sgn(a) = \begin{cases} 1 & ; a < 0 \\ 0 & ; a \ge 0 \end{cases}$$

Following the same way as in Sec.5 - just with more calculations - we can prove the following:

$$Res_2(\alpha_1,\alpha_6) = \left[\begin{array}{l} a := q*d - 2*Ovf(q*d) \\ i := 0 \\ q[24] := 0 \\ z := Ovf(q*d) \end{array}\right] \qquad (6.1)$$

where $Res_2(\alpha_1,\alpha_6)$ is the appropriate I-O relation in Π_2 and $q = \sum_{i=0}^{n} (q[i+1]-q[i])*2^{-1}$.

To get the original microprogram Π_1 every occurrence of the expression $a*2^{-1}$ in Π_2 must be replaced by $a \otimes 2^{-1}$, which denotes the machine multiplication of a by 2^{-1}. This machine multiplication is effectuated by the arithmetical shift right of the register storing the representation of a. Since the length of this register is restricted to 24 bits the "arithmetical effect" of this shift can be described by the following equation:

$$a \otimes 2^{-1} = a*2^{-1} - a[23]*2^{-24}.$$

Of course, we could use this equation to replace every occurrence of $a \otimes 2^{-1}$ in Π_1 by its right side and to get an equivalent program Π_1' with only arithmetical operations. The analysis of such a program, however, would be very cumbersome. It is much easier to observe that Π_2 simulates Π_1 and to verify Π_2. Let us describe briefly what the simulation of Π_1 by Π_2 looks like.

In the original case of our analysis we proceed, of course from Π_1 to Π_2. First we establish the algorithm $A_1 = (D_1, V, \alpha_1, \mathfrak{J}_1)$ which corresponds to Π_1. The domain D_1 is the set of all the vectors of the form $(a,d,q[0],\ldots,q[24],i,z)$ where a and d range over these numbers in $<-1,1)$ which can be represented by (5.1) with n = 23, and the other variables have the ranges as described earlier. Next we establish the algorithm $A_2 = (D_2, V, \alpha_1, \mathfrak{J}_2)$ which corresponds to Π_2 and which differs from A_1 only in D_2 and $\mathfrak{J}_2$. The set D_2 results from D_1 by letting a to range over all numbers in $<-1,1)$. The set $\mathfrak{J}_2$ results from $\mathfrak{J}_1$ by replacing every instruction $(\alpha_i, R_{ij}, \alpha_j)$ by the instruction $(\alpha_i, \hat{R}_{ij}, \alpha_j)$, where $\hat{R}_{ij}$ results from R_{ij} - informally speaking - by replacing all $a \otimes 2^{-1}$ by $a*2^{-1}$. Now, for any number $\alpha \in <-1,1)$ whose standard 2's complement representation is

$$\alpha = -\alpha[0] + \sum_{i=1}^{\infty} \alpha[i]*2^{-i}$$

let

$$t(\alpha) = -\alpha[0] + \sum_{i=1}^{24} \alpha[i]*2^{-i}.$$

$$\mathrm{Res}_1(\alpha_1,\alpha_6) = \begin{bmatrix} a := t(q*d)-2*0vf(q*d) \\ b := 0 \\ q[24] := 0 \\ z := 0vf(q*d) \end{bmatrix} \qquad (6.3)$$

This equation says that the original microprogram Π_1 always terminates ((6.3) implies that $\mathrm{Res}_1(\alpha_1,\alpha_6)$ is a total function) and that it produces the 24-bit representation of the required product. This representation can happen to be modified by an overflow in which case the value of z will become 1.

7. Final remarks

The program-verification method presented in this paper is frequently confused with the approach by regular expressions (or regular events). The confusion is due to the fact that the notation of the algebra of relations (the symbols "$\cup$", "$\circ$" and "*") is very similar to that of the regular expressions. Also the heuristic methods of solving equations in both algebras are the same. It is to be emphasized, however, that the two approaches are essentially different. Using the algebra of relations one can investigate the I-O properties of programs such as partial or total correctness. Using the algebra of regular expressions one can only deal with the language-theoretic properties of sets $\mathrm{Exec}(\alpha_i,\alpha_j)$ rather than with the properties of relations $\mathrm{Res}(\alpha_i,\alpha_j)$.

As mentioned in Sec.1 our method was applied to several "practical" microprograms. It proved to be feasible in spite of the fact that all the calculations were handled manually. The calculations also showed that a considerable part of them (e.g. solving equations and simplifying formulas) could be performed by simple symbol-manipulation programs.

Another feature of the method is that it permits the structuring of analyzed programs. In such a case the modules of the program are analyzed as independent algorithms whose resulting relations became the actions of the algorithm of the next level (cf. programs of Fig.3 and Fig.4). The number of these levels can be, of course, arbitrary and the method can be reapplied at each level in the same way. It is worth mentioning that we admit one-input many-outputs modules which makes our structuring much easier than in the one-input one-output case.

The program analyzed in Sec.5 had the property that its loop was iterated always the same number of times. We should stress that our method applies in exactly the same way to programs where the number of iterations depends on the input data.

The size of the present paper did not permit, of course, to show a "practical" case of a microprogram verification. We wish to inform the reader that a full example of such a verification will appear as a technical report.

8. Appendix: the proof of Theorem 1

Let us start this proof by introducing a few auxiliary mathematical concepts. Consider an arbitrary nonempty set D and the set Rel(D) of binary relations in it. We shall deal with finite sequences of relations $(R_1,\ldots,R_n)$ and the set of all these sequences will be denoted by seq(Rel(D)). Note that this set contains also the empty sequence $\varepsilon = (\ )$. We introduce the operation $C : seq(Rel(D)) \to Rel(D)$ defined as follows:

$$C[\varepsilon] = I, \qquad C[(R)] = R$$

$$C[(R_1,\ldots,R_n)] = R_1 \circ \ldots \circ R_n \text{ for any nonempty sequence } (R_1,\ldots,R_n).$$

We shall also deal with sets of sequences of relations which are subsets of the set seq(Rel(D)) and which can be considered as languages over the "alphabet" Rel(D). The family of all these languages will be denoted by Lan(Rel(D)). We shall introduce the operation $\mathbf{C} : Lan(Rel(D)) \to Rel$ defined as follows:

$$\mathbf{C}(L) = \cup\{C[t] \mid t \in L\} \quad \text{for any } L \subseteq seq(Rel(D)).$$

The following properties of this operation will be used in our proof:

$$1) \qquad \mathbf{C}(\phi) = \phi$$

$$2) \qquad \mathbf{C}\left(\bigcup_{i=1}^{\infty} L_i\right) = \bigcup_{i=1}^{\infty} \mathbf{C}(L_i) \qquad (8.1)$$

$$3) \qquad \mathbf{C}(L_1\, L_2) = \mathbf{C}(L_1) \circledast \mathbf{C}(L_2)$$

where ϕ denotes the empty language and the empty relation at the same time, L_1 and L_2 are arbitrary subsets of seq(Rel(D)) and " $\circ$ " denotes the usual concatenation of languages.

Now we are ready to proceed to the proof. Let $A = (D,V,\alpha_1,\mathcal{T})$ be an arbitrary algorithm. It is an obvious task to see that the sets $Exec(\alpha_1,\alpha_i)$ satisfy the following equations:

$$Exec(\alpha_1,\alpha_i) = \bigcup_{j=1}^{n} Exec(\alpha_1,\alpha_j)^\frown\{(R_{ji})\} \cup \{(R_{1i})\} \; ; \; i = 1,\ldots,n \qquad (8.2)$$

It is also obvious that for all $i \leq n$

$$Res(\alpha_1,\alpha_i) = \mathbf{C}(Exec(\alpha_1,\alpha_i))$$

Applying the operation $\mathbf{C}$ to both sides of the equations (8.2) we get

$$Res(\alpha_1,\alpha_i) = \bigcup_{j=1}^{n} Res(\alpha_1,\alpha_j)R_{ji} \cup R_{1i}; \quad i = 1,\ldots,n$$

which proves that the vector of $Res(\alpha_1,\alpha_i)$ satisfies the CSE of A. Now we shall prove that this vector is the least solution of CSE. Let $(Q_1,\ldots,Q_n)$ be an arbitrary solution of CSE. We shall show that $Res(\alpha_1,\alpha_i) \subseteq Q_i$ for $i = 1,\ldots,n$. For any $i \leq n$ and any $m \geq 1$, let $Exec^m(\alpha_1,\alpha_i)$ denote the set of all (α_1,α_i)-s. executions of length not greater than m. Clearly, for any $i \leq n$

$$Exec(\alpha_1,\alpha_i) = \bigcup_{m=1}^{\infty} Exec^m(\alpha_1,\alpha_i).$$

Let $Res^m(\alpha_1,\alpha_i) = \mathbf{C}(Exec^m(\alpha_1,\alpha_i))$. By (811)

$$Res(\alpha_1,\alpha_i) = \bigcup_{m=1}^{\infty} Res^m(\alpha_1,\alpha_i)$$

for $i \leq n$. We shall show that $Res^m(\alpha_1,\alpha_i) \subseteq Q_i$ for $i \leq n$ and $m \geq 1$. The proof is by induction on m.

Initial step: for all $i \leq n$

$$Res^1(\alpha_1,\alpha_i) = R_{1i} \subseteq \bigcup_{j=1}^{n} Q_j R_{ji} \cup R_{1i} = Q_i$$

Induction step: Let $Res^m(\alpha_1,\alpha_i) \subseteq Q_i$ for $i \leq n$.

Of course

$$Exec^{m+1}(\alpha_1,\alpha_i) = \bigcup_{j=1}^{n} Exec^m(\alpha_1,\alpha_j)\hat{}\{(R_{ji})\} \cup \{(R_{1i})\}.$$

for all $i \leq n$. Therefore, by applying **C**,

$$Res^{m+1}(\alpha_1,\alpha_i) = \bigcup_{j=1}^{n} Res^m(\alpha_1,\alpha_j)R_{ji} \cup R_{1i} \subseteq$$

$$\subseteq \bigcup_{j=1}^{n} Q_j R_{ji} \cup R_{ii} = Q_i. \qquad \square$$

References

[1] J.W. de Bakker and W.P. de Roever, "A calculus for recursive program schemes", in Automata Languages and Programming (M. Nivat, ed.) pp.167-196, North Holland, Amsterdam, 1973.

[2] A. Birman, "On proving correctness of microprograms", IBM J. Res. Develop., vol.18, pp.250-266, May 1974.

[3] A. Blikle, "Iterative systems; an algebraic approach", Bull. Acad. Polon. Sci., Ser. Sci. Math. Astronom. Phys., pp.51-55, vol.20, 1972.

[4] A. Blikle, "Complex iterative systems". ibid, pp.57-61, vol.20, 1972.

[5] A. Blikle, "Proving programs by δ-relations", in Formalization of Semantics of Programming Languages and Writing of Compilers (Proc. Symp. Frankfurt am Oder 1974), Electronische Informationsverarbeitung und Kybernetik (to appear in 1976).

[6] A. Blikle, and A. Budkowski, "A general program-verification method applied to microprograms" a short paper in Proc. of The 1976 Int. Conf. on Fault-Tolerant Computing (FTCS-6), Pittsburg, 1976.

[7] A. Blikle and A. Mazurkiewicz, "An algebraic approach to the theory of programs, algorithms, languages and recursiveness" in Mathematical Foundations of Computer Science (Proc. Symp. Warsaw-Jablonna 1972) Warsaw, 1972.

[8] W.C. Carter, W.H. Joyner and G.B. Leeman, "Automated experiments in validating microprograms", Digest of Papers of the 1975 Int. Conf. on Fault-Tolerant Computing (FTCS-5), p.247, Paris, 1975.

[9] T. Ito, "A theory of formal microprograms" in Proc. of The Int. Adv. Sum. Inst. on Microprogramming, 1971.

[10] G.B. Leeman, "Some problems in certifying microprograms", IEEE Trans. on Comp., vol.C-24, No.5, pp.545-553, May 1975.

[11] G.B. Leeman, W.C. Carter and A. Birman, "Some techniques for micrpprogram validation", Inf. Processing 74, pp.76-80, North Holland 1974.

[12] Z. Manna, "Mathematical Theory of Computation", McGraw-Hill Book Co., New York, 1974.

[13] A. Mazurkiewicz, "Proving algorithms by tail functions", Inf. Cont., vol.18, pp.220-226, 1971.

[14] R. Milner, "An algebraic definition of simulation between programs", 2nd Int. Joint Conf. Artificial Intelligence, London 1971, pp.481-489.

[15] D. Paterson, "The design of a system for the synthesis of correct micro-programs", Micro-8 Proc., Eight Annual Workshop on Microprogramming, Chicago, Sept. 1975.

Figures

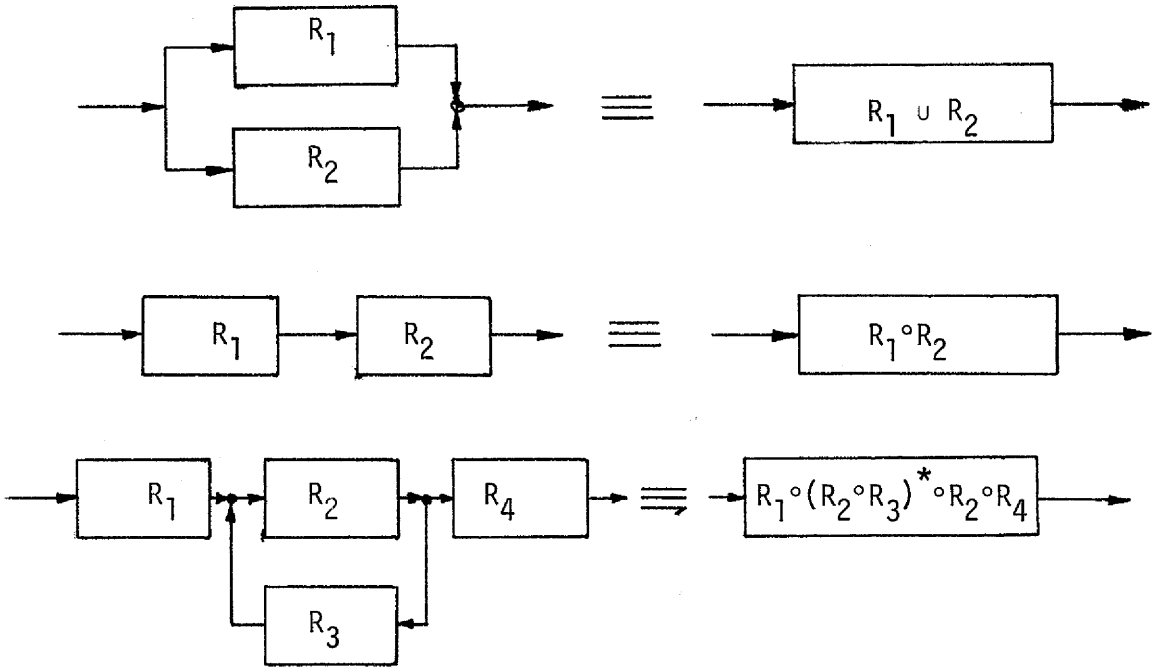

Fig. 1

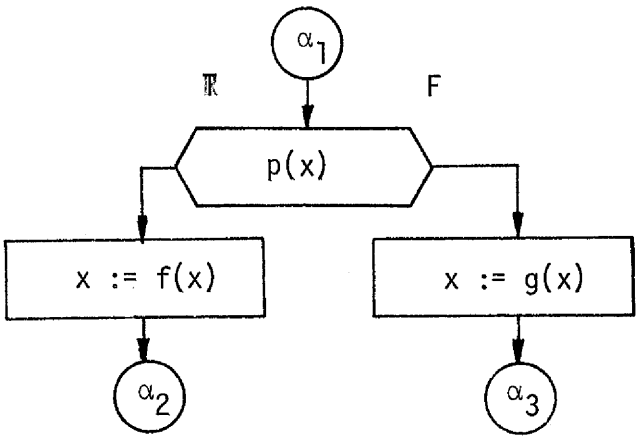

Fig. 2

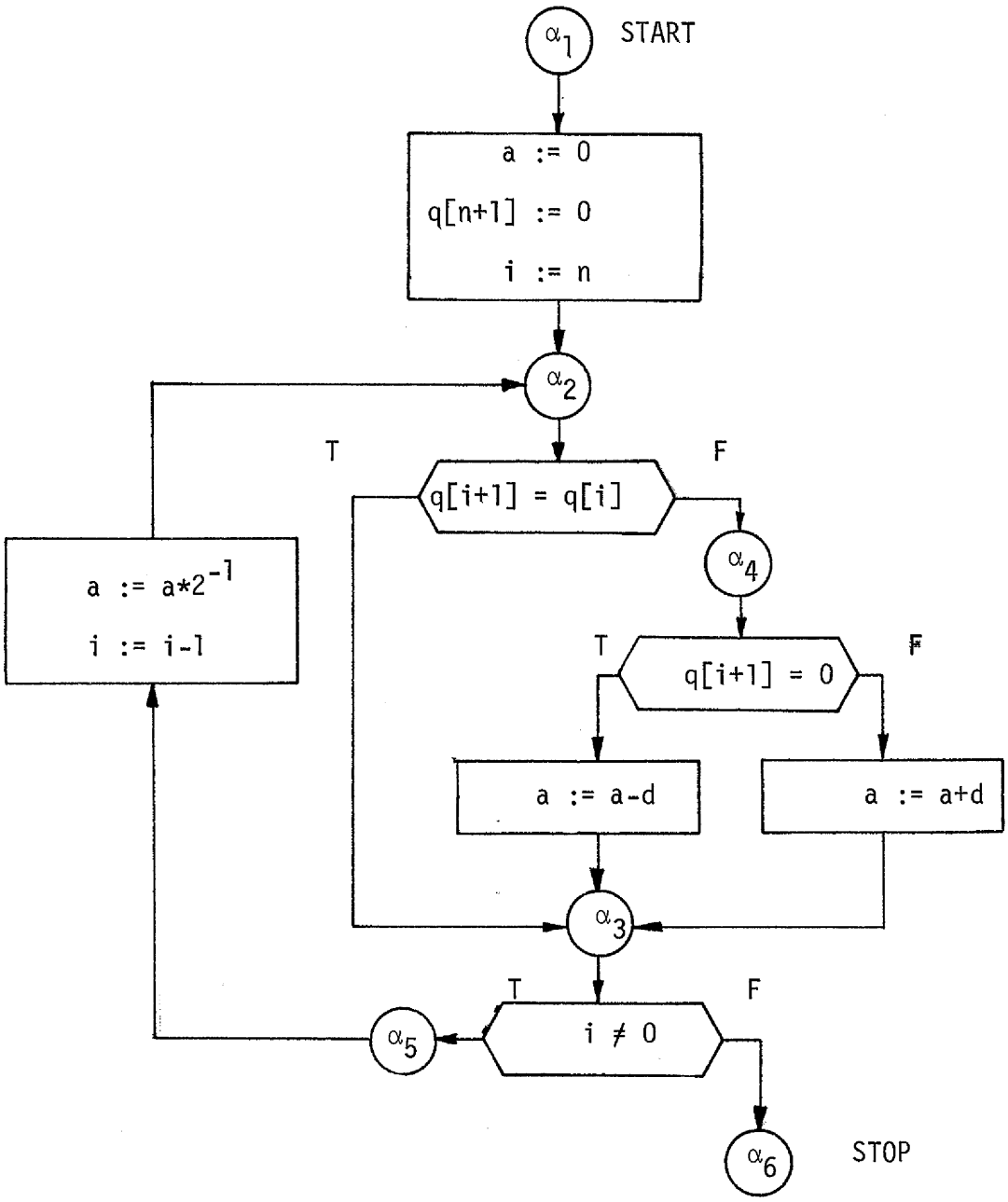
α_1
START
a := 0
q[n+1] := 0
i := n
α_2
T
F
q[i+1] = q[i]
α_4
a := a*2^-1
i := i-1
T
F
q[i+1] = 0
a := a-d
a := a+d
α_3
T
F
α_5
i ≠ 0
α_6
STOP

Fig. 3

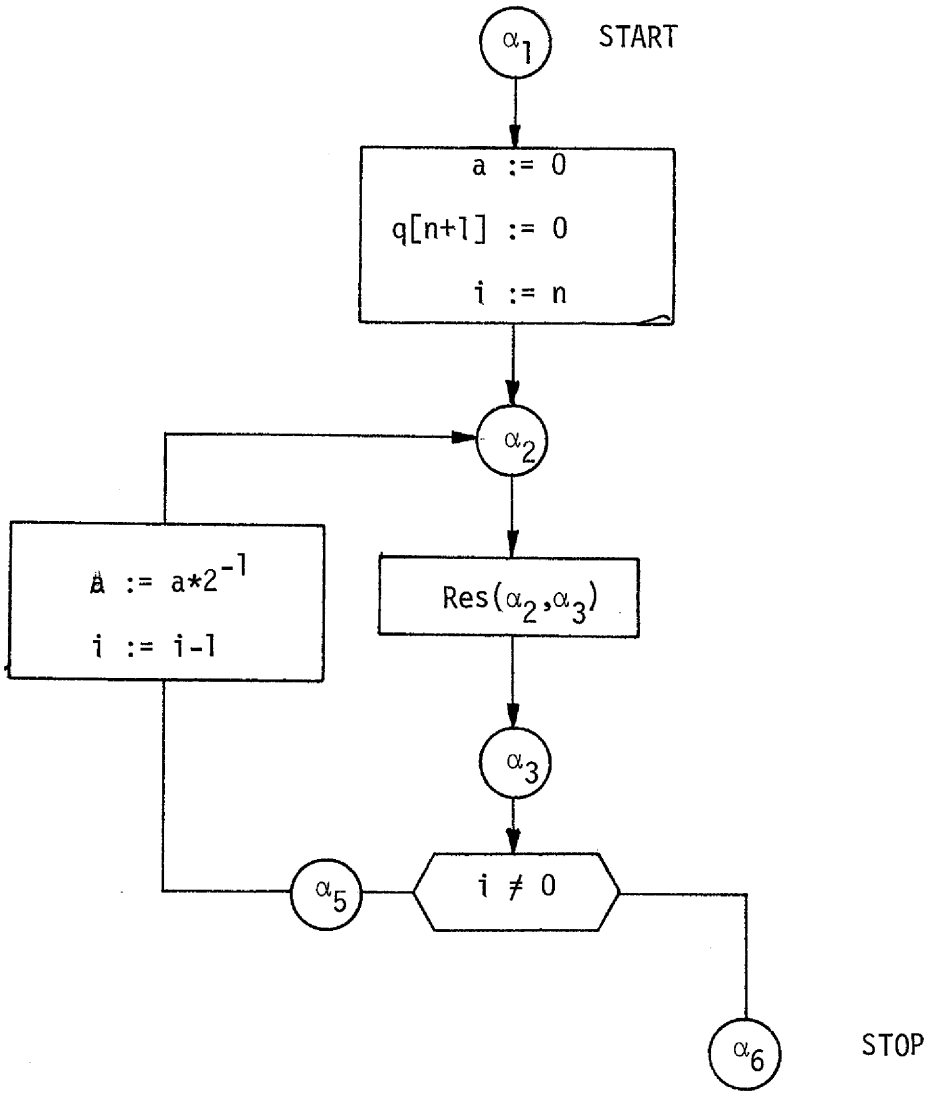
α_1
START
a := 0
q[n+1] := 0
i := n
α_2
a := a*2^-1
i := i-1
Res(α_2,α_3)
α_3
α_5
i ≠ 0
α_6
STOP

Fig. 4