This paper appeared in IEEE Trans. on Software Engineering, vol. SE-5, no. 3, May 1979, pp. 226-236.

Detection of Data Flow Anomaly Through Program Instrumentation

J. C. HUANG, MEMBER, IEEE

1. INTRODUCTION

It appears that there are two basic ways to increase the power of a program test. One is to find better criteria for test-case selection. The other is to find a test scheme that will produce additional information (i.e. information other than the output of the program under test) that can be used for error detection.

Fosdick and Osterweil [1] have shown that information concerning the creation and use of data definitions in a program can be used for error-detection purposes. Such information can be obtained by performing a data flow analysis. All known data flow analysis methods (see,e.g.,[1]-[8]) are designed to carry out the analysis by systematically scanning the text of the program in question. This paper describes a method for obtaining the desired information by means of program instrumentation. By program instrumentation here we mean the process of inserting additional statements into a program for information gathering purposes. The desired information is to be obtained by executing the instrumented program for a properly chosen set of input data. The significance of this approach is that we can increase the power of a program test simply by instrumenting the program to be tested for data flow anomaly detection as described in the following sections.

We begin by presenting the main idea in Section II. One important advantage of the present method is that array elements can be handled individually. We explain how this can be accomplished in Section III. In Section IV we present a criterion for determining whether a set of test cases is sufficient to reveal all possible data flow anomalies. The problem of variable aliasing and multiple use of a name are discussed in Section V. In Section VI we explain what needs to be done if a subprogram is not available for instrumentation for some reason and discuss how to detect data flow anomalies in a program by applying the method to its subprograms individually. Additional comments about the method

and a comparison with existent methods are given in Section VII. The presentation is focused on basic problems associated with the instrumentation method -- which in principle can be applied to any program written in a procedural language -- as well as special problems that may arise when the method is to be applied to Fortran programs.

2. DETECTION OF DATA FLOW ANOMALIES

It is observed that, in program execution, a statement may act on a variable (datum) in three different ways, viz., define, reference, and undefine. A variable is defined in a statement if an execution of the statement assigns a value to that variable. A variable is referenced in a statement if an execution of the statement requires that the value of that variable be obtained from memory. Thus in the assignment statement

```
x := x + y - z
```

y and z are both referenced while x is first referenced and then defined. A variable may become undefined in many circumstances. For example, in a Fortran program, the index variable of a DO statement becomes undefined when the loop is terminated, and the local variables of a subprogram become undefined when the RETURN statement is executed. Also, if a program is written in a language that allows block structure, the local variables of a block may become undefined when the control exits from the block.

A sequence of actions may be taken on a variable in a program while it is being executed. A reference to a variable constitutes a programming error unless the value of the variable is defined previously. Furthermore, there is no need to define a variable unless it is to be referenced (i.e., its value to be used) later. Therefore, if we find that a variable in a program is (1) undefined and then referenced, (2) defined and then undefined, or (3) defined and then defined again, then we may reasonably conclude that a programming error might have been committed. This idea has been utilized by Fosdick and Osterweil [1] to detect programming errors.

A method for detecting the three types of data flow anomalies mentioned above has been developed by Fosdick and Osterweil [1]. The basic idea is to compute the so-called path expressions of paths in a flow graph by making use of data flow analysis algorithms developed in connection with program optimization [1-8]. A path expression describes the sequence of actions taken on a variable when the program is executed along the path. The presence of data flow anomalies can thus be detected by examining the constituent components of path expressions.

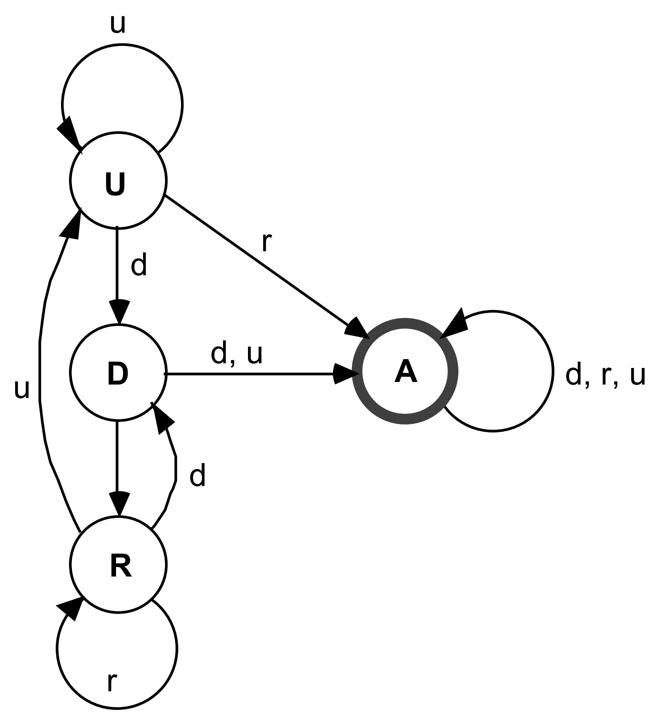


Fig.1. Program variables may assume one of the four states of this state diagram (U, undefined; D, defined but not referenced; R, referenced; A, abnormal).

In the following we shall present a new method for detecting data flow anomalies by means of program instrumentation. For this purpose, it is useful to regard a variable as being in one of four possible states during program execution. The four possible states are state U: undefined, state D: defined but not referenced, state R: defined and referenced, and state A: abnormal state. For error-detection purposes it is proper to assume that a variable is in the state of being undefined when it is declared implicitly or explicitly. Now if the action taken on this variable is "define," then it will enter the state of being defined but not referenced. Then, depending on the next action taken on this variable, it will assume a different state as shown in Fig. 1. Note that each edge in this state diagram is associated with d, r, or u, which stand for "define," "reference," and "undefine," respectively. The three types of data flow anomalies mentioned previously can thus be denoted by ur, du, and dd in this shorthand notation. It is easy to verify that, if a sequence of actions taken on the variable contains either ur, du, or dd as a subsequence, the variable will enter state A, which indicates the presence of a data flow anomaly in the execution path. We let the variable remain in state A once that state is entered. Its implication and possible alternatives will be discussed in Section VII.

It is obvious from the above discussion that there is no need to compute the sequence of actions taken on a variable along the entire execution path. Instead, we need only to know if the sequence will contain ur, du, or dd as a subsequence. Since such a subsequence will invariably cause the variable to enter state A, all we need to do is to monitor the states assumed by the variable during execution. This can be readily accomplished by means of program instrumentation.

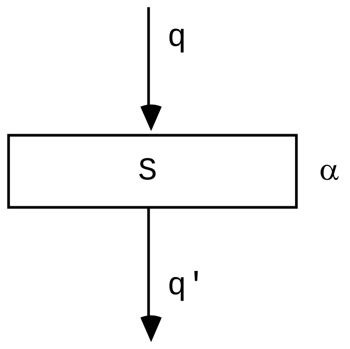


Fig. 2. In this diagram S is a statement, α is the sequence of actions taken on variable x by S, and q and q' are states assumed by x before and after an execution of S. Furthermore, $q'=f(q, \alpha)$.

To see how this can be done, let us consider a fragment of a flowchart shown in Fig. 2. Suppose we wish to detect data flow anomalies with respect to a variable, say, x. If x is in state q before statement S is executed, and if α is the sequence of actions that will be taken on x by S, then an execution of S will cause x to enter state q', as depicted in Fig.2. Given q and α, q' can be determined based on the state diagram given in Fig. 1. However, for the discussions that follow, it is convenient to write

$$q'=f(q, \alpha) \tag{1}$$

where f is called the state transition function and is completely defined by the state diagram shown in Fig. 1. Thus, for example, f(U, d)=D, and f(D, u)=A. For the cases where α is a sequence of more than one action, the definition of f can be given as follows. Let $\alpha = a\beta$, where a is either d, r, or u, and β is a sequence of d's, r's, and u's. Then

$$f(q, a\beta) = f(f(q, a), \beta)$$

for any q in {A, D, R, U}. Thus f(U, dur) = f(f(U, d), ur) = f(D, ur) = f(f(D, u), r) = f(A, r) = A.

Next, we observe that the computation specified by (1) can be carried out by using a program statement of the form

$$q := f(q, \alpha). \tag{2}$$

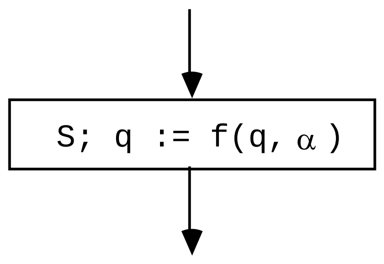


Fig. 3. Program instrumented with `q:=f(q, α).`

Now if we insert the above statement next to statement S in Fig. 2, as shown in Fig. 3, then the new state assumed by variable x will be automatically computed upon an execution. The augmented program depicted in Fig. 3 is said to have been instrumented with the statement q := f(q, α). This statement should be constructed in such a way that there will be no interference between this inserted statement and the original program. A simple way to accomplish this is to use variables other than those which appeared in the program to construct the inserted statement.

To illustrate the idea presented above, let us consider an execution path shown in Fig. 4. Suppose we wish to detect possible data flow anomalies with respect to variable x along this path. According to the method described above, we need to instrument the program with statements of the form xstate := f(xstate, α), as shown in Fig. 5. The variable "xstate" contains the state assumed by x. At the entry variable x is assumed to be undefined, and therefore, variable xstate is initialized to U. By an execution along the path, xstate will be set to different values as indicated on the right-hand side of Fig. 5. Note that there is no need to place an instrument following a statement unless that statement will act on variable x. To see if there is a data flow anomaly with respect to x on the path, all we need to do is to print out the value of xstate by instrumenting the program with an appropriate output statement at the exit. In this example, the data flow with respect to x is anomalous in that x is defined and defined again, and the value of xstate will be set to A to indicate this fact.

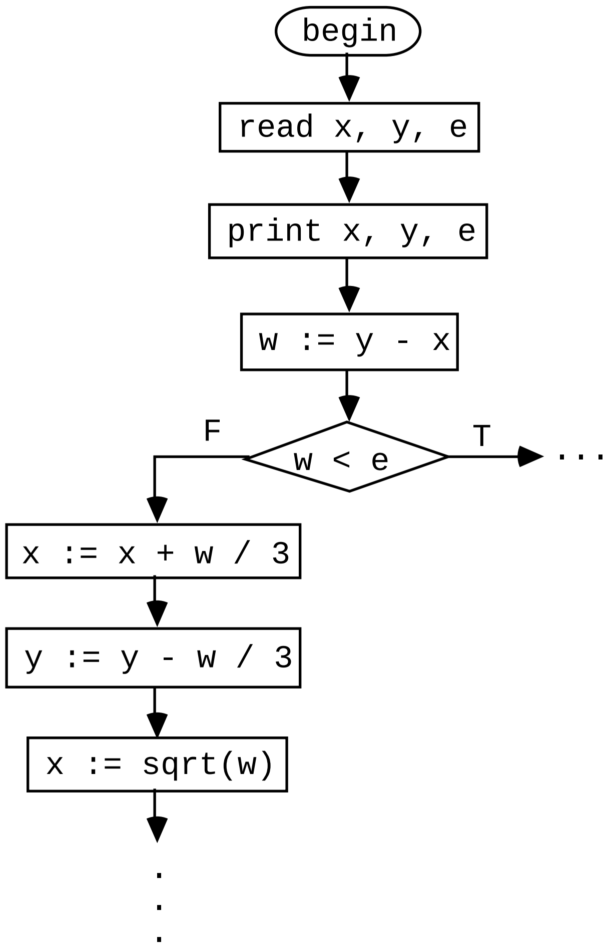
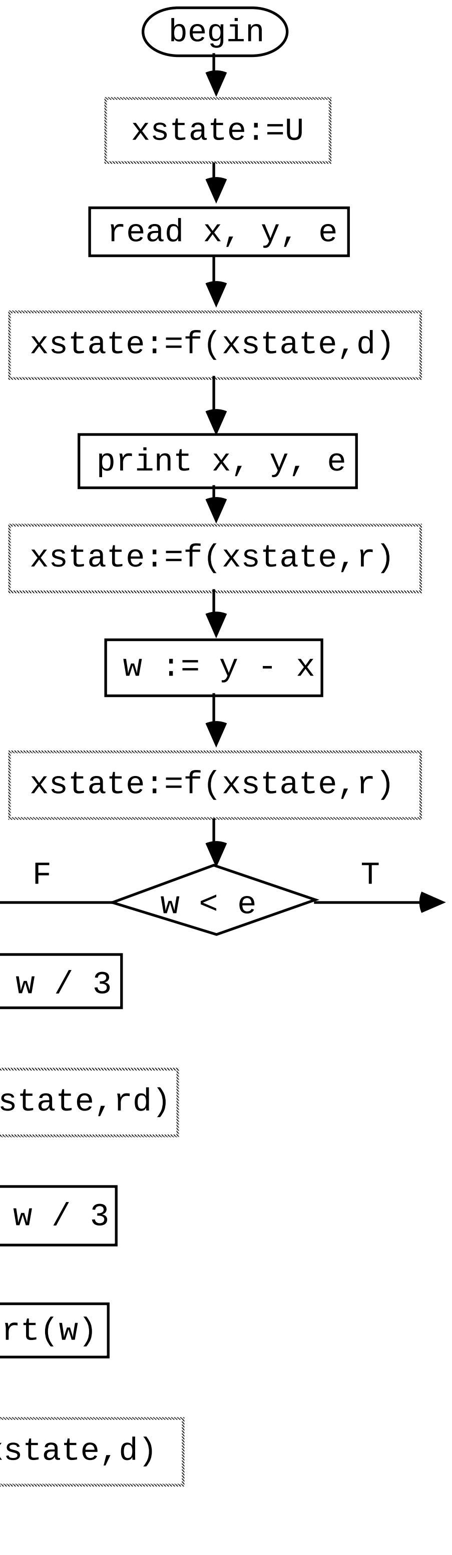


Fig. 4. A program path.



values of xstate
<u>upon an execution</u>

xstate = U

xstate = D

xstate = R

xstate = R

xstate = D

xstate = A

Fig. 5. This diagram illustrates how the program path shown in Fig. 4 can be instrumented to monitor the data flow with respect to variable x (variable xstate contains the state assumed by x).

In practice, it is more appropriate to instrument programs with procedure calls instead of assignment statements. The use of a procedure allows us to save the identification of an instrument as well as the state assumed by the location as well as the type of data flow anomaly detected. This will greatly facilitate anomaly analysis.

3. DATA FLOW OF ARRAY ELEMENTS

To instrument a program for detection of data flow anomalies, as described in the preceding section, we need to be able to identify the actions taken by each statement in the program as well as the objects of actions taken. This requires additional considerations if array elements are involved. The sequence of actions taken by a statement on a subscripted variable can be determined as usual. However, identification of the object may become a problem if the subscript is a variable or an arithmetic expression. First, we do not know which element of the array that variable is meant to be without looking elsewhere. Second, the object of action taken may be different every time that statement is executed.

This problem becomes very difficult when data flow anomalies are to be detected by means of static analysis. In the method described in [1], this problem is circumvented entirely by ignoring subscripts and treating all elements of an array as if they were a single variable. It is interesting to see what entails when this approach is taken. For this purpose, let us consider the amiliar sequence of three statements given below which exchanges the values of a[j] and a[k]:

```
temp := a[j];
a[j] := a[k];
a[k] := temp;
```

It is obvious that the data flow for every variable involved is not anomalous, provided j <> k. However, if a[j] and a[k] are created as the same variable, the data flow becomes anomalous because it is defined and defined again by the last two statements. This example shows that a false alarm may be produced if we treat all elements of an array as if they were a single variable. False alarm is a nuisance, and most importantly, a waste of programmer's time and effort. In some cases, a data flow anomaly will not be detected if we treat all elements of an array as if they were a single variable. For example, let us consider the following program:

```
i := 1;
while i <= 10 do begin a[i] := a[i+1]; i := i + 1 end;
```

If a[i] is mistakingly written as a[1], the data flow for a[1] becomes anomalous because it is repeatedly defined ten times. This is not so if all elements of the array are treated as a single variable.

From the above discussion it is obvious that separate handling of array elements is highly desirable. The problem posed by array elements can be easily solved if the present method of program instrumentation is used. In this method data flow anomalies are to be detected by the software instruments placed among program statements. When it comes to execute a software instrument involving a subscripted variable, the value of its subscript has already been computed (if the subscript is a single variable) or can be readily computed (if it is an arithmetic expression). Therefore, in the process of instrumenting a program for checking the data flow of a subscripted variable, there is no need to know which element of the array that variable is meant to be. The true object of actions taken on this variable can be determined dynamically at the execution time.

To implement the idea outlined above on a computer, we need 1) to allocate a separate memory location to each and every element in the array for the purpose of storing the state presently assumed by that element, and 2) to instrument the program with statements that will change the state of the right array element at the right place. The complexity of statements required depends on the data structure used in storing the states of the array elements.

One simple structure that can be used is to store the states of elements of an rray in the corresponding elements of another array of the same dimension. Statements of the form shown in Fig. 5 can then be used to monitor the states assumed by the array elements. For example, suppose a program makes use of a two-dimensional array a[1:10, 1:20]. To instrument the program to monitor the data flow of elements in this array, we can declare another integer array of the same size, say, sta[1:10, 1:20], for the purpose of storing the states of elements in array a. Specifically, the state of a[i, j] will be stored in sta[i, j]. If the program contains the following statement:

```
a[i, j] := a[i, k] * a[k, j]
```

then the required instruments for this statement will be

```
    sta[i, k] := f(sta[i, k], r);

    sta[k, j] := f(sta[k, j], r);

and sta[i, j] := f(sta[i, j], d).
```

Here f is the state transition function defined by the state diagram shown in Fig. 1.

4. SELECTION OF INPUT DATA

After having a program instrumented, as described in the preceding sections, possible data flow anomalies can be detected by executing the program for a properly chosen set of input data. The input data used determines the execution paths and, therefore, affects the number of anomalies that can be detected in the process. The question now is: how do we select input data so that all data flow anomalies can be detected? It turns out that there is a relatively simple answer to this question. Roughly speaking, we need to select a set of input data that will cause the program to be executed along all possible execution paths that iterate a loop zero or two times. For instance, if the program has a path structure depicted in Fig. 6, we need to choose a set of input data that will cause the program to be executed along paths ae, abd, and abccd. In the remainder of this section we show how this selection criterion is derived, and discuss how a set of input data satisfying this criterion can be found.

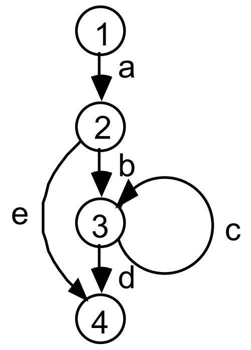


Fig. 6. A graph.

It is intuitively clear that all data flow anomalies will be detected if the instrumented program is executed along all possible execution paths. However, it is impractical, if not impossible, to do so because in general the number of possible execution paths is very large, especially if the program contains a loop and the number of times the loop will be iterated is input dependent. The crucial problem then is to determine the minimum number of times a loop has to be iterated in order to ensure detection of all data flow anomalies.

To facilitate discussion of the problem stated above, we shall adopt the following notational convention. We shall use special symbols α, β, and γ to denote strings of d's, r's, and u's. If α is a string and n is a nonnegative integer, then α^n denotes a string formed by concatenating n α's. For any string α, α^0 is defined to be an empty string.

Now let us consider the data flow with respect to a variable, say, x, on an execution path. Let β represent the sequence of actions taken on x by the constituent statements of a loop on this path. If the loop is iterated n times in an execution, then the sequence of actions taken by this loop structure can be represented by β^n. Thus, if the program is executed along this path, the string representing the sequence of actions taken on x will be of the form $\alpha\beta^n\gamma$. Recall that to determine if there is a data flow anomaly with respect to x is to determine if dd, du, or ur is a substring of $\alpha\beta^n\gamma$. Therefore, the present problem is to find the least integer k such that if $\alpha\beta^n\gamma$ (for some $n > k$) contains either dd, du, or ur as a substring, then so does $\alpha\beta^k\gamma$.

For convenience, we shall use *.substr.* to denote the binary relation "is a substring of". Thus r *.substr.* rrdru, and ur *.substr.* ddrurd.

Theorem 1: Let α, β, and γ be any nonempty strings, and let τ be any string of two symbols. Then, for any integer $n > 0$,

$$\tau \textit{ .substr. } \alpha\beta^n\gamma \text{ implies } \tau \textit{ .substr. } \alpha\beta^2\gamma$$

Proof: For $n > 0$, τ can be a substring of $\alpha\beta^n\gamma$ only if τ is a substring of α, β, γ, $\alpha\beta$, $\beta\beta$, or $\beta\gamma$. However, all of these are a substring of $\alpha\beta^2\gamma$. Thus the proof immediately follows from the transitivity of the binary relation *.substr.*. Q.E.D.

Note that dd, du, and ur are strings of two symbols, representing the sequences of actions that cause data flow anomalies. Theorem 1 says that, if there exists a data flow anomaly on an execution path that traverses a loop at least once, anomaly can be detected by iterating the loop twice during execution. Such a data flow anomaly may not be detected by iterating the loop only once because dd, du, and ur may be a substring of $\beta\beta$, and $\beta\beta$ is not necessarily a substring of $\alpha\beta\gamma$.

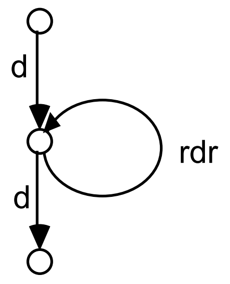


Fig. 7. An example execution path in which the data flow becomes anomalous only if the loop is not executed.

Observe that Theorem 1 does not hold for the case $n = 0$. This is so because τ *.substr.* $\alpha\gamma$ implies that τ is a substring of α, γ, or $\alpha\gamma$, and $\alpha\gamma$ is not necessarily a substring of $\alpha\beta^n\gamma$ for any $n > 0$. The significance of this fact is that a certain type of data flow anomaly may not be detected if a loop is traversed during execution. Fig. 7 exemplifies this type of data flow anomaly. In general, if the data flow anomaly is caused by exclusion of a loop from the execution path, then it may not be detected if the loop is traversed during execution.

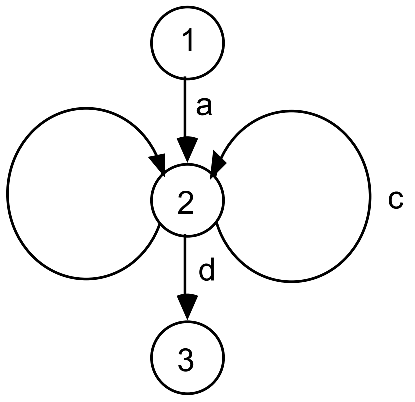


Fig. 8. A path structure.

Based on Theorem 1 and the above discussion, we can conclude that *to ensure detection of all data flow anomalies, each loop in a program has to be iterated zero and two times in execution*. Unfortunately, it is not clear how this result can be applied to the cases where a loop consists of more than one path. For instance, if we have a path structure, shown in Fig. 8, we are certain that paths abbd, accd, and ad have to be covered in input data selection. However, it is not clear whether paths such as abbccd, abcbcd, or abcd have to be covered.

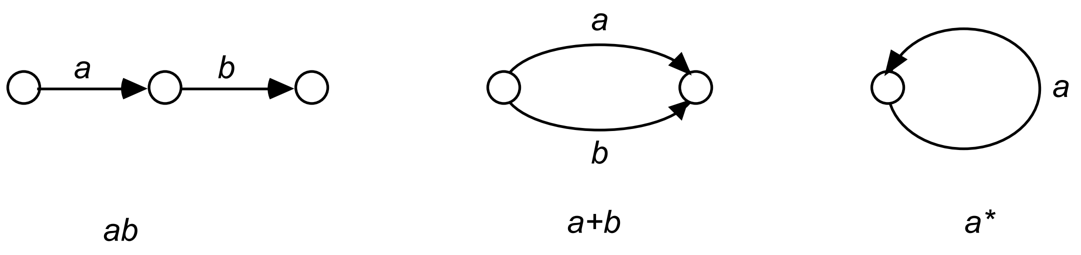


Fig. 9. Three basic path structures and their descriptions.

To clarify this point, first we need to be able to speak of a path structure precisely and concisely. This can be accomplished by making use of the language of regular expressions (see, e.g., [9-11]). Briefly, a set of paths between any two nodes in a (directed) graph can be described in terms of symbols associated with the constituent edges as follows. For two edges (labeled by) a and b, we shall use ab to describe the path formed by connecting a in cascade with b, use a+b to describe the path structure formed by connecting a and b in parallel, and use a* to describe the loop formed by using a, as shown in Fig. 9. The same rules also apply to the cases where a and b are expressions describing complex path structures. Hence a set of paths can be described by an expression composed of edge symbols and three connectives: concatenation, disjunction (+), and looping (*). For example, the set of paths between nodes 1 and 4 in Fig. 6 can be described by a(e + bc*d), and that between nodes 1 and 3 in Fig. 8 can be described by a(b + c)*d. (Remark: Those who are familiar with the concept may have noted that the latter can be alternatively described by a(b*c*)*d. We shall comment on this later.)

If p describes a path, then p* describes a loop formed by p and hence a set of paths obtained by iterating the loop for any number of times. Formally, $p^* = \lambda + p + pp + ppp + \ldots$. Here λ is a special symbol denoting the identity under concatenation (i.e., $x\lambda = \lambda x = x$ for any x) and is to be interpreted as a path of length zero (obtained by iterating the loop zero times). According to the result presented above, we need only to iterate the loop zero and two times in order to ensure detection of all data flow anomalies. Thus if a path description contains p* as a subexpression, we can replace it with (λ + p2) to yield the description of the paths that have to be traversed in execution.

Does the same method apply if p is a description of a set of two or more paths? In that case, an execution of statements on p will result in having two or more sequences of actions taken on the variable. Therefore, the answer hinges on whether or not we can extend Theorem 1 to the cases where β is a set of strings. It turns out that the answer is affirmative.

To see why this is so, we shall first restate Theorem 1 for the cases where α, β, and γ are sets of strings. Note that the concatenation of two sets is defined as usual. That is, if α and β are sets of strings, then $\alpha\beta = \{ab \mid a \text{ in } \alpha \text{ and } b \text{ in } \beta\}$ is again a set of strings.

Theorem 2: Let α, β, and γ be any nonempty sets of nonempty strings, τ be any string of two symbols, and n be an integer greater than zero. If τ is a substring of an element in $\alpha\beta^n\gamma$ then τ is a substring of an element in $\alpha\beta^2\gamma$.

Theorem 2 is essentially the same as Theorem 1 except that the binary relation of "is a substring of" is changed to that of "is a substring of an element in." As such, it can be proved in the same manner. The proof of Theorem 1 *mutatis mutandis* can be used as the proof of Theorem 2.

For convenience, we now introduce the notion of a zero-two (ZT) subset. Given an expression E that describes a set of paths, we can construct another expression E_{02} from E by substituting $(\lambda + p^2)$ for every subexpression of the form p* in E. For example, if E is a*bc*d, then E_{02} is $(\lambda + a^2)b(\lambda + c^2)d$. The set of paths described by E_{02} is called a ZT subset of that described by E.

The development presented above shows that, to ensure detection of all data flow anomalies, it suffices to execute the instrumented program along paths in a ZT subset of the set of all possible execution paths. The question now is: how do we select input data to accomplish this? Described in the following are the steps that may be taken to find the required set of input data for a given program.

Step 1: Find all paths from the entry to the exit in the flow chart of the program. A flowchart is essentially a directed graph, and there are several methods available for finding all paths between two nodes in a directed graph (see, e.g., [12-15]).

Step 2: Find a ZT subset of the set of paths found in Step 1. Note that the regular-expression representation of a set of paths is not unique in general. For instance, the set of paths between nodes 1 and 3 in Fig. 8 can be described by $a(b + c)^*d$ or equivalently by $a(b^*c^*)^*d$. Since a ZT subset is defined based on the set description, a set may have more than one ZT subset. In this example, there are two. One is described by $a(\lambda + (b + c)^2)d$ and the other by $a(\lambda + ((\lambda + b^2)(\lambda + c^2))^2)d$. However, this is of no consequence because in the light of Theorem 2 the use of either one is sufficient to ensure detection of all data flow anomalies.

Step 3: For each path in the set obtained in Step 2, find input data that will cause the program to be executed along that path. This may prove to be a rather difficult task in practice. The reader may wish to refer to [16-18] for methods available. Note that the set obtained in Step 2 may contain paths that cannot be executed at all. If a path is unexecutable because there is a loop on the path that has to be iterated for a fixed number of times other than that specified, then disregard the number of times the loop will be iterated in execution. Just select input data that will cause the path (and the loop) to be executed. If a path is found to be unexecutable because a loop can only be traversed a number of times other than that specified, then replace it with an executable path that traverses the loop two or more times. If a path is found to be unexecutable because it is intrinsically so, then it can be excluded from the set. The result is a set of input data that will ensure detection of all data flow anomalies.

5. VARIABLE ALIASING AND RELATED PROBLEMS

In a program, the same name may be used by different data, and the same datum may have different names. This complicates the process of program instrumentation in two ways. First, additional analysis will be required to recognize multiple use of a name and to identify aliases of a variable. Second, based on the analysis results, appropriate mechanisms must be incorporated into the software instruments to ensure that there will be no confusion about the identity of a datum. The first part can be done in a straightforward manner and, therefore, will not be discussed here. The second part will be treated in detail in this section.

In most programming languages, local variables in different blocks or subroutines may have the same name. It is possible to instrument the program in such a way that the state of the data with the same name will be stored in the same memory location. However, this requires additional software instruments at the boundaries to store and restore properly the state of the variable in the outer block or the calling program. Another possible solution is to give each local variable an additional identification so that they will become different. This can be done, for example, by identifying each datum by its name in conjunction with the name of the block or subroutine in which it exists. The state of each variable will then be stored in a separate location, and thus no additional instruments will be required at the block or subroutine boundaries.

Two different variables, say, x and y, may in fact represent the same datum under the following circumstances. 1) Through the use of an EQUIVALENCE statement in Fortran or DEFINED attribute in PL/1, the same memory space is allocated to x and y. 2) Through the use of COMMON statements in Fortran, variable x in one subprogram shares the same memory with variable y in another subprogram. 3) x is the so-called actual name used in the invoking procedure, and y is the so-called dummy name used in the invoked procedure. To show a possible solution to the problem of variable aliasing, let us suppose that the states assumed by x and y are to be contained in variables stx and sty. Then, in the first two cases mentioned above, all we need to do is to make variables stx and sty share the same memory through the use of the same mechanism. If there is an EQUIVALENCE statement for x and y in the original program, then we should have the corresponding EQUIVALENCE statement for stx and sty in the instrumented program. If x and y are declared common in two subprograms, then stx and sty should be similarly declared.

To see how we may solve the problem for the third case, let us suppose that y is the sole parameter of a subroutine named exampro. What needs to be done when this subroutine is invoked by the statement: call exampro(x)? Obviously, we need to initialize the value of sty to the value of stx just before the subroutine exampro is entered. Furthermore, the value of stx must be restored to the value of sty upon the exit from exampro. The transfer of value between stx and sty cannot be achieved through the use of an assignment statement because stx is not accessible to the subroutine exampro, and sty is not accessible to the calling program. In general, they cannot be made accessible by means of inserting additional statements into the program. A possible solution to this problem is to make use of a software instrument which we call a subroutine named que(func, var). The first parameter can be "in" or "out," and the second parameter is a variable name. When invoked by the statement: call que(in, stx), this subroutine will store the value of variable stx in a queue (i.e., a first-in-first-out memory device). When invoked by the statement: call que(out, stx), the subroutine wil delete an element from the queue and store it in stx. We can use this instrument to properly transfer the value from stx to sty and vice versa, as illustrated below.

```
calling program                          called subroutine

      .

      .

      .

call que(in, stx);
call exampro(x);              subroutine exampro(y);
```

```
   call que(out, stx);             call que(out, sty);
         .                               .
         .                               .
         .                               .
                                (body of the subroutine)
                                         .
                                         .
                                         .
                                   call que(in, sty);
                                   return;
```

The italicized statements are the software instruments. Their purpose is to store the state assumed by x just before the subroutine exampro is called and then to initialize the state of y to the stored value immediately after the subroutine is entered. The reverse of this process is carried out at the exit. A queue is used to store the state because a subroutine may have more than one parameter. The states of parameters thus can be passed through subroutine boundaries one at a time by using a sequence of calls to subroutine que(func, var). The subroutine que must be designed in such a way that the content of the queue will not be lost upon termination.

In summary, there is a need to pass the value of state variables when the control crosses block or subroutine boundaries. The above example is used to show how the problem can be solved in principle. In practice, the use of a global queue described above has several shortcomings. For example, when the variable x is an array, stx will also be an array. Not many languages allow a subroutine to have scalars, arrays, and arrays of different dimensions to be called all in the same argument position. Instruments for expressions containing several function calls must be properly ordered to ensure that elements are stored in the queue according to the order in which the functions will be called. Clearly, a more elaborate mechanism for interprogram communication must be employed in implementing the present method.

6. SEGMENTATION OF DATA FLOW

In the previous discussions we have tacitly assumed that the whole program will be instrumented for the data flow analysis. However, there are cases in which we would like to apply the method to a portion of the program only. For instance, we might have made changes to a small part of a large program and want to apply the method to detect possible errors in the portion changed. We should be able to do this without instrumenting the entire program. The applicability of this method to parts of a program is also important in the development of a large software. A large program normally consists of many subprograms, each of which is constructed and tested separately. It will be very helpful if the method can be applied to each subprogram individually. The need

to analyze an isolated segment of data flow may also arise when the program calls a subroutine that is not available for instrumentation. A system library routine is an example. We should be able to analyze data flow in the rest of the program when a part of it is not instrumented. In the remainder of this section, we shall discuss the problems involved and show how they may be solved.

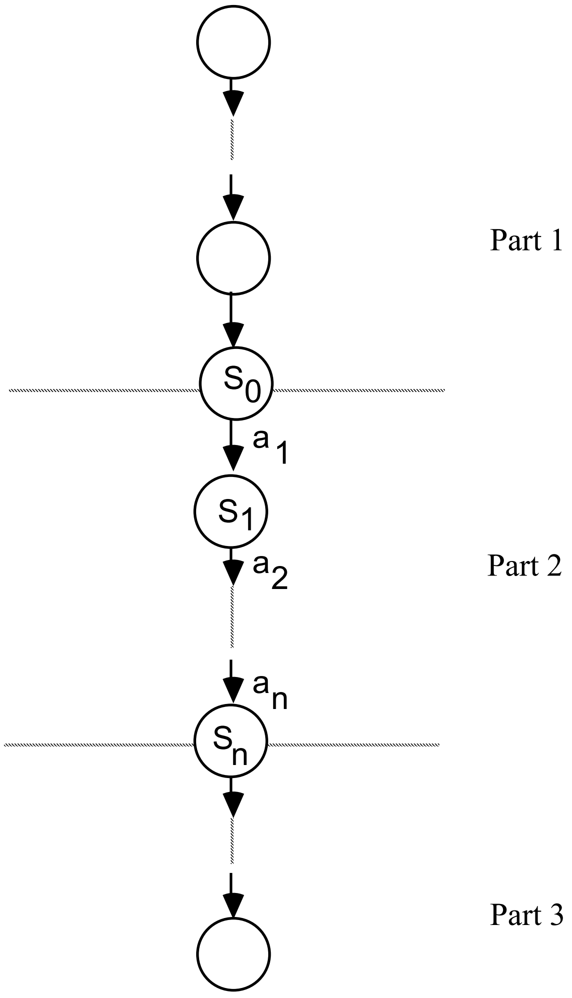


Fig. 10. An execution path.

To see what is involved, let us suppose that we wish to analyze the data flow with respect to a variable, say, x, along the middle part of the execution path depicted in Fig. 10. Let us further suppose that the statements on this part will take a sequence of n actions a_1, a_2, ..., a_n on variable x, and as the result x will change its state from the initial state so to a sequence of new states s_1, s_2, ..., s_n. If we start the analysis from the entry of the program, we know that so = U, i.e., the variable is in the state of being undefined. However, if we start the analysis from any point other than the program entry, we would not know the state of the variable at the starting point. Without this knowledge, we cannot compute the subsequent states assumed by the variable. A possible solution to this problem is to assume that the variable is in a certain state at the starting point. There are four possible choices, viz., we can assume that so is U, D, R, or A.

Recall that state A indicates the existence of a data flow anomaly, and the variable will stay in state A once that state is entered. Hence it is pointless to assume that so is A.

Next, we observe that if s_0 = U and a_1 is "reference", then s_1 will be A (cf., Fig. 1). Also, if s_0 = D, and if a_1 is "define" or "undefine," then s_1 will become A as well. Since our purpose here is to analyze the data flow within the middle part of the execution path depicted in Fig. 10, and since it takes two consecutive actions to cause a data flow anomaly, it is undesirable to produce an indication of data flow anomaly after the first action. Therefore, it is undesirable to assume that s_0 is U or D.

We shall assume that so is R because state R does not have the above-mentioned problem with states U and D. In addition, the variable will enter state A if the sequence of actions $a_1a_2...a_n$ contains dd, du, or ur as a subsequence. That is, by assuming that s_0 = R, we can detect all data flow anomalies that may exist within the boundaries of the execution path under consideration.

From the above analysis it is clear that we can instrument a portion of a program for data flow analysis as usual. And if we do not know the state of the variable at the starting point, simply assume it to be state R. The same rule applies to the case in which the program calls a subroutine that is not instrumented for some reason. All we need to do is to reset the state of a variable to R when the control returns to the calling program and continue the analysis as usual.

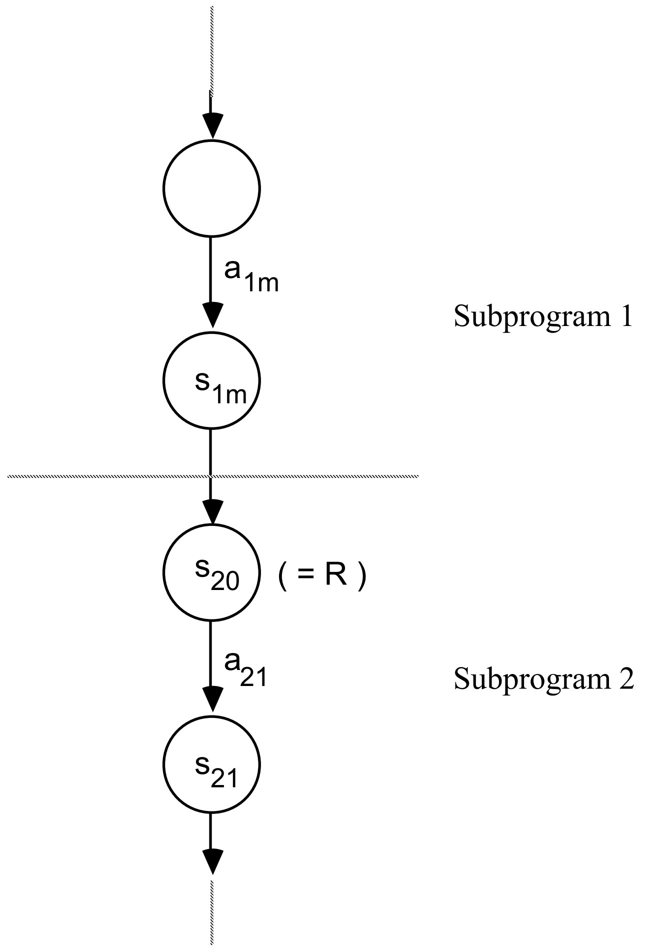


Fig. 11. Data flow at the subprogram boundary.

If we wish to analyze the data flow of a program by applying the method to each subprogram individually, we need to be able to detect the data flow anomalies that may occur on the boundarry of two subprograms. To see how this can be accomplished, let us consider the execution path depicted in Fig. 11, which crosses the boundary of two subprograms. Let a_{1m} be the last action taken by subprogram 1 on a particular variable, and let a_{21} be the first action taken by subprogram 2 on the same variable. By definition, there is a data flow anomaly if $a_{1m}a_{21}$ is either dd, du, or ur. We can make use of the following relation to detect such anomalies:

$a_{1m} = d$ if $s_{1m} = D$

$a_{1m} = r$ if $s_{1m} = R$

$a_{1m} = u$ if $s_{1m} = U$

$a_{21} = d$ if $s_{21} = D$

$a_{21} = r$ if $s_{21} = R$

$a_{21} = u$ if $s_{21} = U$

We need not be concerned with the case in which $s_{1m} = A$ because it is an indication that there is a data flow anomaly within subprogram 1. Presently, we are concerned with the detection of anomalies on the boundary. We also need not consider the case in which $s_{21} = A$. That will never happen because we assume that $s_{20} = R$ in computing s_{21}. Combining the results, we see that there is a data flow anomaly on the boundary if and only if

$s_{1m} = D$ and $s_{21} = D$ (because $a_{1m}a_{21} = dd$)

$s_{1m} = D$ and $s_{21} = U$ (because $a_{1m}a_{21} = du$)

or

$s_{1m} = U$ and $a_{21} = R$ (because $a_{1m}a_{21} = ur$). (3)

For the sake of argument, let us call s_{20} and s_{21} the initial state and the first computed state, respectively. For a variable whose data flow crosses a subprogram boundary, we can determine if there is a data flow anomaly on the boundary if we know its last state in the first subprogram and its first computed state in the second subprogram, provided the initial state is set to R in performing the analysis. Hence in applying the present method to a subprogram, we shall use appropriate software instruments to initialize the variable state to R at the entry, and to record the first computed state and the last state.

The information so obtained can be interpreted as follows. According to (3), there is no anomaly at the boundary if the last state is R. If the last state is either U or D, then we have to check if one of the three conditions in (3) is satisfied. If so, and if it occurs on a possible execution path, then there is a data flow anomaly at the boundary. If the last state is A, there is a data flow anomaly within the subprogram.

7. CONCLUDING REMARKS

The state diagram shown in Fig. 1 is such that, once a variable enters state A, it will remain in that state all the way to the end of the execution path. This implies that, once the data flow with respect to a variable is found to be anomalous at a certain point, the error condition will be continuously indicated throughout that particular execution. No attempt will be made to reset the state of the variable and continue to analyze the rest of the execution path. This appears to be a plausible thing to do because in general it takes a close examination of the program by the programmer to determine the nature of the error committed at that point. Without knowing the exact cause of the anomalous condition, it is impossible to correctly reset the state of that variable.

A possible alternative would be to abort the program execution once a data flow anomaly is detected. This can be accomplished by instrumenting programs with procedure calls that invoke a procedure with this provision. By halting program execution upon discovery of a data flow anomaly, we may save some computer time, especially if the program is large.

As explained in [1], the presence of a data flow anomaly does not imply that execution of the program will definitely produce incorrect results. It implies only that execution may produce incorrect results. Thus we may wish to register the existence of a data flow anomaly when it is detected and then continue to analyze the rest of the execution path. In that case, we can design the software instrument in such a way that, once a variable enters state A, it will properly register the detection of a data flow anomaly and then reset the state of the variable to state R. The reason for resetting it to state R is obvious in the light of the discussions presented in the preceding section. Another alternative is to use the state diagram shown in Fig. 12 instead of the one shown in Fig. 1. The data flow with respect to a variable is anomalous if the variable enters either state define-define (DD), state define-undefine (DU), or state undefine-reference (UR). The use of this state diagram has the additional advantage of being able to identify the type of data flow anomaly detected.

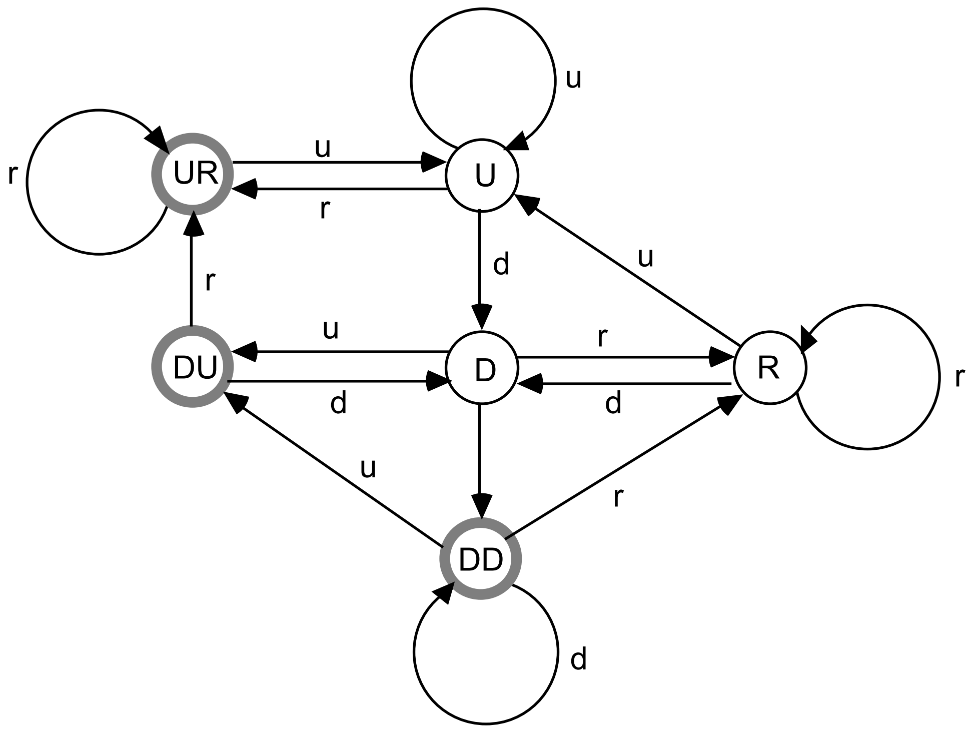


Fig. 12. An alternative to the state diagram shown in Fig. 1.

To simplify the discussion, we have limited ourselves to analysis of data flow with respect to a single variable. However, the method presented can be readily extended to analyze more than one variable at the same time. All we need to do is to modify the method to handle vectors of variables, states, and sequences of actions instead of single ones.

The utility of data flow analysis in error detection is obvious and has been confirmed by practical experience [19], [20] for Fortran programs. Fosdick and Osterweil have developed a static analysis method to obtain the desired information [1]. In this paper we present another method to achieve the same goal by properly instrumenting a program and then executing it for a set of input data. In comparison, the present method has the following advantages.

1) The present method is conceptually much simpler than that described in [1] and, therefore, is much easier to implement.

2) From the nature of computation involved, it is obvious that the present method requires a much smaller program to implement it on a computer.

3) From the user's point of view, the present method is easier to use and more efficient because it produces information about the locations and types of data flow anomalies in a single process. In the method developed by Fosdick and Osterweil, additional effort is required to locate the anomaly once it is detected.

4) As indicated in Section II, the present method can be readily applied to monitor the data flow of elements of an array, which cannot be adequately handled in the static method. Thus the present method has a greater error-detection capability and will produce fewer false warnings.

5) In the present method, there is no need to determine the order in which the subroutines are invoked, and thus the presence of a recursive subprogram will not be a problem.

The method presented in this paper is particularly advantageous if it is used in conjunction with a conventional program test to enhance the error-detection capability. In a conventional test, a program has to be exercised as thoroughly as possible (see, e.g., [16]), and, therefore, the task of finding a suitable set of input data to carry out the data flow analysis will not be an extra burden to the programmer.

It is difficult to compare the cost. Very roughly speaking, the cost of applying the method described in [1] is linearly proportional to the number of statements in the program whereas that of applying the present method is linearly proportional to the execution time. Therefore, it may be more economical to use the method described in [1] if the program is of the type that consists of a relatively small number of statements, but it takes a long time to execute (viz., a program that iterates a loop a great number of times is of this type).

In conclusion, we have shown that the data flow anomaly can be detected by means of program instrumentation. Most importantly, we have found a simple criterion for input data selection, satisfaction of which guarantees detection of all data flow anomalies. It is interesting to note that the technique of instrumenting a program for information gathering purposes has many other applications. For example, it has been utilized to generate a program profile by Russell and Estrin [21], to measure instruction mix and execution time by Bussell and Koster [22], to produce information about syntactic and operational characteristics of programs by Stucki [23], and to measure the thoroughness of a test [24-29]. The work reported in this paper further demonstrates the utility of this technique as a tool for program analysis and testing.

References

[1] L. D. Fosdick and L. J. Osterweil, "Data flow analysis in software reliability," *ACM Computing Surveys*, vol. 8, Sept. 1976, pp. 305–330.

[2] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling*. Englewood Cliffs, NJ: Prentice-Hall, 1972.

[3] M. Schafer, *A Mathematical Theory of Global Program Optimization*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[4] M. S. Hecht and J. D. Ullman, "A simple algorithm for global data flow analysis problems," *SIAM J. Computing*, vol. 4, pp. 519–532, Dec. 1975.

[5] K. Kennedy, "A global flow analysis algorithm," *Int. J. Comput. Math.*, vol. 3, pp. 5–15, 1971.

[6] G. A. Kildall, "A unified approach to global program optimization," in *Conf. Record ACM Symp. on Principles of Programming Languages*, Boston, MA, pp. 194–206, 1973.

[7] J. D. Ullman, "Fast algorithm for the elimination of common subexpressions," *Acta Informatica*, vol. 2, pp. 191–213, 1973.

[8] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, vol. 19, pp. 137–147, Mar. 1976.

[9] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[10] M. A. Harrison, *Introduction to Switching and Automata Theory*. New York, NY: McGraw-Hill, 1965.

[11] T. L. Booth, *Sequential Machines and Automata Theory*. New York, NY: Wiley, 1967.

[12] N. J. A. Sloane, "On finding the paths through a network," *Bell Syst. Tech. J.*, vol. 51, pp. 371–390, Feb. 1972.

[13] A. G. Lunts, "A method of analysis of finite automata," *Soviet Physics–Doklady*, vol. 10, pp. 102–103, 1965.

[14] C. Berge, *Theory of Graphs and Its Applications*. New York, NY: Wiley, 1962.

[15] F. Harary, *Graph Theory*. Reading, MA: Addison-Wesley, 1969.

[16] J. C. Huang, "An approach to program testing," *ACM Computing Surveys*, vol. 7, pp. 113–128, Sept. 1975.

[17] L. Clarke, "A system to generate test data and symbolically execute program," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 215–222, Sept. 1976.

[18] W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. Comput.*, vol. C-24, pp. 554–559, May 1975.

[19] L. J. Osterweil and L. D. Fosdick, "DAVE–A Fortran program analysis system," in *Proc. Comput. Sci. and Statistics 8th Annual Symp. on the Interface*, pp. 329–335, 1975.

[20] ——, "DAVE–A validation, error detection, and documentation system for Fortran programs," *Software–Practice and Experience*, vol. 6, pp. 473–486, 1976.

[21] E. C. Russell and G. Estrin, "Measurement based automatic analysis of Fortran programs," in *Proc. 1969 AFIPS Spring Joint Comput. Conf.*, vol. 34, 1969.

[22] B. Bussell and R. A. Koster, "Instrumenting computer systems and their programs," in *Proc. 1970 AFIPS Fall Joint Comput. Conf.*, vol. 37, pp. 525–534, 1970.

[23] L. G. Stucki, "Automatic generation of self-metric software," in *Proc. 1973 IEEE Symp. on Comput. Software Reliability*, New York, NY, Apr. 1973, pp. 94–100.

[24] L. G. Stucki, "A prototype automatic program testing tool," in *Proc. 1972 AFIPS Fall Joint Comput. Conf.*, vol. 41, pp. 829–836, 1972.

[25] J. R. Brown, A. J. DeSalvio, D. E. Heine, and J. G. Purdy, "Automatic software quality assurance," in *Program Test Methods*, W. C. Hetzel, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1973, pp. 181–203.

[26] M. R. Paige and J. P. Benson, "The use of software probes in testing FORTRAN programs," *Comput.*, vol. 7, pp. 40–47, July 1974.

[27] C. V. Ramamoorthy and S. B. F. Ho, "Testing large software with automated software evaluation systems," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 46–58, Mar. 1975.

[28] B. C. Hodges and J. P. Ryan, "A system for automatic software evaluation," in *Proc. Second Int. Conf. on Software Eng.*, pp. 617–623, Oct. 1976.

[29] J. C. Huang, "Program instrumentation and software testing," *Comput.*, vol. 11, pp. 25–32, April 1978.