Technical Report CS75022-R

# An Extension of the Direct Method for Verifying Programs

by

Andy N. C. Kang
Department of Computer Science
V. P. I. & S. U.
Blacksburg, Virginia 24061

Shih-Ho Wang
Department of Electrical Engineering
University of Colorado
Colorado Springs, Colorado 80907

October 1975

Abstract

A direct method based on a finite set of path formulas which describe the input-output relations of a given program can be used to verify programs containing no overlapping loops. One major difficulty in verifying programs with overlapping loops using the above method is that too many path formulas (possibly infinite) needed to be considered. In this paper, we circumvent the above difficulty by applying the concept of modularity. The idea is to divide a program with overlapping loops into several small modules so that each module contains no overlapping loop. This can always be achieved if the program is in structured form. Then the path formulas will be derived for each module. By combining the path formulas for the modules, one can further obtain the path formulas for the given program and then use them to verify the program.

## I. Introduction

In [1] a direct method is given to verify programs containing no overlapping loops. Basically, the idea is to find a finite set of logical formulas called "path formulas" which describe the input-output relations of a given program. Then one tries to verify the program by deducing the desired input-output specifications from the above set of path formulas. This method is direct and intuitive and avoids the difficult process of finding the inductive assertions which are commonly used for verifying programs. However, only programs without overlapping loops are considered in [1] . As suggested by the authors, their method might be extended to prove the correctness of programs with overlapping loops. To verify programs with overlapping loops using their procedure one major difficulty is that too many path formulas (possibly infinite) needed to be considered. In the present paper, we circumvent the above difficulty by applying the concept of modularity. The idea is to divide a program with overlapping loops into several small modules. Each module contains no overlapping loops, hence the path formulas for each module can be derived. By combining the path formulas for each module, one can further obtain the path formulas for the given program and then use them to verify the program.

In the next section, some definitions and preliminary results are given. In section III, an example is used to illustrate our method.

## II. Preliminary Developments

We begin with the following definitions.

Definition. A program consists of an input vector $\bar{x} = (x_1,\ldots,x_L)$, a program vector $\bar{y} = (y_1,\ldots,y_M)$, an output vector $\bar{z} = (z_1,\ldots,z_N)$, and a finite directed

graph (V, A) such that the following conditions are satisfied:

1. In the graph (V, A), there is exactly one vertex, called the <u>starting vertex</u> $S \in V$, that is not a terminal vertex of any arc; there is exactly one vertex, called the <u>halting vertex</u> $H \in V$, that is not an initial vertex of any arc; and every vertex v is on some path from S to H.
2. In (V, A), each arc a is associated with a quantifer-free formula $P_a(\bar{x}, \bar{y})$ called <u>testing predicate</u> and an <u>assignment</u> $\bar{y} \leftarrow f_a(\bar{x}, \bar{y})$.
3. For each vertex v, let $a_1,\ldots,a_r$ be all arcs leaving v and let $P_{a_1},\ldots,P_{a_r}$ be the testing predicates associated with arcs $a_1,\ldots,a_r$ respectively. Then for all $\bar{x}$ and $\bar{y}$, one and only one $P_{a_1}(\bar{x}, \bar{y}),\ldots,P_{a_r}(\bar{x}, \bar{y})$ is true.

To analyze a program, we associate every vertex v in the program with an <u>access predicate</u> $Q_v(\bar{x}, \bar{y})$ (or $Q_v(\bar{x}, \bar{z})$ if v=H), which represents the condition that control is passed from vertex S to vertex v with the values of the input and program vectors being $\bar{x}$ and $\bar{y}$ respectively ( or the values of the input and the output vectors being $\bar{x}$ and $\bar{z}$, respectively). Using access predicates, we can associate a path from vertex v to vertex u with a logical formula. For example, if a path consists of only one arc a with v and u being the initial and terminal vertices of arc a, then a formula $W_a$ associated with path a is as follows:

$$W_a: \quad Q_v(\bar{x}, \bar{y}) \ \& \ P_a(\bar{x}, \bar{y}) \rightarrow Q_u(\bar{x}, f_a(\bar{x}, \bar{y})).$$

This formula indicates that if control is at the initial vertex v of the path with the values of the input and the program vectors being $\bar{x}$ and $\bar{y}$, respectively, i.e., $Q_v(\bar{x}, \bar{y})$ is true, and if $P_a(\bar{x}, \bar{y})$ is also true, then control will pass along to the terminal vertex u of the path, with the value of the program vector changed to $f_a(\bar{x}, \bar{y})$, i.e., $Q_u(\bar{x}, f_a(\bar{x}, \bar{y}))$ is true. In the subsequent section, we shall first find

all paths for S to H. Then we shall associate a path (from S to H) with a logical formula (called path formula) for describing control passing along the path. Once path formulas for a program are obtained, they can be used to prove the program correctness.

## III. Main Result

In a flowgraph, we say two loops are overlapping if they share some common arcs. In [1] only programs having no overlapping loop are considered. In this section, we give an example to illustrate how to derive the path formulas for programs with overlapping loops. This example is taken from [3] with modifications.

Example:

Consider the program shown in Figure 1, where x is a non-negative integer. This program calculates the sum of $\sqrt{1}$, $\sqrt{2}$..., $\sqrt{x}$, where $\sqrt{i}$ is defined as an integer k such that $k^2 \leq i < (k+1)^2$. Prove that this program is correct, i.e., $z = \sqrt{1} + \sqrt{2} + \ldots + \sqrt{x}$.

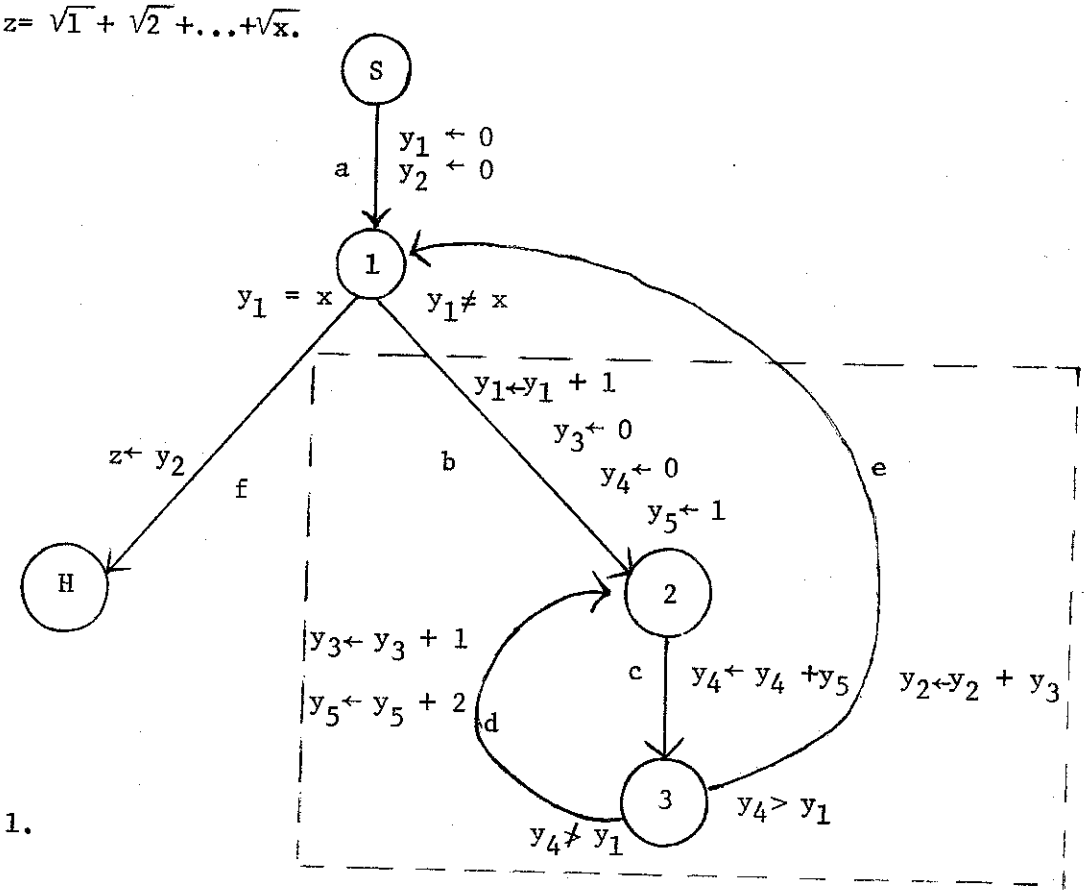


Figure 1.

The regular expression [4] associated with the flowgraph is a(bc(dc)*e)*f. This means that for an input x, if the program is going to terminate, the control has to take the path from vertex S along arc a to vertex 1, then take the path "bc(dc)*e" a number of times (possibly 0) and back to vertex 1, and finally take arc f to vertex H. Each time the control takes the path "bc(dc)*e", it has to take path bc, then takes path dc several times (possibly 0) and finally takes path e.

The flowgraph of Figure 1 has an overlapping loop. Let us first consider the portion of the flowgraph within the dotted line and redraw the flowgraph as follows:

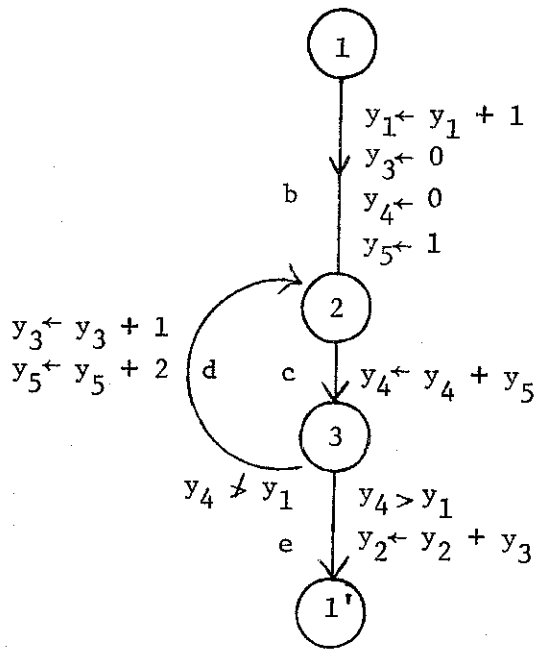


Figure 2.

Now this flowgraph does not contain overlapping loops. Thus, we can apply the procedure in [1] and derive the following path formula:

$(1^2 \not> (y_1+1)) \& \ldots \& (n^2 \not> (y_1+1)) \& ((n+1)^2 > (y_1+1)) \rightarrow Q_1(y_1+1, y_2+n, n, (n+1)^2, 1+2n)$.

This formula means that if the control begins at vertex 1 with the program vector $(y_1,y_2,y_3,y_4,y_5)$ and takes the path $bc(dc)^n e$ and arrives at vertex 1', then the program vector will become $(y_1+1,\ y_2+n, n,\ (n+1)^2, 1+2n)$. This will

happen if the conditions $(n+1)^2 > (y_1+1)$, $n^2 \not> (y_1\ 1), \ldots, 1^2 \not> (y_1+1)$ are satisfied.

Now we can simplify the original flowgraph into the following one, where $g_n$ denotes the path $bc(dc)^n e$ and n denotes the number of times the control has taken path dc.

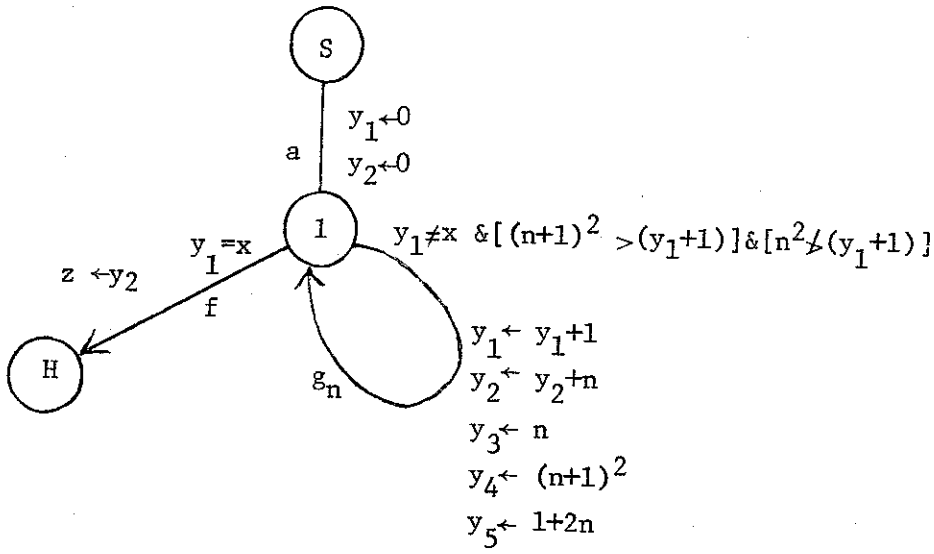


Figure 3.

Notice that the new graph does not contain any overlapping loop. So we can follow the procedure described in [1] to derive path formulas for the following path types:

(i) Path type af

$$W_{af}: x=0 \rightarrow Q_H(x,0) \qquad (1)$$

(ii) Path type $ag_{n_1}g_{n_2}\cdots g_{n_m}f$

$$\begin{aligned} Wag_{n_1}g_{n_2}\cdots g_{n_m}f: \ & 0 \neq x \ \& \ (n_1+1)^2 > 1 \ \& \ n_1^2 \not> 1 \\ & \& \ 1 \neq x \ \& \ (n_2+1)^2 > 2 \ \& \ n_2^2 \not> 2 \\ & \& \ \ldots \\ & \& \ m-1 \neq x \ \& \ (n_m+1)^2 > m \ \& \ n_m^2 \not> m \\ & \& \ m=x \longrightarrow Q_H(x,\ n_1+n_2+\ldots+n_m). \end{aligned} \qquad (2)$$

Using these path formulas, we can prove the program correct. Since x is zero or a positive integer, we consider the following two cases:

Case 1. $x=0$

Using (1), we obtain $Q_H(x,0)$. This implies $z=0$.

Case 2. $x > 0$

Since $x > 0$, we can find integers $m$, $n_1$, $n_2$,...,$n_m$ such that $m=x$, $m-1\neq x$, ..., $0\neq x$, and $n_1=1$ (i.e. $(n_1+1)^2>1$ & $n_1^2 \not=1$),

$n_2=1$ (i.e. $(n_2+1)^2>2$ & $n_2^2 \not=2$),

...

$n_m= \max\ell[\ell^2 \not=m]$ (i.e. $(n_m+1)^2 \not> m$ & $n_m^2 \not> m$).

Applying (2) we obtain $Q_H(x, n_1+n_2+...+n_x)$. This means that $z=n_1+n_2+...+n_x$ $=\sqrt{1} + \sqrt{2} + ...+ \sqrt{x}$.

Q.E.D.

## IV. Conclusions

For some programs with overlapping loops, it is possible to write down the path formulas as described in [1] without dividing the program into several modules. For instance, in our previous example, one can write down the following path formula: $a(bc(dc)^{n_1}e)(bc(dc)^{n_2}e)...(bc(dc)^{n_m}e)f$. However, as those path formulas are usually very complicated, it is rather difficult to deduce the desired program specifications from them.

In this paper we have considered the problem of verifying programs by using logical formulas. Our method extends that in [1] with several important features. Based on the concept of modularity, our method can be used to verify programs with overlapping loops. Another definite advantage of dividing programs into several modules is that one can easily locate the error of a program in case there is any.

It is obvious that the concept of modularity should play an important role in verifying programs. However, it is still not clear how to divide a general program into several modules in a most convenient way. Much future research effort should be led to this direction.

References

1. C. L. Chang, R. C. T. Lee and J. R. Slagle (1974): "A Direct Method for Verifying Programs", submitted for publication.

2. C. L. Chang and R. C. T. Lee (1973): Symbolic Logic and Mechanical Theorem Proving, Academic Press.

3. S. M. Katz and Z. Manna (1973): "A Heuristic Approach to Program Verification", Proc. of Third International Joint Conference on Artificial Intelligence, Stanford University.

4. M. Minsky (1967): Computation, Finite and Infinite Machines, Prentice-Hall.