CHAPTER
SEVEN

Program Instrumentation

7.1 Introduction

Normally, programs are designed to produce outputs (and only those outputs) that are required to perform their intended functions. Thus, if a test on a program produced the intended result, the only definitive conclusion that we can draw from that event is that the program will execute correctly with that particular input. It should be of interest to a program tester if we can make a program under test to produce additional information.

In the following, we shall explore the idea of inserting additional statements (instruments) into a program for information gathering purposes. By test-executing the instrumented program for a properly chosen set of test cases, we will be able to obtain information useful for error detection and test management purposes.

Possible applications of this idea include:

- Test-coverage measurement
- Test-case effectiveness assessment
- Assertion checking
- Dataflow-anomaly detection
- Pathwise decomposition

7.2 Test-Coverage Measurement

As mentioned in the preceding chapters, there are at least two commonly used measurements for test coverage, viz.,

C1: Each statement in the program is exercised (i.e., executed) at least once during the test.
C2: Each branch in its control flow is traversed at least once during the test.

Given a program P and a set, T, of test cases, how can we utilize program instrumentation to determine the extent of test coverage achieved by test-executing P with T?

Step 1: Identify a set of points in the control flow of P such that, if we know the number of times each point is crossed during execution, we can determine the number of times each statement is executed (or, each branch is traversed).
Step 2: Instrument P with software counters at these points.
Step 3: Test-execute P with T.
Step 4: Examine the counter values to determine the extent of coverage achieved.

How can a software counter be built?

It can be implemented by using a function (procedure) named, say, *count(j)*, which makes use of an integer array *counter[1..n]*. Every element of this array is set to zero initially. An invocation of *count(j)* causes the value of *counter[j]* to be increased by 1.

Where the counters should be placed?

A possible answer is to place a counter on each branch in the control-flow diagram that emanates from an entry node, or a node with out-degree of two or greater. Such a branch is at the head of a path in the control flow diagram called a decision-to-decision path [MILL74]. When that branch is traversed, every branch on that decision-to-decision path is traversed once and only once. This method is easy to apply, but the number of counters required will not be minimum.

After a test execution of the program, the values of the counters can be listed to show the extent of coverage achieved. If all the counters have a non-zero count, it signifies that C2 has been achieved, i.e., every branch in the control flow has been traversed at least once during the test. If any counter has a zero count, an additional input data should be chosen to cause that branch to be exercised.

Exercise 7.2.1: Formulate a systematic method for finding such an input data, and outline the functional requirements of a software tool that may be used to facilitate its application.

Incidentally, the counter values also point out which portions of the program will be executed many more times than others, and thus will have a greater optimization payoff.

Satisfaction of C1 and C2 can be determined with fewer counters. An interesting question in this regard is: given a program, what is the minimum number of counters required to determine satisfaction of C1 (or C2)? This question is mostly of theoretical interest only because, in practice, the cost of determining the required number of counters, and the locations at which these counters have to be placed, often outweighs the cost of using extra counters in the decision-to-decision-path method. Additional discussions about this question can be found in [KNUT??].

7.3 Test-Case Effectiveness Assessment

By "effectiveness" of a test case here we mean its capability to reveal errors in the program. A test case is ineffective if it causes the program to produce fortuitously correct results, even though the program is erroneous.

One reason why a program may produce a fortuitously correct result is that it contains expressions of the form *exp1 op exp2*, and the test case used causes *exp1* to assume a special value such that *exp1 op exp2* = *exp1* regardless of the value of *exp2*. In that event, if there is an error in *exp2*, it will never be reflected in the test result. Here are some examples of such expressions and test cases:

(a + b) * (c - d) if the test case used is such that a + b = 0.

P(x) and Q(y, z)	if the test case used is such that predicate P(x) becomes false.
P(x, y) or Q(z)	if the test case used is such that predicate P(x,y) becomes true.

We shall say such expressions are *multifaceted* and such test cases *singularly focused* because test-executing a program with a test case is in many ways like inspecting an object in the dark with a flashlight. If the light were singularly focused only on one facet of a multifaceted object, the inspector would not be able to see any flaw on the other facets.

The *singularity index* of a test case with respect to a program is defined as the number of times that a test case is singularly focused on the multifaceted expressions encountered during a particular test run. To compute the singularity index of a test case automatically, a thorough inspection and analysis of every expression contained in the program is required. If we limit ourselves to multifaceted expressions of the form *exp1 op exp2*, the instrumentation tool can be designed to instrument every facet in a multifaceted expression to count the number of time a test case is singularly focused on a facet.

For example, suppose a statement in the program contains the following expression:

```
a * (c + (d / e)).
```

Since there are three facets in this expression, viz., a, (c + (d / e)),and d, a tool can be designed to instrument this expression as shown below:

```
if (a == 0) si++;
if ((c + (d / e)) == 0) si++;
if (d == 0) si++;

a * (c + (d / e));
```

Here si is a variable used to store the singularity index of the test case. The higher the singularity index, the higher the probability that that test case is unable to reveal an error, and hence that test case is less effective.

A measure of singularity index can be used to determine the relative effectiveness of a test case. Suppose that a program has been tested by using two different test cases. If only one of the two sets revealed an error, the relative effectiveness of these two sets is obvious. Nevertheless, if both sets failed to reveal an error, the relative effectiveness of these two sets becomes open to question. A way to settle the question is to measure their singularity indices: the one with a higher singularity index is likely to be less effective for the reason given above.

7.4 Instrumenting Programs for Assertion Checking

Often the intended function of a program can be expressed in terms of assertions that must be satisfied, or values that must be assumed by some variables, at certain strategic points in the program.

Software instruments can be used to monitor the values of variables or to detect any violation of assertions.

The instruments can be constructed by using the host language. The use of a special language, however, will facilitate construction of the instruments and make the process less error prone.

For example, a special high-level language was first used in the Program Evaluator and Tester (PET) developed by Stucki [STFO75]. This language allows the user to describe the desired instrumentation precisely and concisely. A preprocessor in the PET translates all instruments into statements in the host language before compilation.

An interesting feature of the PET is that all instruments are inserted into the program as comments in the host language. After the program is thoroughly tested and debugged, all instruments can be removed simply by recompiling the program without using the preprocessor. In general, the instruments make the source code of a program more readable. There is no reason to remove them physically after the program is completely tested and debugged.

The syntax of this special instrumentation language and its applications are illustrated by the examples given below:

(NOTE: Expressions enclosed in brackets are optional. The vertical bar "|" delimits alternatives.)

Local Assertions:

ASSERT (*extended logical expression*) [HALT on n [VIOLATIONS]]

By "extended logical expression" here we mean a logical expression expressed in the special high-level language, which is not necessarily a grammatically correct logical expression in the host language.

ASSERT ORDER (*array cross-section*)
[ASCENDING | DESCENDING] [HALT ON n VIOLATIONS]]

Examples:

ASSERT (MOVE .LT. 9) HALT ON 10

REMARK: The report produced by PET includes total number of time this instrument was executed, number of time the assertion was violated, and values of MOVE that violated the assertion.

ASSERT ORDER (A(*, 3)) ASCENDING

REMARK: If there were a violation of this assertion, PET would produce a report indicating the array elements and their values that caused the violation.

TRACE [FIRST | LAST | OFF] n [VIOLATIONS]

This construct allows the user to control the number of execution snapshots reported for local assertion violations.

Global Assertions:

Global assertions can be used to replace the use of several similar assertions within a particular program region. Such assertions appear in the declaration section of the program module, and allow us to extend our capacity to inspect certain behavioral patterns for entire program modules. Possible assertions of this type include:

ASSERT RANGE (*list of variables*) (*min*, *max*)
ASSERT VALUES (*list of variables*) (*list of legal values*)
ASSERT VALUES (*list of variables*) NOT (*list of illegal values*)
ASSERT SUBSCRIPT RANGE (*list of array specifications*)
ASSERT NO SIDE EFFECTS (*parameter list*)
HALT ON n [VIOLATIONS]

The last one is designed to stop program execution after n global assertion violations are detected. It is useful in preventing the tool from producing a prohibitively voluminous violation report when there is an assertion violation in a loop, and that loop is iterated a great number of times during program execution.

Monitors:

MONITOR [NUMERIC | CHARACTER] [RANGE]
FIRST [n VALUES] LAST [n VALUES] [ALL | (*list of variables*)]

MONITOR SUBSCRIPT RANGE [ALL | (*list of array names*)]

Examples:

MONITOR RANGE FIRST LAST ALL
MONITOR CHARACTER RANGE (XVAR, YVAR)
MONITOR RANGE (A(*, 3))
MONITOR SUBSCRIPT RANGE (A, B, C)

Assertion checking is potentially an effective means to detect programming errors. When a fault is detected through this mechanism, it not only shows the user how that fault manifests itself in the program execution but also indicates the vicinity, if not the exact location, of that fault. This valuable information is generally not available from an incorrect program output resulting from an ordinary test execution.

A programmer, however, may find it difficult to use in practice. To be effective, the programmer has to place right assertions at the right places in the program, which is not easy to do. The problem of finding right assertions in this application is the same as that in proving program correctness. There is no effective procedure for this purpose. Thus, when a violation is detected, the user has to find out if it is caused by a programming error or an inappropriate assertion. This often is the source of frustration in applying the technique of assertion checking.

7.5 Instrumenting Programs for Dataflow Anomaly Detection

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 fetched 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, if a program is written in some versions of FORTRAN, the index variable of a DO-loop becomes undefined upon termination. Also, the local variables of a block or subprogram become undefined when the control exits from that block or subprogram.

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 without any intervening reference, then we may reasonably conclude that a programming error might have been committed. This idea has been utilized by Fosdick and Osterweil [FOOS76] to detect programming errors.

The three types of data flow anomalies mentioned above can be detected by means of source code analysis [FOOS76]. 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 [ALCO76, FOOS76, HEUL75]. 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.

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 the 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. 7.5.1. 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 implications and possible alternatives will be discussed later.

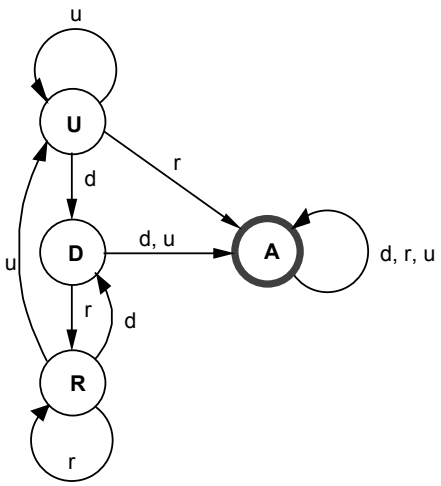


Fig. 7.5.1. Program variables may assume one of the four states of this state diagram (U, undefined; D, defined but not referenced; R, referenced; A, abnormal).

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.

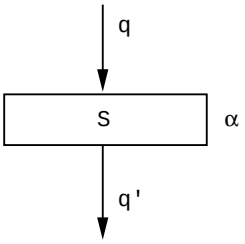


Fig. 7.5.2. In this diagram `S` is a statement, α is the sequence of actions taken on variable `x` by `S`, and `q` and `q'` are states assumed by `x` before and after an execution of `S`. Furthermore, `q'=f(q, α).`

To see how this can be done, let us consider a fragment of a flowchart shown in Fig. 7.5.2. 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. 7.5.2. Given q and α, q'

can be determined based on the state diagram given in Fig. 7.5.1. However, for the discussions that follow, it is convenient to write

$$q' = f(q, \alpha)$$

where f is called the state transition function, and is completely defined by the state diagram shown in Fig. 7.5.1. 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 the expression q' = f(q, α) can be carried out by using a program statement of the form

$$q := f(q, \alpha).$$

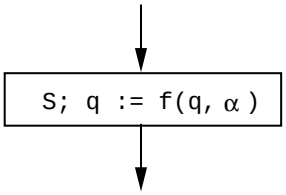


Fig. 7.5.3. Program instrumented with q:=f(q, α).

Now if we insert the above statement next to statement S in Fig. 7.5.2, as shown in Fig. 7.5.3, then the new state assumed by variable x will be automatically computed upon an execution. The augmented program depicted in Fig. 7.5.3 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 that appeared in the program to construct the inserted statement.

To illustrate the idea presented above, let us consider an execution path shown in Fig. 7.5.4. 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. 7.5.5. 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. 7.5.5. 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 reflect this fact.

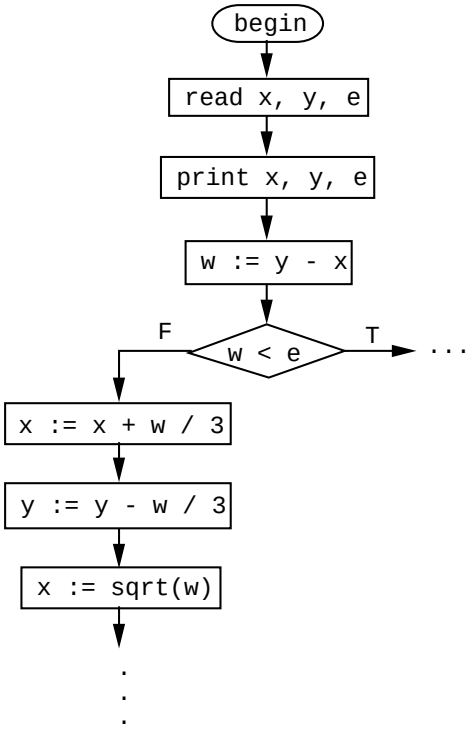


Fig. 7.5.4. A program path.

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, the state assumed by the variable, and the type of data flow anomaly detected. This information will significantly facilitate anomaly analysis.

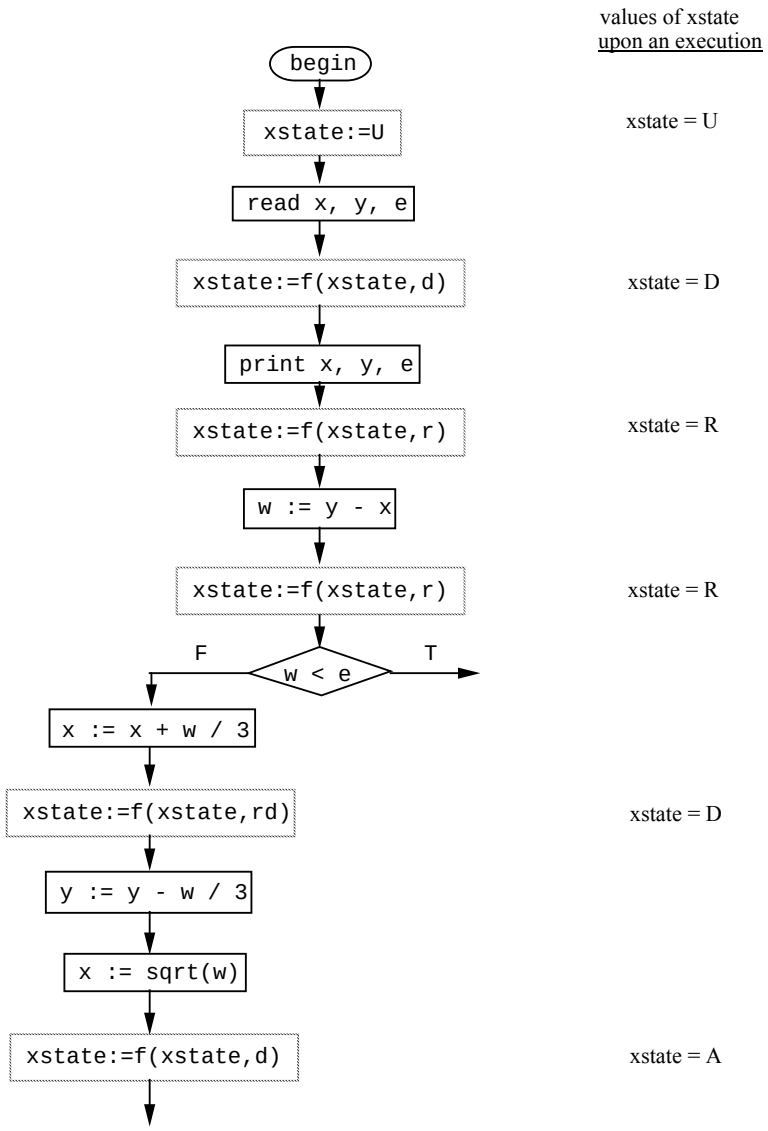


Fig. 7.5.5. This diagram illustrates how the program path shown in Fig. 7.5.4 can be instrumented to monitor the data flow with respect to variable x (variable xstate contains the state assumed by x).

Dataflow of Array Elements

To instrument a program for detection of data flow anomaly as described above, we must 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. Identification of the object, however, 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 [FOOS76], 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 familiar 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 treated 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 mistakenly 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.

These examples clearly demonstrate that, if we treat all elements of an array as the same variable, as it is done in the static analysis method, then it is inevitable to produce false alarm in some cases, and to overlook an anomaly in some others.

Obviously, 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 array in the corresponding elements of another array of the same dimension. Statements of the form shown in Fig. 7.5.5 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. 7.5.1.

Selection of Input Data

After having a program instrumented as described above, 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. 7.5.6, 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.

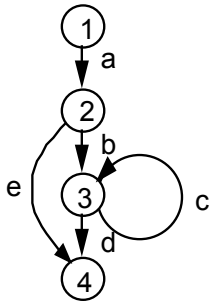
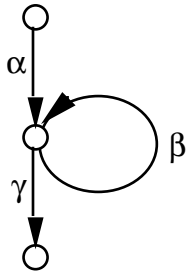


Fig. 7.5.6. A graph.

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 as depicted above. 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 7.5.1: Let α, β, and γ be any nonempty strings, and let τ be any string of two symbols. Then, for any integer $n > 0$,

τ *.substr.* $\alpha\beta^n\gamma$ implies τ *.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 7.5.1 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 $\alpha\beta\gamma$.

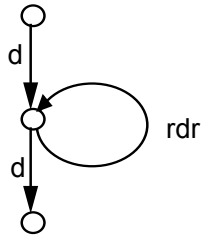


Fig. 7.5.7. An example execution path in which the data flow becomes anomalous only if the loop is not executed.

Observe that Theorem 7.5.1 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. 7.5.7 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.

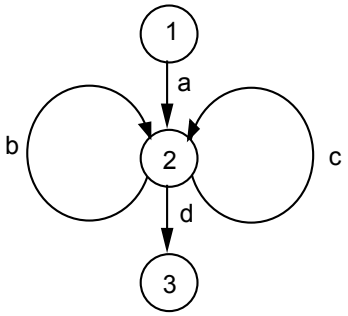


Fig. 7.5.8. A path structure.

Based on Theorem 7.5.1 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. 7.5.8, 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.

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 (ref. Sec. 1.4), we can replace it with $(\lambda + p^2)$ 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 7.5.1 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 7.5.1 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 7.5.2: 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 7.5.2 is essentially the same as Theorem 7.5.1 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 7.5.1 *mutatis mutandis* can be used as the proof of Theorem 7.5.2.

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 the 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., [LUNT65,SLOA72]).

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. 7.5.8 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 [CLAR76,HOWD75,HUAN75] for methods available. Note that the set obtained in Step 2 may contain paths that cannot be executed at all. If a path is not executable 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.

Concluding Remarks

The state diagram shown in Fig. 7.5.1 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 [FOOS76], 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. 7.5.9 instead of the one shown in Fig. 7.5.1. 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.

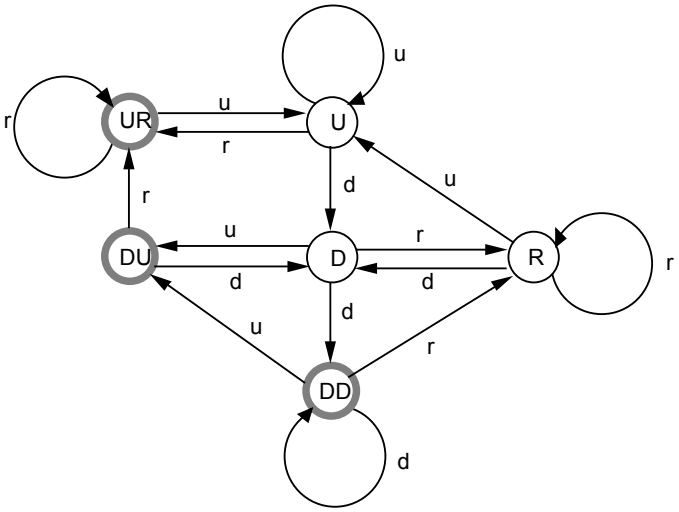


Fig. 7.5.9. An alternative to the state diagram shown in Fig. 7.5.1.

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 [OSFO76] for FORTRAN programs. Fosdick and Osterweil have developed a static analysis method to obtain the desired information [FOOS76]. 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 [FOOS76] 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 previously, 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., [HUAN76]), 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 [FOOS76] 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 [FOOS76] 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).

7.6 Instrumenting Programs for Trace-subprogram Generation

When a program is executed for a particular input, the program output represents the external behavior of the program. To the end users of the program, external behavior is all what it matters. However, to a software engineer, it is often important to understand its internal working as well. The internal behavior of the program (i.e., how does it work to produce the output) can be readily determined by examining the trace subprogram (as defined in Sec. 1.6) associated with that execution.

The trace subprogram of a program can be generated automatically through program instrumentation. The syntax of the host programming language and the format of statements in a trace subprogram dictate how the instrumentation is done. The format of statements should be chosen to facilitate analysis. For example, a trace subprogram can be expressed as a constrained subprogram as described in Sec. 1.6.

For programs in C language, a trace subprogram may be generated as follows. First, we need to determine the format in which each statement or predicate appears in a trace subprogram. For this purpose, we shall use "TRACE(S) = t" as the shorthand notation for "the trace subprogram of statement S is t". Listed below are TRACE(S) for different types of statement in C language.

(1) Expression Statement

`TRACE(E;) = E` if `E` is an expression statement.

Examples: The trace subprogram of assignment statement `x = 1` is simply `x = 1` itself, and that of `printf("\n")` is `print("\n").`

The syntax of C language allows the use of certain shorthand notations in writing an expression statement. To facilitate symbolic analysis, such notations should be rewritten in full in a trace subprogram as exemplified below:

statement	trace subprogram
`+n;`	`n = n + 1`
`--n;`	`n = n - 1`
`x = ++n;`	`n = n + 1` `x = n`
`x = n++;`	`x = n` `n = n + 1`
`i += 2;`	`i = i + 2`
`x *= y + 1;`	`x = x * (y + 1)`

(2) Conditional Statement

(a) `TRACE(if (P) S) = /\ P`
`TRACE(S)` if P is true,

`TRACE(if (P) S) = /\ !(P)` otherwise.

P is enclosed in parentheses because it may contain an operator with a priority lower that that of the unary operator "!". Incorrect interpretation of P may result if the parentheses are omitted.

Example: The trace subprogram of statement

```
if (c == '\n') ++n;
```

is dependent on the value of c just before this statement is executed. If it is '\n' (new line) then the trace subprogram is

```
/\ c == '\n'
++n;
```

Otherwise it is

```
/\ !(c == '\n')
```

(b) `TRACE(if (P) S1 else S2) =` `/\ P` `TRACE(S1)` if P is true,

`TRACE(IF (P) S1 else S2) =` `/\ !(P)` `TRACE(S2)` otherwise.

(c)
```
TRACE(if (P1) S1
        else if (P2) S2
        else if (P3) S3
             .
             .
             .
        else if (Pn) Sn
        else Sn+1)

     =  /\ !(P1)
        /\ !(P2)
         .
         .
         .
        /\ Pi
        TRACE(Si)
```
if Pi is true for some $1 \le i \le n$,

```
     =  /\ !(P1)
        /\ !(P2)
         .
         .
         .
        /\ !(Pn)
        TRACE(Sn+1)
```
otherwise.

(3) WHILE statement

```
TRACE(while (B) S) =  /\ B
                      TRACE(S)
                      /\ B
                      TRACE(S)
                        .
                        .
                        .
                      /\ B
                      TRACE(S)
                      /\ !(B)
```

Example: For the following statements

```
i = 1;
while (i <= 3)
i = i + 1;
```

the trace subprogram is defined to be

```
i = 1;
/\ i <= 3
i = i + 1;
/\ i <= 3
i = i + 1;
/\ i <= 3
i = i + 1;
/\ !(i <= 3)
```

(4) DO statement

```
TRACE(do S while (B)) = TRACE(S)
                        /\ B
                        TRACE(S)
                        /\ B
                           .
                           .
                           .
                        TRACE(S)
                        /\ !(B)
```

(5) FOR statement

```
TRACE(for (E1, E2, E3) S) = E1
                            /\ E2
                            TRACE(S)
                            E3
                            /\ E2
                            TRACE(S)
                            E3
                            .
                            .
                            .
                            /\ !(E2)
```

Example: The trace subprogram of the following statement

```
for( x = 1; x <= 3; x = x + 1 )
     sum = sum + x;
```

is defined to be

```
x = 1;
```

```
/\ x <= 3
sum = sum + x;
x = x + 1;
/\ x <= 3
sum = sum + x;
x = x + 1;
/\ x <= 3
sum = sum + x;
x = x + 1;
/\ !(x <= 3)
```

(6) SWITCH statement

```
TRACE(switch (c)  {
        case c1: S1;
        case c2: S2;
                 .
                 .
                 .
        case cn: Sn;
        default: Sn+1;})
   = /\ c == ci
     TRACE(Si)
```

if c = ci for some $1 \le i \le n$, assuming that Si ends with a "break" statement,

```
   = /\ c != (c1)
     /\ c != (c2)
         .
         .
         .
     /\ c != (cn)
     TRACE(Sn+1)
```

otherwise.

NOTE: Parentheses around ci's are needed because ci's may contain operators such as "&", "^", and "|", which have a precedence lower than that of "==" and "!=".

(7) BREAK statement

`TRACE(break;)` = empty string
(i.e., need not generate trace subprogram for such a statement)

(8) CONTINUE statement

`TRACE(continue;)` = empty string

(9) RETURN statement

```
(a) TRACE(return;) = return
(b) TRACE(return E;) = return E
```

(10) EXIT statement

```
(a) TRACE(exit;) = exit
(b) TRACE(exit E;) = exit E
```

(11) GOTO statement

```
TRACE(goto LABEL;) = empty string
```

(12) Labeled statement

```
TRACE(LABEL: S;) = TRACE(S)
```

(13) Null statement

```
TRACE( ;) = empty string
```

(14) Compound statement

```
TRACE({declaration-list, statement-list}) = TRACE(statement-list)
TRACE(S1; S2;) = TRACE(S1)
                 TRACE(S2)
```

The instrumentation tool examines each statement in the program, constructs its trace as defined above, assigns an identification number called TN (trace number) to the trace, and stores the trace with its TN in a file. The tool then constructs INST(S), which stands for "the instrumented version of statement S", and writes it into a file created for storing the instrumented version of the program. Production of the trace is done by the program execution monitor pem(). A function call pem(TN(S)) causes the trace of S numbered TN(S) to be fetched from the file and appended to the trace being constructed. The definition of INST(S) is given below:

(1) Expression Statement: if E is an expression then

```
INST(E;)  =  pem (TN(E));
             E;
```

e.g., if 35 is the trace number associated with statement

```
printf("This is a test. \n");
```

then

```
INST(printf("This is a test. \n");)  =  pem (35);
                                        printf("This is a test. \n");
```

(2) Condition Statement

```
INST(if (P) S)  =  if (P) {
```

```
                                    pem (TN(P));
                                    INST(S)
                                    }
                                    else
                                    pem (TN(!(P)));
```

```
INST(if (P) S1 else S2)   =   if (P) {
                              pem(TN(P));
                              INST(S1)
                              }
                              else {
                              pem (TN(!(P)));
                              INST(S2)
                              }
```

Note that "if" statement may be nested, i.e., S1 or S2 above may be another "if" statement. With the instrumentation method described recursively as shown above, we do not need a special rule to deal with nested "if" statements, or to instrument "if" statements of the form (c) given previously.

(3) WHILE Statement

```
INST(while (P) S)   =   while (P) {
                        pem (TN(P));
                        INST(S)
                        }
                        pem (TN(!(P)));
```

(4) DO Statement

```
INST(do S while (P);)   =   _do_?:
                            INST(S)
                            if (P) {
                            pem (TN(P));
                             goto _do_?;
                            }
                            else
                            pem (TN(!(P)));
```

The question mark here will be replaced by an integer assigned by the analyzer-instrumentor. Note that it is incorrect to instrument the DO statement as shown below:

```
                                    do {
                                    INST(S)
                                    if (P) pem (TN(P));
```

```
}
while (P);
pem (TN(!(P)));
```

The reason is that predicate P will be evaluated twice here. If P contains a shorthand or an assignment operator, the instrumented program will no longer be computationally equivalent to the original one.

(5) FOR Statement

```
INST(for (E1; E2; E3) S)   =   pem (TN(E1));
                               for (E1; E2; E3) {
                               pem (TN(E2));
                               INST(S)
                               pem (TN(E3));
                               }
                               pem (TN(!(E2)));
```

(6) SWITCH Statement

```
INST(switch (C)  {
        case C1: S1
        case C2: S2
                  .
                  .
                  .
        case Cn: Sn
    default: Sn+1})
                           =   { int _i_;
                               _i_ = 0;
                               switch (C)  {
                               case C1: if (_i_++ == 0) pem (TN(C == (C1)));
                               INST(S1)
                               case C2: if (_i_++ == 0) pem (TN(C == (C2)));
                               INST(S2)
                               .
                               .
                               .
                               case Cn: if (_i_++ == 0) pem (TN(C == (Cn)));
                               INST(Sn)
                               default: if (_i_++ == 0) {
                               pem (TN(C != (C1)));
                               pem (TN(C != (C2)));
                               .
                               .
                               .
                               pem (TN(C != (Cn)));
```

```
}
INST(Sn+1)
}
}
```

The reason why we use _i_ here is that cases serve just as labels. After the code for one case is done, execution falls through to the next unless one takes explicit action to escape. The flag _i_ is used to ensure that only the condition that is true is included in the trace subprogram.

(7) RETURN Statement

```
INST(return;)  =  pem (TN(return));
                  return;

INST(return E;)  =  pem (TN(return E));
                    return E;
```

(8) EXIT Statement

```
INST(exit;)  =  pem (TN(exit));
                 exit;

INST(exit E;)  =  pem (TN(exit E));
                  exit E;
```

(9) Labeled Statement

```
INST(LABEL: S)  =  LABEL: INST(S)
```

(10) Compound Statement

```
INST({declaration-list statement-list})  =  {
                                            declaration-list
                                            INST(statement-list)
                                            }

INST(S1; S2)  =  INST(S1)
                 INST(S2)
```

(11) Other Statements

```
INST(break;)  =  break;

INST(continue;)  =  continue;

INST(goto LABEL;)  =  goto LABEL;

INST(;)  =  ;
```

Exercise: Define TRACE( ) and INST( ) functions for a programming language of your choice.