Chapter 7
Program Instrumentation

J. C. Huang
Department of Computer Science
University of Houston

Program instrumentation

The main ideal is to insert additional statements (instruments) into the program to be tested for information gathering purposes.

By test executing the instrumented program for a properly chosen set of test cases, we will be able to obtain additional information for error detection and test management.

Possible applications

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

Possible measurements for test coverage:

C1: Each statement in the program is executed at least once.

C2: Each branch in the flowchart is traversed at least once.

C3: Each possible execution path is traversed at least once.

How can it be done?

1) Identify a set of points in the control flow 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).
2) Instrument the program with software counters at these points.
3) Test the program for a set of test cases.
4) Examine the counter values to determine the extent of coverage achieved.

How to build a software counter?

- It can be implemented by using a subroutine (procedure) named, say, count(j).
- This subroutine makes use of an integer array counter[1..n].
- Each element of this array is set to zero initially.
- A call of count(j) causes the value of counter[j] to be increased by 1.

Where to place the counters?

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 outdegree of two or greater. This method is easy to apply; but the number of counters required may not be minimum.

Summary

We can measure the test coverage through instrumentation as outlined above.

The contents of counters also indicate where the optimization pay-off will be greater.

Measuring the effectiveness of a test case

By the effectiveness of a test case 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.

A cause of ineffectiveness

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.

Examples

(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.

Multifaceted expressions

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

Singularity index

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.

Computation of singularity index

- 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.

Example

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

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

Example (continued)

There are three facets in this expression, viz., `a, (c + (d / e)),` and `d`.

A tool can be designed to instrument this expression as follows.

Example (continued)

```
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.

Summary

We can measure the effectiveness of a test case as outlined above.

Observe that, the higher the singularity index, the lower the effectiveness of the test case.

Assertion Checking

The intended function of a program can often be expressed in terms of assertions that must be satisfied, or values that must be assumed by 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 use of a special language

The instruments can be constructed by using the host language. The use of a special language, however, facilitates construction of the instruments and makes it less error prone.

The language of PET

For example, a special high-level language is used in the Program Evaluator and Tester (PET) developed by Stucki. 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.

Inserting instruments as comments

- 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.
- The instruments generally make the program more readable, and thus need not be removed physically.

PET syntax

Local Assertions:

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

PET syntax (continued)

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

Examples:

ASSERT(MOVE .LT. 9) HALT ON 10

PET report

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

Example assertions

ASSERT ORDER (A(*, 3)) ASCENDING

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

Example assertions (continued)

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

NOTE: 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.

Examples of global assertions

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]

Monitors

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

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

Example monitors

MONITOR RANGE FIRST LAST ALL

MONITOR CHARACTER RANGE (XV, YV)

MONITOR RANGE (A(*, 3))

MONITOR SUBSCRIPT RANGE (A, B, C)

Comment

- Assertion checking is potentially an effective means for error detection.
- 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.

Comment (continued)

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.

Comment (continued)

Thus, when a violation is detected, the user has to find out if it is caused by a programming error, or if it is the result of an incorrect assertion. This often is the source of frustration in using assertion checking.

Data flow analysis

During program execution, a statement may act on a variable (datum) in three different ways, viz., *define, reference,* and *undefine*.

Data flow analysis (continued)

A variable is *defined* in a statement if an execution of the statement assigns a value to that variable.

Data flow analysis (continued)

A variable is *referenced* in a statement if an execution of the statement requires that the value of that variable be fetched from memory.

Example

Thus in the assignment statement

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

y and z are both referenced while x is first referenced and then defined.

Data flow analysis (continued)

A variable may become undefined, e.g., when

- a loop is terminated
- the control exit from a block
- when the control exit from a subprogram

Normal sequence of DF events

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.

Abnormal sequence of DF events

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,

we may reasonably conclude that a programming error might have been committed.

Dataflow anomalies

We shall say there is a dataflow anomaly (of *ur*, *du*, or *dd* type, respectively,) if a variable is

- undefined and then referenced,
- defined and then undefined, or
- defined and then defined again.

Detection method - static

A static method for detecting data flow anomalies has been developed by Fosdick et al. 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.

Detection method - static

1. A path expression describes the sequence of actions taken on a variable when the program is executed along the path.

2. The presence of data flow anomalies can thus be detected by examining the constituent components of path expressions.

Detection method - dynamic

It is useful to think of 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,

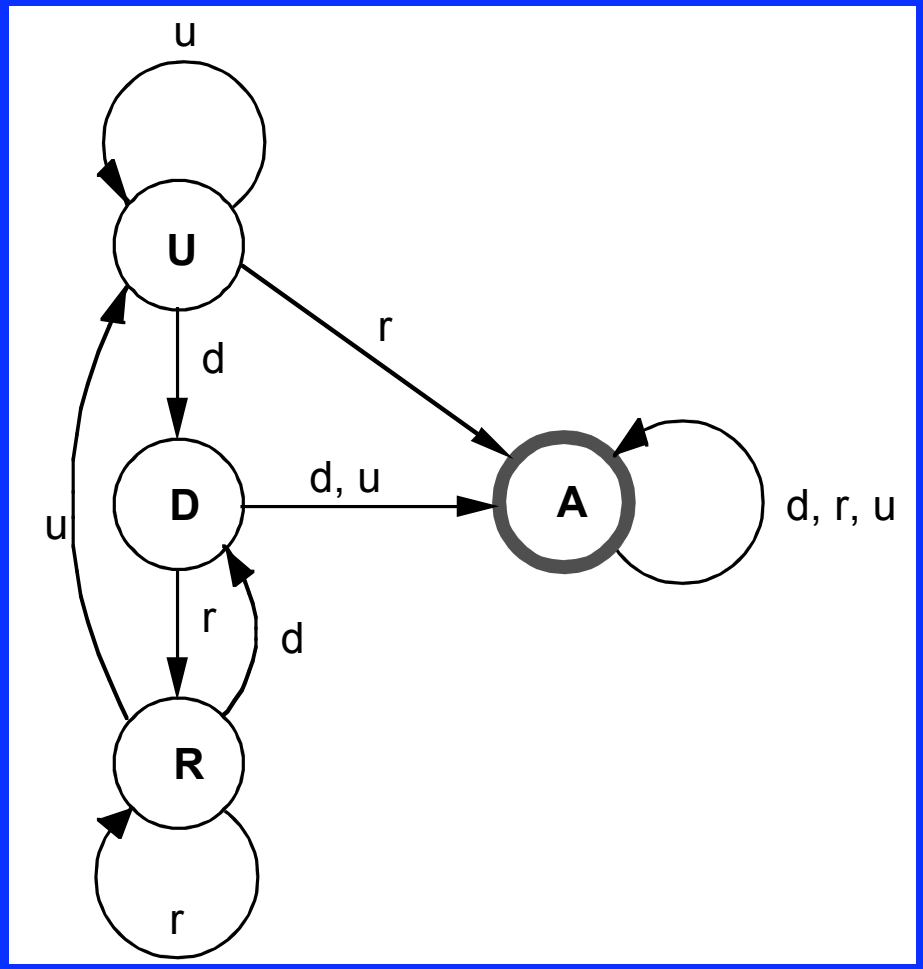
state R: defined and referenced, and

state A: abnormal state

A finite-state machine model

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 next.

The state diagram



The anomalous sequences

Note that each edge in this state diagram is associated with `d`, `r`, or `u`, which stand for "define," "reference," and "undefine," respectively.

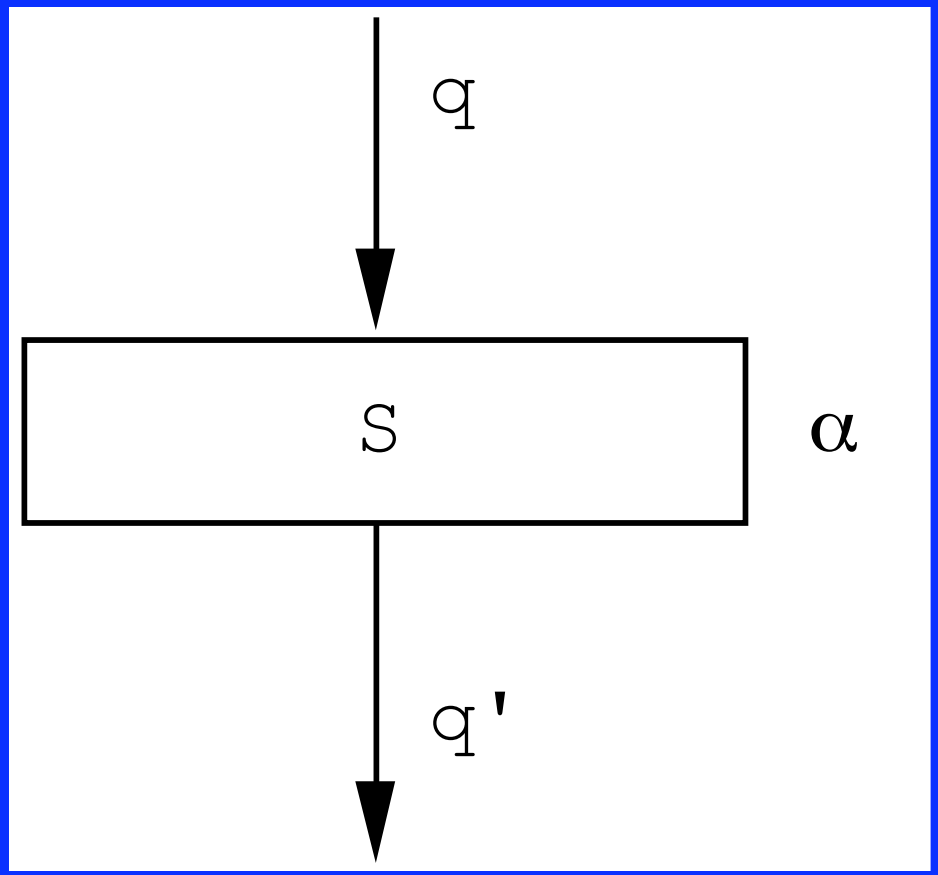
Three types of dataflow anomaly

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.

Detection by monitoring state transition

We need only to know if the sequence of actions contains `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.

Finite state machine model



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

State transition function

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

E.g., $f(U, dur) = f(f(U, d), ur)$

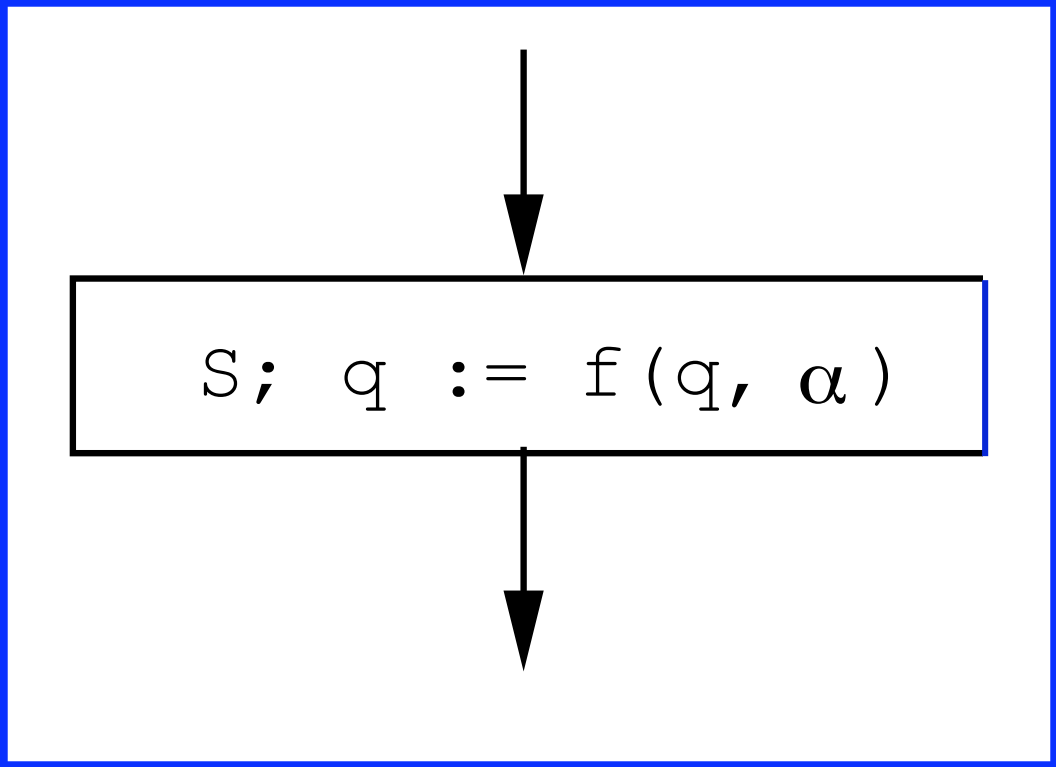
$= f(D, ur)$

$= f(f(D, u), r)$

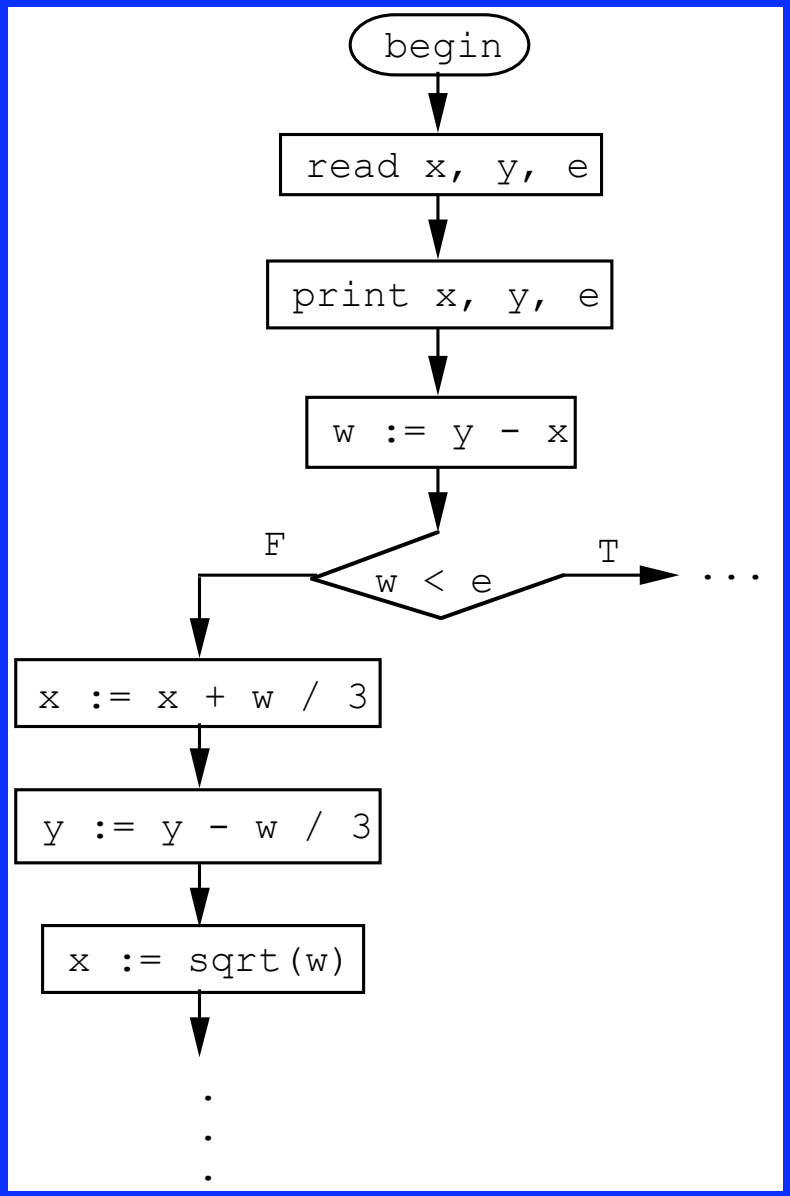
$= f(A, r)$

$= A.$

Instrumented program

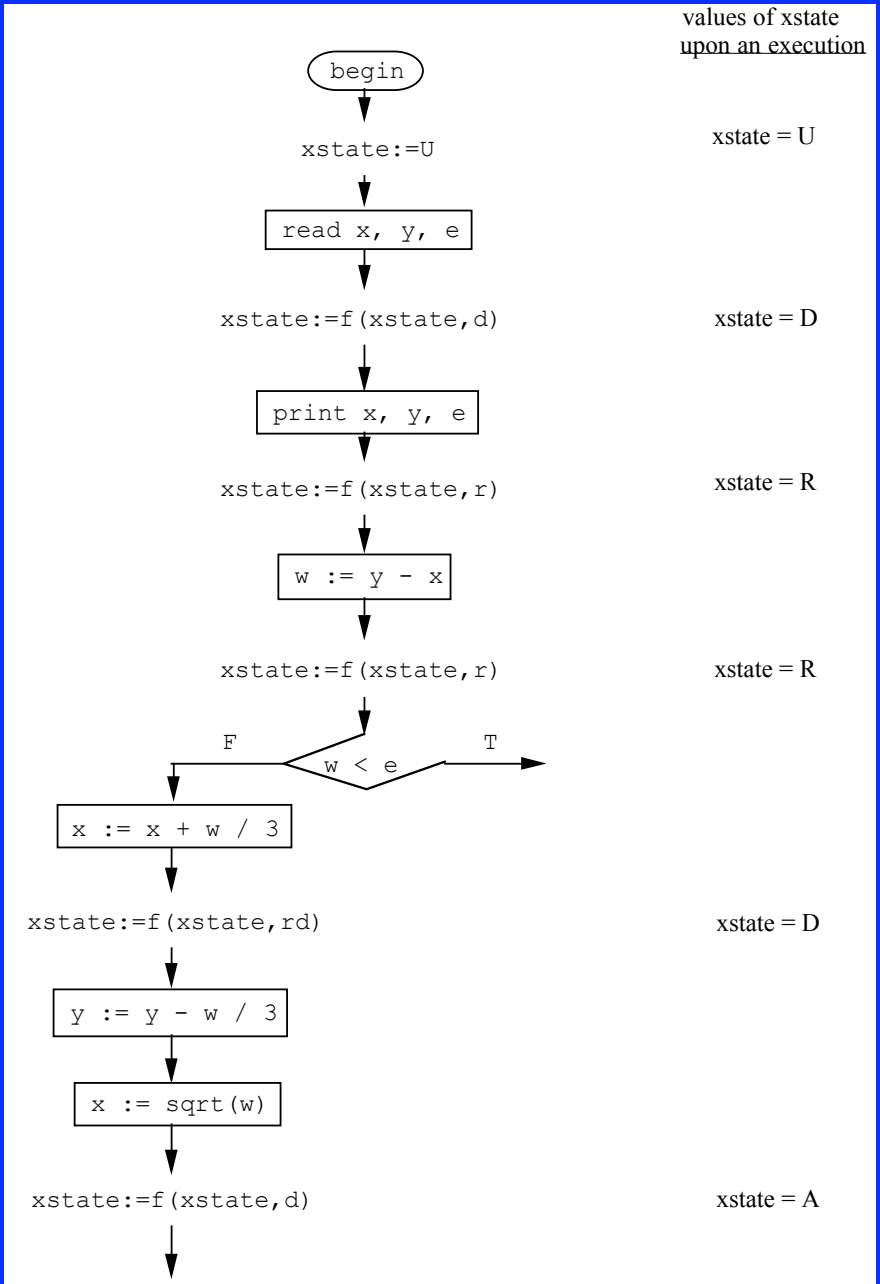


q := f(q, α) is the instrument



An example program

Instrumented program



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.

Data flow of array elements

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.

Possible solution in static analysis

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.

Implication

Consider the familiar sequence of three statements given below:

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

Implication (continued)

It is obvious that the data flow for every variable involved is not anomalous, provided j <> k.

If a[j] and a[k] are created as the same variable, however, the data flow becomes anomalous because it is defined and defined again by the last two statements.

Implication (continued)

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.

Implication (continued)

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

Implication (continued)

For example, consider

```
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.

Conclusion

- Separate handling of array elements is highly desirable.
- The problem posed by array elements can be easily solved if program instrumentation is used.
- The true object of actions can be determined dynamically at the execution time.

Implementation requirements

1. Need 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. Need to instrument the program with statements that will change the state of the right array element at the right place.

Possible solution

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.

Example

For statement like

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

the required instruments would be

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

Test-case selection problem

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?

Possible solution

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.

Notational convention

Let α, β, and γ 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 denoted by λ.

Data flow on a loop

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.

Data flow on a loop (continued)

Thus, if the program is executed along this path, the string representing the sequence of actions taken on x will be of the form $\alpha\beta^n\gamma$.

Data flow on a loop

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 $\alpha\beta^k\gamma$.

Substring relation

We shall use .*substr.* to denote the binary relation "is a substring of".

Thus r .*substr.* rrdru, and ur .*substr.* ddrurd.

A theorem for simple loops

Theorem 7.1: Let α, β, and γ be any nonempty strings, and let τ be any string of two symbols. Then, for any integer $n > 0$,

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

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.

How many iterations are required?

Theorem 7.1 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.

How many iterations are required?

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 $\alpha\beta\gamma$.

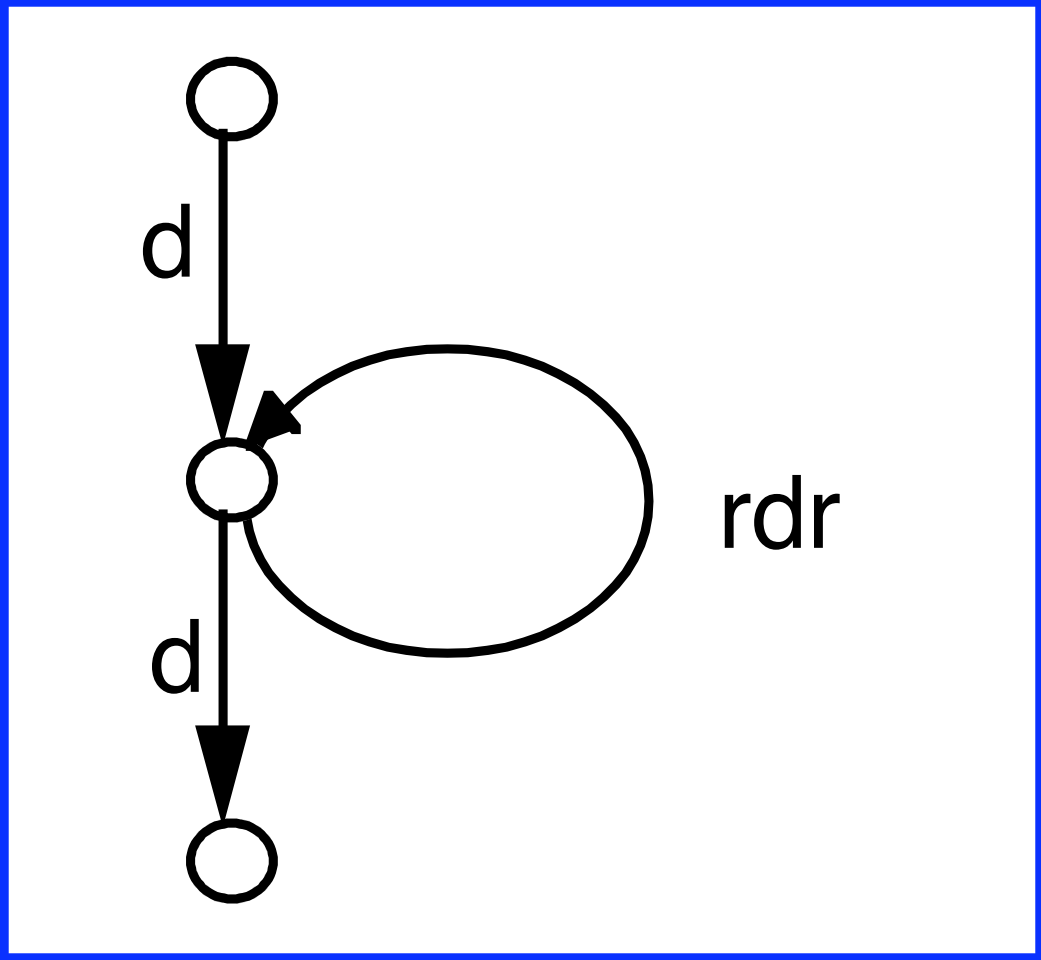
How many iterations are required?

Observe that Theorem 7.1 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 $n > 0$.

How many times… (continued)

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. Next figure exemplifies this type of data flow anomaly.

Example



How many times… (continued)

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.

How many times… (continued)

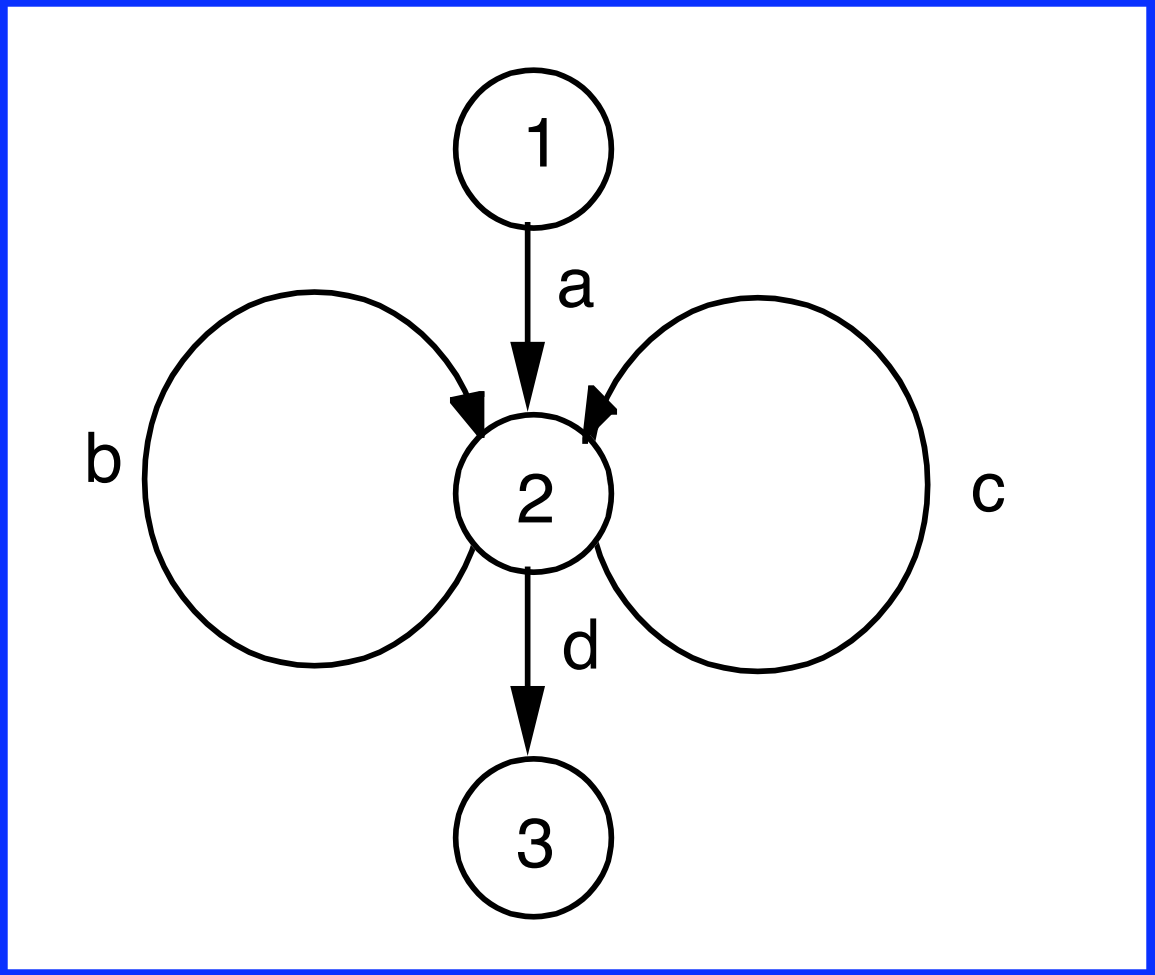
Based on Theorem 7.1 and the above discussion, we can conclude that to ensure detection of all data flow anomalies, each loop in a program has to be iterated zero and two times in execution.

Loop consisting of multiple path

It is not intuitively 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 below, 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.

Multiple-path loop



Multiple-path loop (continued)

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 $(\lambda + p^2)$ to yield the description of the paths that have to be traversed in execution.

Multiple-path loop (continued)

Does the same method apply if p is a description of a set of two or more paths?

Multiple-path loop (continued)

The answer hinges on whether or not we can extend Theorem 7.1 to the cases where β is a set of strings.

It turns out that the answer is affirmative.

Multiple-path loop (continued)

To see why this is so, we shall first restate Theorem 7.1 for the cases where α, β, and γ are sets of strings.

A theorem for multiple-path loop

Theorem 7.2: 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 $\alpha\beta^2\gamma$.

Proof?

Theorem 7.2 is essentially the same as Theorem 7.1 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 7.1 *mutatis mutandis* can be used as the proof of Theorem 7.2.

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.

Summary

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.

How can this be achieved?

Input data selection

Step 1: Find all paths from the entry to the exit in the flow chart of the program.

Step 2: Find a ZT subset of the set of paths found in Step 1.

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.

Comment on the state diagram

The state diagram shown previously 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.

Comment on the state diagram

A possible alternative would be to abort the program execution once a data flow anomaly is detected.

The significance of an anomaly

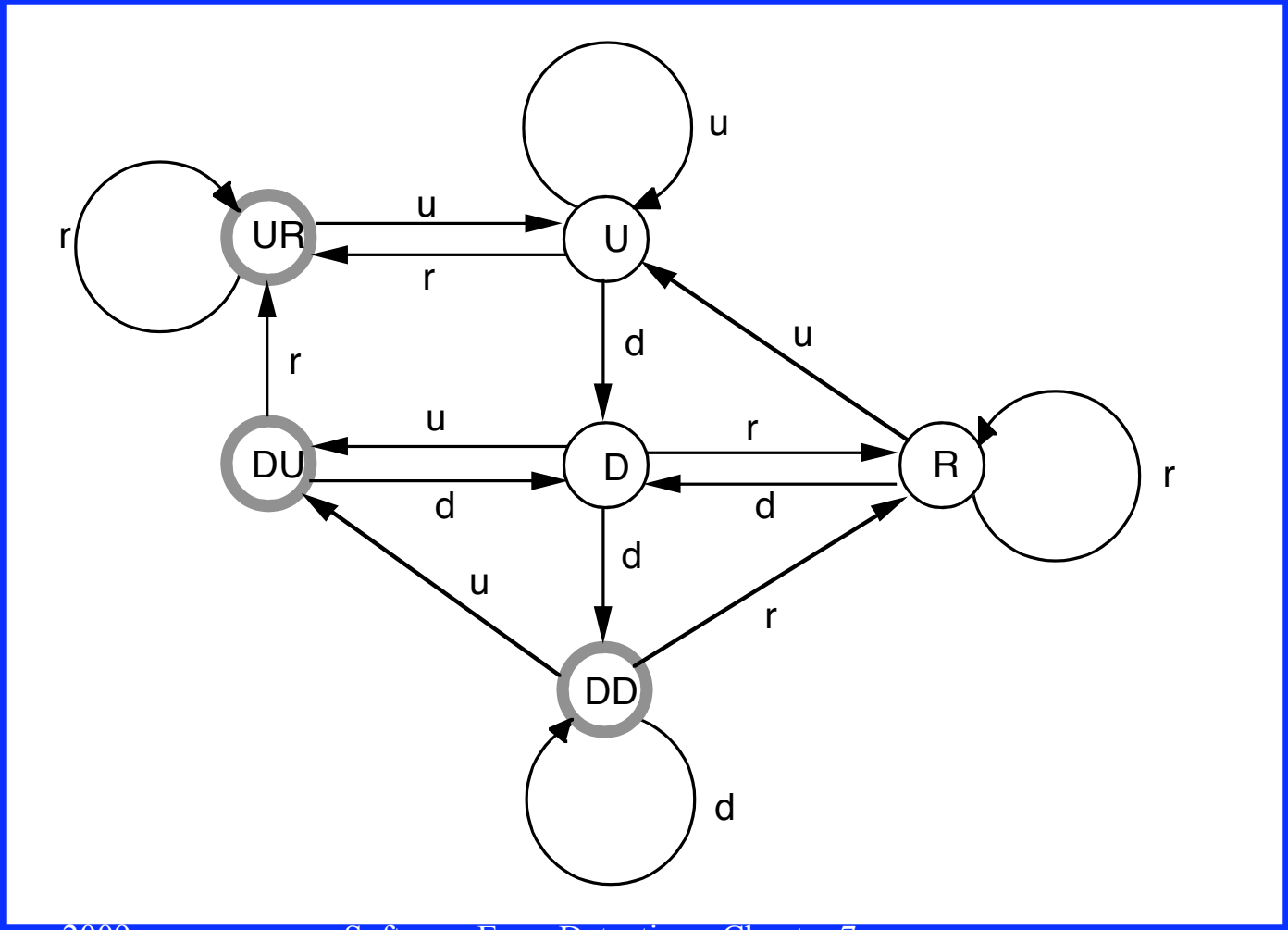
The presence of a data flow anomaly does not imply that execution of the program will definitely produce incorrect results. It implies only that execution may produce incorrect results.

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.

Reset the state (to R)

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, or use the alternative state diagram shown next.

An alternative state diagram



Static vs. Dynamic methods

1. The present method is conceptually much simpler than the static method and, therefore, is much easier to implement.

Static vs. Dynamic methods

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.

Static vs. Dynamic methods

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.

Static vs. Dynamic methods

4. 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.

Static vs. Dynamic methods

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.

Static vs. Dynamic methods

6. 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, 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.

Static vs. Dynamic methods

7. It is difficult to compare the cost. Very roughly speaking, the cost of applying the static method 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 static method 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).

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 it works to produce the output) can be readily determined by examining the trace subprogram associated with that execution path.

Definition

A trace subprogram is a constrained subprogram defined for some execution path in the program.

(see Chapter 5)

Info provided by trace subprogram

- explicitly describes the path along which the program is executed,
- displays the conditions that must be satisfied at various points along the path, and
- clearly describes the computation performed in terms of the statements executed.

Generation of trace subprogram

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.

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.

Conditional statement

(a) `TRACE(if (P) S) = /\ P`

`TRACE(S)` if P is true,

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

Example

The trace subprogram of statement

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

If c is '\n' (new line) then the trace subprogram is

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

Otherwise it is

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

Conditional statement

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

if P is true,

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

otherwise.

Conditional statement

(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 $1 \le i \le n$.

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)
```

DO statement

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

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)
```

Generation of trace subprogram(continued)

Next, 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.

Generation of trace subprogram(continued)

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.

Generation of trace subprogram(continued)

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.

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");`

Conditional statement

INST(if (P) S) =

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

Conditional statement (continued)

INST(if (P) S1 else S2) =

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

WHILE statement

INST(while (P) S) =

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

DO statement

INST(do S while (P);) =

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

Remark

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)));
```

Remark

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.

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)));
```