THE DETECTION OF UNEXECUTABLE PROGRAM PATHS THROUGH STATIC DATA FLOW ANALYSIS*

Leon J. Osterweil
Department of Computer Science**
University of Colorado at Boulder
Boulder, Colorado 80309

and

Boeing Computer Services***
Seattle, Washington 98124

#CU-CS-110-77 May, 1977

* Work supported by NSF Grant #DCR 75-09972 and NASA Contract NAS9-14853

** Address through June 16, 1977

*** Address beginning June 30, 1977

ABSTRACT

An unfortunate characteristic of current static analysis algorithms is their apparent inability to distinguish between executable and unexecutable program paths. The definitive determination of executability of a given path has long been known to be unachievable. This paper presents some heuristics for detecting certain classes of unexecutable paths and preliminary findings tending to indicate that the heuristics can be expected to be rather effective. The heuristics are based upon the application of existing static data flow analysis algorithms and hence offer hope of coexisting with and guiding diagnostic and optimization scans which also use data flow analysis.

I. INTRODUCTION

Static program analysis is a term which describes the process of examining the source text of a program for the purpose of inferring characteristics of the execution behavior of the program specified by the text. Compilers perform static analysis (e.g. the lexical and syntactic scans) as a necessary prelude to the process of generating the object code which effects the specified execution behavior. Object code optimizers, viewed in this light, are seen to be phases of the compilation process which perform more sophisticated static analysis in attempting to draw deeper inferences aimed at the production of more efficient object code. This optimization has typically been aimed at speed and space efficiences [1,2], but more recently has also been focussed on gaining efficiency by recognizing and exploiting parallelism [3].

Static analysis is also used to detect errors in programs. Indeed, some error detection derives quite naturally from the compilation process, as source text which is unrecognizable by the compiler is by definition thereby found to be in error. As might be suspected, however, more powerful error detection capabilities are obtainable as the result of applying more powerful static analysis techniques.

Recently [4] it has been demonstrated that the static analysis techniques used in the most powerful program optimizers also can be used to effect powerful error detection. These techniques are based upon a methodology known as data flow analysis. In data flow analysis the flow of values into and out of program variables is examined as program execution is hypothesized through sequences of statements. Program optimizers use data flow analysis to search for repetitious, redundant, or wasteful data creation and usage sequences, in hopes of being able to synthesize object programs whose input/output behavior replicates that defined by the original source program, but which execute more efficiently than object programs which would be produced by a more mechanical translation of the source text. Error detection and validation systems use data flow analysis to search for erroneous, illegal or suspicious patterns of value creation and usage [5].

Data flow analysis entails the analysis of the program flow graph, a labelled directed graph derived from the program's source text. Each

node of the flow graph represents a different execution unit of the program (e.g. a statement or basic block [6]) and is labelled with information about the variables referred to in the execution unit and the ways in which these variables are used. A directed edge (v_i,v_j) appears in the flow graph if and only if examination of the program source text indicates that execution unit v_j can be executed immediately after execution of unit v_i. Hence it is readily seen that paths through the program flow graph correspond to sequences of execution units, and the corresponding sequences of variable usages. Hence the effects of all possible program executions can be modelled and analyzed by propagating the labels on the program flow graph nodes along paths through the graph to all other graph nodes bearing related labelings. This is the essence of data flow analysis.

From the preceding, it can be seen that data flow analysis operates only on a graph model of the program, and not on the program itself. Often the model can be constructed to represent only a narrow range of program characteristics, pinpointing, for example, a specific but important family of programming errors. In such cases the model generally occupies less space than the program it represents, and the analysis of the model can be completed more rapidly than an execution of the program. These storage and speed efficiencies are often cited as important advantages of data flow analysis (and all static analysis approaches in general). Another important advantage derives from the fact that data flow analysis algorithms examine all possible execution paths through a program. Thus they can either certify that a particular data usage pattern cannot occur during any program execution or they can identify specific paths along which the pattern will occur. These characteristics, crucial to the process of object code optimization, also serve to make data flow analysis a particularly attractive error detection and validation methodology.

Data flow analysis is not without its weaknesses, however. Because its algorithms operate on a model of the program, the validity and efficiency of its results depend upon the accuracy and parsimony of the model. Unfortunately it has been found that some characteristics of program execution behavior (such as the functioning of many real arithmetic units) are difficult to model accurately and efficiently. The process of modelling many important classes of program errors as data

flow anomalies, moreover, is just now being systematically pursued [7,8] and seems to have some significant limitations.

A more fundamental weakness of current data flow anlaysis methodologies, however, is that current algorithms are unable to detect and compensate for the presence of paths through program flow graphs which correspond to unexecutable program sequences. The problem arises because each of the edges of a program flow graph represents a local condition, namely a pair of nodes one of which is reachable from the other. A path through the graph, however, specifies a sequence of these edges, and hence a more complex global condition. In order for any given edge to be traversed during execution of the program, a certain condition, specified by the source text, must be satisfied. It is possible for the conditions specified for two program edges to be mutually incompatible, however. In such a case, the two edges could not be traversed in sequence as part of a single program execution. Hence a path through the flow graph containing these two edges in sequence could not correspond to a possible program execution. Such a path is called an unexecutable path. It should be clear that any procedure which studies all paths through a flow graph of a given program will, for some programs, thus be studying some unexecutable paths. It is possible that erroneous or inefficient situations may be detected along such paths, in which case errors which cannot occur would be reported or effort would be wasted in trying to make unexecutable code more efficient. It is also possible that optimization attempts might be thwarted by needless concern over troublesome aspects of unexecutable program sequences. In one recent study [9], it was observed that approximately 15% of the error messages generated by the DAVE static error detection system [5] reported errors occurring on unexecutable paths. Hence it appears that the usefulness and credibility of static analysis might be increased if unexecutable paths could be detected and removed from consideration by the static analyzer. This paper describes a method for approaching this problem.

Before proceeding with a detailed description of the proposed method, it is important to state some theoretical limitations and other approaches, however. First it must be observed that the problem of determining the executability of a given path through a given program

is unsolvable, being equivalent to the Halting Problem [10]. Specifically the unsolvability of the Halting Problem indirectly implies that it is not possible to construct a procedure which will always correctly determine whether any path through any program is executable. There are, however, heuristic procedures which are capable of determining the executability of some paths through some programs. Hence it becomes important to empirically determine whether heuristics, such as the one described here, reliably determine executability for a significant class of paths and programs. Early experiments seem to indicate that the heuristic described here is effective for significant classes of programs.

The method presented here seems particularly noteworthy because it utilizes algorithms and methodologies of static data flow analysis in order to detect and suppress unexecutable paths. Other methods for determining the executability of a path have been proposed and implemented. The most notable involve attempting to solve systems of predicates as is done in automatic program verification [11] and attempting to solve systems of inequalities in conjunction with symbolic execution [12]. These approaches involve the use of more sophisticated, higher cost methodologies, however, and seem best suited for use as stand-aside executability checkers. The benefits of the proposed method seem to derive from its dependence upon existing data flow analysis techniques. This seems to cause the method to be relatively fast and cheap and raises the possibility of imbedding it as an integral part of existing static error detection and optimization analyzers.

II. AN INTRODUCTION TO DATA FLOW ANALYSIS

As observed earlier, data flow analysis is a term which is used to describe a class of techniques which can determine the extent and nature of the interactions of the various local data manipulations within a program. These techniques generally operate by propagating descriptions about data usage from the nodes at which each usage occurs along paths in the program flow graph to all graph nodes which could conceivably be affected by such a data usage. Data flow anlaysis algorithms are usually used to scan the patterns of data usage along all paths through a program, searching for some specific patterns. In

program optimization, for example, certain patterns of usage indicate the existence of superfluous computations, which can be removed. An example of this is a dead variable -- i.e., a variable which has been assigned a value at a program node, and which is subsequently not referenced at any program node prior to a redefinition or termination of the program. The computations involved in producing the value used to define the variable represent a waste of effort and need not be performed by the program object code. It is important to note that this determination of wasted effort is a conclusion about the overall pattern of usages of a particular variable, and that this global conclusion has been reached through a consideration of the effects of propagating local information about data usage. Further elaboration about the use of data flow analysis in global program optimization can be found in [1].

In program testing and validation, certain patterns of variable reference and definition are considered to be outright errors or symptoms of errors. For example, a reference to a variable which is not preceded by a definition of the variable is regarded to be an outright error, which may, moreover, also be symptomatic of a more subtle underlying error such as a misspelling. Similarly, two consecutive definitions of a variable which are not separated by a reference along any intervening path are regarded as a likely symptom of error. Here too, it is seen that the error and anomaly conditions are detected by studying the effects of propagating information about local data usage to other graph nodes which are impacted by that usage. A more detailed and mathematical treatment of the application of data flow analysis to error and anomaly detection can be found in [4].

Two data flow analysis procedures which will be useful in unexecutable path detection will now be introduced. These two procedures, LIVE and AVAIL, both recognize the existence of only two distinct types of data operations at a given node. The two operations are usually denoted by _gen_ and _kill_. The following notational conventions will be used in this paper.

If the data object, d, is used in a _gen_ operation at node n, this will be indicated by assigning one as the value of the function gen(n,d). If d is not used in a _gen_ operation at n, the value of gen(n,d) will be zero.

Similarly, if the data object, d, is used in a kill operation at node n, this will be indicated by assigning one as the value of the function kill(n,d). If d is not used in a kill operation at n, the value of kill(n,d) will be zero.

Now suppose that a given program contains D data objects indexed from 1 to D. It is useful to define the following Boolean vector valued functions defined on the nodes of the program flow graph:

> GEN(n) is the Boolean D-vector whose i^{th} component is given by gen(n,d), where d is the data object indexed by the integer i.
>
> KILL(n) is the Boolean D-vector whose i^{th} component is given by kill(n,d) where d is the data object indexed by the integer i.

The two Boolean D-vector valued functions GEN(n) and KILL(n) are taken as inputs to the LIVE and AVAIL procedures. Procedure LIVE produces as its output a Boolean D-vector valued function, LIVE(n); procedure AVAIL produces as its output a Boolean D-vector valued function AVAIL(n).

The ith component of LIVE(n) will be denoted by live(n,i), and will be one if and only if there exists a path through the flow graph, p,

$$p = (n, n_1, n_2, \ldots, n_f)$$

such that $kill(n_j,i) = 0$, for all $1 \leq j < f$, and such that $gen(n_f,i) = 1$. In this case it is said that the data object, d, corresponding to this bit position i, is live at node n. If no such path (and pattern of gen's and kill's) exists then live(n,i) will have the value zero and d will be said to be not live or dead at n.

The ith component of AVAIL(n) will be denoted by avail(n,i) and will have the value one if and only if for all paths, p, through the flow graph which start at n_0, the unique program start node, and lead up to n,

$$p = (n_0, n_1, n_2, \ldots, n_f, n),$$

there exists a j, $0 \leq j \leq f$, such that $gen(n_j,i) = 1$ and $kill(n_k,i) = 0$ for all k, $j \leq k \leq f$. In this case, the data object, d, corresponding to bit position i is said to be avail at n. If there exists a path, p, for which the above pattern of gen's and kill's does not exist, then avail(n,i) will have the value zero and d will be said to be not avail at node n.

It is not the purpose of this paper to survey the literature describing algorithms for carrying out the LIVE and AVAIL procedures. There has been considerable effort in this area recently, and the interested reader is encouraged to see [1,4,13,14,15,16] for a representative survey of this effort.

It does seem worthwhile, however, to present here an example algorithm for each of the two procedures to illustrate how they can be carried out and as an aid to a deeper intuitive understanding. The algorithms presented here are perhaps the oldest and most straightforward. They have assumed new importance recently, however, with the finding [15] that the nodes of most program graphs can readily be ordered in such a way as to assure that the algorithms terminate rapidly (generally in O(V) time where V is the number of graph nodes).

Both algorithms assume that _kill_ and _gen_ information is available in the form of the functions KILL(n) and GEN(n), already introduced, and that _live_ and _avail_ information is computed in the form of the functions LIVE(n) and AVAIL(n). Further, it is assumed that the graph nodes are numbered from 1, the start node, to V, the stop node. In the algorithms, moreover, S(j) is used to denote the set of nodes which are successors to node j, and P(j) is used to denote the set of nodes which are predecessors to node j. The D-vector consisting of all zeros is denoted by _0_ and the D-vector of all ones is denoted by _1_.

Proofs of the correctness of both algorithms can be found in [15], along with discussions of their efficiency.

It is quite important to note that the LIVE and AVAIL procedures compile information about the patterns of _gen_'s and _kill_'s on the nodes of paths leading into and out of fixed flow graph nodes without regard to the rule used in determining how the values of _gen_ and _kill_ are initially obtained. By creating different rules for intializing the _gen_ and _kill_ values and by placing different interpretations on the resulting _live_ and _avail_ values, different data flow analysis problems are created and solved.

```
PROCEDURE:  LIVE

   FOR j = 1 TO V;
      LIVE(j) ← 0;
   END
   CHANGE ← TRUE;
      WHILE CHANGE = TRUE DO
      CHANGE ← FALSE;
      FOR j = 1 TO V;
         PREVIOUS ← LIVE(j);
         FOR all successor nodes of j, S_{j,k}
         LIVE(j) ← LIVE(j) ∪ ((LIVE(S_{j,k}) ∩ (1-KILL(S_{j,k}))) ∪ GEN(S_{j,k}) ;
         END
         IF PREVIOUS ≠ LIVE(j)
            THEN CHANGE ← TRUE;
         END
      END
   END
END:  LIVE

------------------------------------------------------------------------

PROCEDURE:  AVAIL
   AVAIL(1) ← 0;
   FOR j = 2 TO V;
      AVAIL(j) ← 1
   END
   CHANGE ← TRUE;
   WHILE CHANGE = TRUE DO
      CHANGE ← FALSE;
      FOR j = 2 TO V;
         PREVIOUS ← AVAIL(j);
         FOR all predecessor nodes of j, P_{j,k}
            AVAIL(j) ← AVAIL(j) ∩ ((AVAIL(P_{j,k}) ∩ (1-KILL(P_{j,k})) ∪ GEN(P_{j,k});
         END
         IF PREVIOUS ≠ AVAIL(j);
            THEN CHANGE ← TRUE;
         END
      END
   END
END:  AVAIL
```

For example, in [4] it is shown that references to uninitialized variables can be detected by judicious application of the LIVE and AVAIL algorithms. Suppose that, for all nodes n, which reference the variable indexed by i, gen(n,i) is set to 1, and that for all nodes n which define a value for the variable indexed by i, kill(n,i) is set to 1. Suppose that all other values of gen(n,i) and kill(n,i) are set to zero and LIVE is run. Then if 1 is the program start node and LIVE(1,i) is 1, there exists a path through the program flow graph which, if executed, will cause the variable indexed by i to be referenced before definition.

Further, suppose that instead of the above labelling, the following labelling were employed: gen(1,i) is set to 1 for all i, $1 \leq i \leq D$, and kill(n,i) is set to 1 for all nodes, n, at which the variable indexed by i is defined. Now, if all other values of gen(n,i) and kill(n,i) are set to zero and AVAIL is executed, then if avail(n,i) is found to be 1, then whenever node n is executed for the first time the variable indexed by i will be referenced and found to uninitialized. It is important to emphasize that these algorithms are highly parallel, enabling the simultaneous detection of uninitialized references to any variable at any node with only a single execution of LIVE or AVAIL. Yet the worst case time bound on these executions is $O(V^2)$, with $O(V)$ being the ordinary expectation (see [4,13])

Related problems concerning uninitialized variable reference and superfluous variable definition can readily be posed in terms of _gen_'s and _kill_'s and solved by the use of LIVE and AVAIL also. The interested reader is referred to [17]. Moreover _gen_'s, _kill_'s, _live_'s, and _avail_'s also seem useful in modeling and detecting other error and validation phenomena such as zero divisor checks, out of bounds subscript checks, and writing to unopened files (see [7,8]).

III. DETECTION OF UNEXECUTABLE PATHS USING DATA FLOW ANALYSIS

The previous section has shown that data flow analysis algorithms can be useful in detecting the existence of flow graph paths along which various error or anomaly conditions may arise. These algorithms, employed in more classical optimization applications, are used to identify classes of paths along which wasteful and inefficient computations occur. In all cases, however, the paths studied are flow graph

paths. Hence some of the identified error and waste phenomena may occur on unexecutable paths and are thus ephermeral and not worthy of further consideration. In this section we present some heuristics for detecting some classes of these unexecutable paths which appear to be significant. These heuristics rely upon the use of the LIVE and AVAIL algorithms.

All of the heuristics entail a search of the flow graph for edges which are rendered unexecutable by the execution of other parts of the flow graph. This can be viewed as an elaboration of the work on impossible pairs, first advanced in [18]. The notions of pathwise impossible pairs (PIP's) of edges, unconditionally impossible pairs (UIP's) of edges, and always unexecutable edges (AUE's) shall be advanced as elaborations here.

It is important to precede this discussion by noting that each edge in a program flow graph represents a transition from one execution unit to another which could be made under certain conditions. Hence each edge may be tagged by the condition which, when met upon exit from its tail node, will necessarily cause the traversal of the edge and subsequent execution of its head node. This condition can be expressed as a predicate, i.e., a Boolean function mapping some subset of the set of program variables onto either of the values _true_ or _false_. For a given edge, e, this function will be denoted by R(e). Further, S(e) will be used to denote the subset of the set of program variables upon which R(e) is explicitly defined. Some examples of how these R(e) and S(e) are evolved are the following:

1. For an edge, e, representing a GO TO or a simple sequence between consecutive code units

   R(e) = _true_; S(e) = $\emptyset$

2. For the edge, e, representing the _true_ exit from a FORTRAN logical IF statement represented by

   IF(B)X; where B is a Boolean expression
   R(e) = B; S(e) = {variables explicitly named in B}

3. For the edge, e, representing the _false_ exit from the above logical IF, R(e) = $\neg$B; S(e) = {variables explicitly named in B.}

4. For e, the fall-through edge out of a DO loop begun by the statement

   $$\text{DO L P} = I_1, I_2, I_3 \; ; \; (\text{if } I_3 > 1)$$

   $R(e) = P > I_2 + I_3$; $S(e) = \{P\}$

5. For e, the fall-through edge out of a DO loop begun by the statement

   $$\text{DO L P} = I_1, I_2 \; ; \; R(e) = (P = I_2) \; ; \; S(e) = \{P\}$$

It is readily seen that, in an ordinary program, there exist pairs of predicates $R(e_1)$, $R(e_2)$ such that $R(e_1) \wedge R(e_2) \equiv$ false. Such a pair of predicates will be referred to as a mutually inconsistent pair. The knowledge of which pairs of predicates are mutually inconsistent will be most important.

It is now possible to explain the notion of a pathwise impossible pair of edges. A pathwise impossible pair of edges (PIP) is defined to be a pair of edges (e_i, e_j) of a program flow graph for which there exists a path p from the head of e_i to the tail of e_j such that the sequence of execution units along the path e_i, p, e_j cannot be executed in sequence under any circumstances. A PIP, (e_i, e_j) would be present in a program, for example, in which $R(e_i) \wedge R(e_j) \equiv$ false and no node of some path p between e_i's head and e_j's tail resets any of the variables in $S(e_i)$ or $S(e_j)$. In this case, the conditions which caused the execution of edge e_i would still be in existence after traversing p to preclude the execution of e_j. An example of a program displaying such a PIP is shown in Figure 1. In a subsequent section, it shall be shown that such PIP's are detectable by suitable labelling of the flow graph, application of the LIVE algorithm, and interpretation of the results.

The detection of PIP's seems important because, as Figure 1 illustrates, only some paths between e_i and e_j may be executable, and only some may exhibit error or waste phenomena. In general a correlation between the anomalous and unexecutable paths should not be expected. As Figure 1 shows, however, an error condition may exist on some detectably unexecutable paths, as well as on some paths which appear exectuable. In such a case, once this has been determined (through identification of PIP's) the error pheonomenon should

```
SUBROUTINE SUB1(X,S1,S2)
IF(X.GT.0) S1 = S1 + 1
IF(MOD(S1,2).EQ.0) X = -X
IF(X.LE.0) S2 = S2 + 1
TOT = TOT + 1
RETURN
END
```

Figure 1a) A FORTRAN subroutine having a data flow anomaly (TOT is referenced before definition) and pathwise impossible pairs of edges (PIP's).

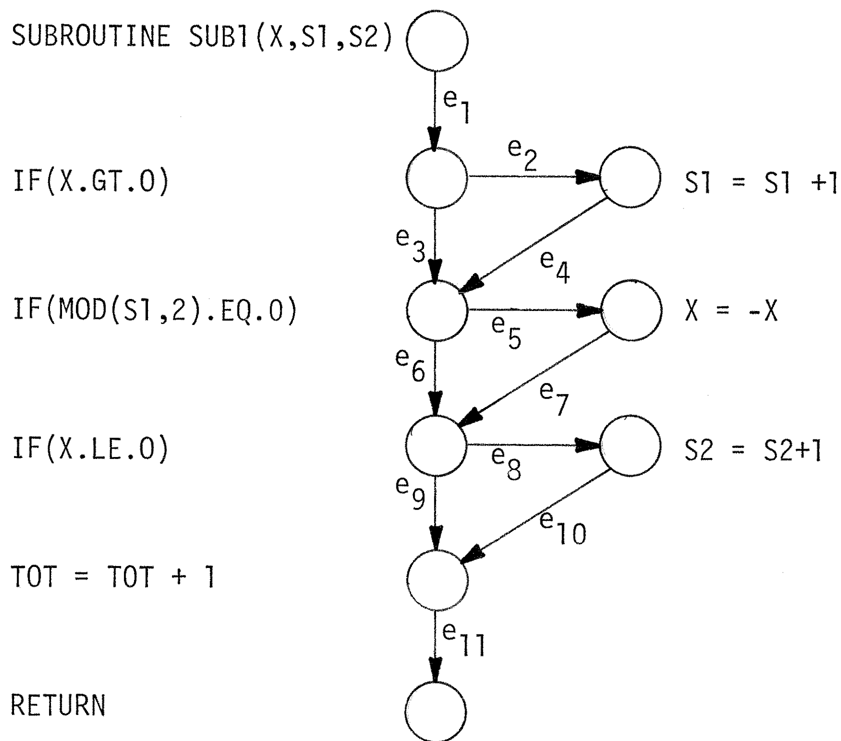

Figure 1b) The flow graph of the subroutine in Figure 1a). The edge pairs (e_2,e_8) and (e_3,e_9) are PIP's. The undefined variable reference occurs on the unexecutable path $e_1e_3\ e_6e_9$ and on the executable paths $e_1e_2e_4e_6e_9$; $e_1e_2e_4e_5e_7e_8e_{10}$.

preferably be reported and related to a path which is not detectably unexecutable. Analogously, if wasteful computation sequences are observed only on unexecutable paths, optimization should not be attempted.

An unconditionally impossible pair of edges (UIP) is defined to be a pair of edges (e_i, e_j) of a program flow graph such that the execution of edge e_j is not possible if the most recent execution of e_i's head resulted in the traversal of e_i. It should be noted that all UIP's are also PIP's. A UIP, (e_i, e_j) would be present in a program, for example, in which $R(e_i) \wedge R(e_j) \equiv$ false and no node on any path between e_i's head and e_j's tail resets any of the variables in $S(e_i)$ or $S(e_j)$. In this case it is seen that the conditions which caused the execution of edge e_i must necessarily still be in existence when e_j's tail is encountered, precluding the possibility of executing e_j. An example of a program displaying such a UIP is shown in Figure 2. In a subsequent section it shall be shown that UIP's are detectable by suitable labelling of the flow graph and applications of LIVE and AVAIL algorithms.

The detection of UIP's seems important because through identification of UIP's whole classes of flow graph paths can be ruled out as execution possibilities. Hence, for example if the first edge of a UIP is associated with an error phenomenon, then an example path exhibiting this phenomenon must be constrained to omit the second edge of the UIP. It may be possible to demonstrate that all example paths necessitate the second edge. In such a case, the error pheonomenon is shown to be unexecutable, and reports of the error to the user should be suppressed.

An always unexecutable edge (AUE) is defined to be an edge of a program flow graph which can never be traversed. An AUE, e, would be present in a flow graph in which there existed another edge e' such that $R(e) \wedge R(e') \equiv$ false, e' lies on all paths from the start node of the program to e (e' is then said to dominate e) and no node on any path from e' to e resets the value of any variable in S(e) or S(e'). In this case the tail node of e cannot be reached unless e' has been traversed, but that assures that conditions precluding the traversal of e are in effect when the tail of e is reached. An unexecutable fall-through edge out of a FORTRAN DO-loop provides an example of an AUE (see Figure 3). In a subsequent section it shall be shown that AUE's can be detected through the use of the AVAIL procedure.

```
SUBROUTINE SUB2(X,S2)
IF(X.GT.0) S1 = 1
IF(X.LE.0) S1 = 0
S2 = S1
END
```

Figure 2a) A FORTRAN subroutine having two data flow anomalies (S1 is defined twice without an intervening reference on some path; and S1 is referenced before definition on some path) and unconditionally impossible paths (UIP's)

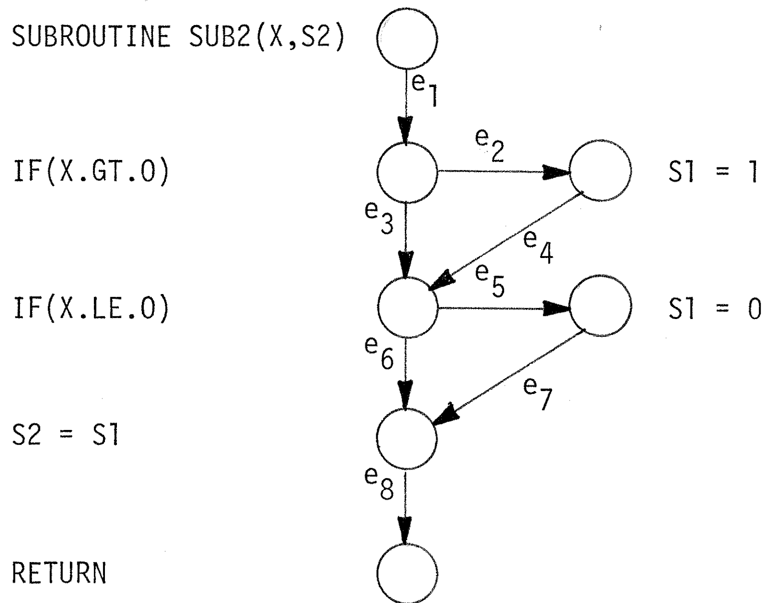

Figure 2b) The flow graph of the subroutine in Figure 2a). The edge pairs (e_2,e_5) and (e_3,e_6) are UIP's. The double definition error occurs on the path $e_1e_2e_4e_5e_7e_8$ and the undefined variable reference error occurs on the path $e_1e_3e_6e_8$. Both errors are seen to be unexecutable because of their dependence on UIP's, and hence they should not be pursued or reported to the user.

```
      SUBROUTINE SUB3(X,LOC)
      DIMENSION X(100)
      DO 10 I = 1, 100
      IF(X(I).EQ.0) GO TO 20
      IF(I.LT.100) GO TO 10
      LOC = 0
      GO TO 30
   10 CONTINUE
   20 LOC = I
   30 RETURN
      END
```

Figure 3a) A FORTRAN subroutine having an apparent data flow anomaly (I is referenced after becoming undefined on the fall-through edge out of the DO-loop) and an always unexecutable edge (AUE).

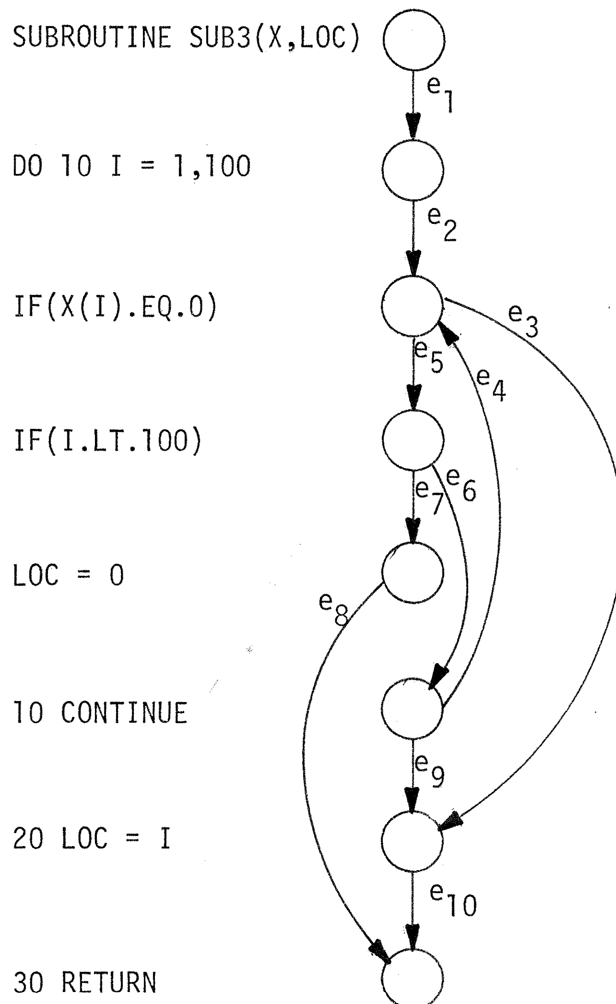

Figure 3b) The flow graph of the subroutine in Figure 3a). Edge e_9 is an AUE. The undefined variable reference cannot occur unless e_9 is traversed. Hence no error message should be transmitted to the user.

The detection of AUE's seems most important because all AUE's can and should be removed from the flow graph before any diagnostic or optimization scans begin. In this way the scans are prevented from ever considering large classes of unexecutable paths. PIP's and UIP's seem useful in censoring out unexecutable paths presented for consideration by diagnostic or optimization procedures. AUE's can be used to assure that these procedures do not consider, and hence cannot present, certain of these paths.

IV. PROCEDURES FOR DETECTING PIP's, UIP's AND AUE's.

In this section procedures employing LIVE and AVAIL to detect certain classes of PIP's, UIP's and AUE's will be presented. These procedures all require the hypothesis of gen and kill functions, but also require that information about mutually inconsistent edge predicates be available. Hence the following definitions are made: Assume that the edges of the flow graph are indexed from 1 to E. Define inc(i,j) to be one if and only if $R(e_i) \wedge R(e_j) \equiv$ false where e_i is the edge whose index is i and e_j is the edge whose index is j; inc(i,j) is defined to be zero otherwise.

Define INC(i) to be the Boolean E-vector whose j^{th} component is given by inc(i,j).

Hence INC(i) is a Boolean vector which indicates those edges whose predicates are mutually inconsistent with the predicate on edge i by having a one set as the value of those components of INC(i) which correspond to those edges having inconsistent predicates.

It is important to stress that the determination of the values of the E-vectors INC(i) is not a trivial matter; being ultimately an unsolvable problem. Approaches to studying that problem will not be dealt with here, but can be found in [19,20]. It should be noted, however, that most programs in common use seem to have rather simple predicates on the predominant majority of their edges [21,22]. For pairs of such predicates, the determination of the correct value for inc(i,j) can usually be made quite easily. For other pairs where the determination is harder, it shall be seen that the safe course is to set the value of inc(i,j) to zero. Fortunately, the results in [21,22] indicate that difficulty in determining the correct value of inc(i,j) should not arise frequently.

It is now possible to state a procedure for detecting PIP's. Suppose that, for a given program, the set of data objects is the set of all predicates on the E edges of the program flow graph. Suppose that the flow graph is annotated by creating an E-vector, GEN(e), for each of the E edges of the graph and an E-vector, KILL(n), for each node of the N nodes of the graph by means of the following procedure:

```
PROCEDURE INIT_PIP
   FOR e = 1 TO E DO
      GEN(e) ← 0
      gen(e,e) ← 1
   END
   FOR n = 1 TO N DO
      KILL(n) ← 0
      T(n) ← {variables reset at n}
      FOR e = 1 TO E DO
         IF T(n) ∩ S(e) ≠ ∅ THEN kill(n,e) ← 1
      END
   END
END INIT_PIP
```

Then execute:

```
PROCEDURE PIP
   CALL INIT_PIP
   CALL LIVE1*
   FOR e = 1 TO E DO
      PIP(e) ← LIVE(e) ∧ INC(e)
   END
END PIP
```

Now, if the jth bit of PIP(e_i) is 1 then it has been shown that (e_i,e_j) is a PIP.

---

* It is actually not possible to run LIVE on the graph annotated exactly as above, because the GEN information is attached to edges rather than nodes. Some thought, however, should persuade the reader that only minor adjustments to the LIVE procedure are necessary in order to create LIVE1, its analog for graphs annotated as in INIT_PIP. One possible line of reasoning might be to create one new, synthetic node to be inserted onto each edge, and to consider that each such node has as its sole function the task of bearing the GEN information previously associated with the edge on which the node has been placed.

The procedure for detecting UIP's operates by determining which PIP's are UIP's as well. For each PIP, the procedure attempts to find a path from the head of the first edge of the PIP to the tail of the second edge of the PIP which goes through a node which resets the value of a variable in a predicate on one of the edges of the PIP. Hence this procedure manipulates P-vectors, where P is the number of PIP's discovered by procedure PIP.

The procedure has three main preparatory phases: 1) the nodes which can potentially destroy the UIP property, by altering a variable in a PIP are identified by creating a function, KILLERS; 2) a variant of LIVE, LIVE2*, is used to determine which nodes of the flowgraph are at the start of paths to second-edges of which PIP's; 3) a variant of AVAIL, AVAIL2*, is used to determine which nodes can never be reached subsequent to traversal of first-edges of which PIP's. At each node this information is examined to see for which PIP's there is a KILLER node at the end of a path from the PIP's first-edge and which is at the head of a path to the PIP's second-edge. All such PIP's cannot be relied upon to be UIP's; the other PIP's, however, must be UIP's.

```
PROCEDURE UIP
   CALL PIP
   COMMENT: First phase - determine KILLERS
   P ← 0
   FOR e = 1 TO E DO
      FOR f = 1 TO E DO
         IF bit f of PIP(e) = 1 THEN P ← P + 1
                                     A(P) ← e
                                     B(P) ← f
                                END
      END
   END
   FOR n = 1 TO N DO
      T(n) ← {variables reset at n}
      KILLERS(n) ← 0
      FOR p = 1 TO P DO
         IF R(n) ∩ (S(A(p))) ∪ S(B(p))) ≠ ∅ THEN KILLERS (n,p) ← 1
      END
   END
```

```
    COMMENT second phase - set up and execute LIVE2
    FOR e = 1 TO E DO
       GEN(E) ← 0
       KILL(E) ← 0
    END
    FOR p = 1 TO P DO
       GEN(B(p),p) ← 1
       KILL(A(p),p) ← 1
    END
    CALL LIVE2*
    COMMENT: LIVE2(n,p) is now one if node n "sees" B(p) before A(p)
    COMMENT: third phase - set up and execute AVAIL2*
    FOR e = 1 TO E DO
       KILL(e) ← 0
    END
    FOR p = 1 TO P DO
       KILL(A(p),p) ← 1
    END
    COMMENT: assume the start node has index 1
    FOR n = 2 TO N DO
       GEN(n) ← 0
    END
    CALL AVAIL2*
    COMMENT: AVAIL2(n,p) is now zero if node n is reachable
             from A(p)
    COMMENT: now use the above to compute NIP, a function iden-
             tifying nonunconditionally impossible pairs, and UIP
    AND ← 0
    FOR n = 1 TO N DO
       NIP(n) ← (AVAIL2(n) ∧ KILLERS(n)) ∧ LIVE2(n)
       AND ← AND ∧ NIP(n)
    END
    UIP ← 1 ∧ ¬AND
    COMMENT: if the p^th bit of UIP is one, then (A(p), B(p))
             is a UIP.
END UIP
```

* Here too LIVE and AVAIL, as described in an earlier section cannot be used directly here because some labels are attached to edges. As in the case of LIVE1, the necessary modifications are not difficult to envision.

The procedure for detecting AUE's involves only a rather straightforward application of the AVAIL procedure. The procedure is initialized by first executing INIT _ PIP.

```
PROCEDURE  AUE
   CALL INIT _ PIP
   CALL AVAIL1*
   AUE ← 0
   FOR e = 1 TO E DO
      IF (AVAIL1(e) ∧ INC(e)) ≠ Ø THEN
                                     COMMENT e is AUE;
                                             bit e of AUE ← 1;
                                     END
   END
END  AUE
```

Procedure AUE could be made more powerful if the gen and kill vectors were initialized somewhat differently. Suppose that the set of data objects used by AUE was chosen to be the set of different predicates in the program (AUE currently considers each edge to bear a different predicate - hence no use is made of the fact that a program flowgraph may have the same predicate on more than one edge). GEN(e) would still contain exactly one bit set to one, but now bit i of GEN(e) would be set to one for all edges e bearing predicate i. As before KILL(n) would have a one in every position corresponding to a predicate having a variable which was reset at node n. After running AUE with this initialization, additional edges might be found to be unexecutable. Specifically an edge e might be unreachable without prior traversal of e' or e", where R(e') = R(e"), R(e') ∧ R(e) ≡ false, and no variable in S(e) ∪ S(e') ∪ S(e") is reset at any node on any path either from the head of e' to the tail of e or from the head of e" to the tail of e. In this case e is clearly an AUE. The original AUE procedure would not detect this because neither e' nor e" dominates e. The improved AUE procedure would correctly determine that e is an AUE.

---

* Here too LIVE and AVAIL, as described in an earlier section cannot be used directly because some labels are attached to edges. As in the case of LIVE1, the necessary modifications are not difficult to envision.

In closing, it is important to discuss the execution time bounds of these algorithms. It has already been noted that LIVE, LIVE1, LIVE2, AVAIL, AVAIL1 and AVAIL2 are all $O(V^2)$ algorithms in the worst case, where bit-vector operations are considered to take unit time [4,15]. For ordinary program flow graphs, moreover, a time bound of $O(V)$ should be expected. Examination of algorithms PIP, UIP, and AUE seems to indicate that the process of creating the bit vectors used by LIVE, AVAIL and their variants is the important factor in computing execution time bounds for PIP, UIP and AUE.

Procedure INIT_PIP shows that creating the KILL vectors may take $O(E*V)$ time in the worst case, although methods exploiting the far less difficult usual cases can easily be devised. Hence, nevertheless, PIP must be considered an $O(E*V)$ algorithm in the worst case.

The initialization phases of UIP show that an $O(E^2)$ loop may be needed to build all P PIP's. In the expectation that $P<<E^2$, a more efficient procedure for building A and B, the P-vectors needed by UIP, can readily be devised, reducing this to an $O(P)$ process. Nevertheless, in the worst case construction of A and B is $O(E^2)$. Hence, in the worst case construction of A and B is $O(E^2)$. Hence, in the worst case construction of KILLERS must correspondingly be $O(V*E^2)$. This is the longest process in UIP, hence UIP's worst case execution time is $O(V*E^2)$.

Analysis of procedure AUE is similar to the analysis of procedure PIP, showing that the worst case execution time for AUE is $O(V*E)$.

Finally, it should be stressed that a single execution of each of these algorithms is sufficient to determine not one, but all PIP's, UIP's and AUE's which are detectable by these algorithms. Hence the time bounds just stated are sufficient for the parallel production of all PIP's, UIP's and AUE's to which the methods described in this paper apply.

V. EVALUATION OF THESE ALGORITHMS AND FUTURE WORK

As noted earlier the algorithms proposed here are heuristics, and hence their ultimate value can best be determined by implementation, experimentation and observation. Preliminary studies, mostly based on hand simulation of the algorithms, have shown that approximately 15%

of the error and anomaly messages produced by DAVE [4,5] for a broadly selected set of subprograms were tied by DAVE to unexecutable paths. In virtually all of these cases, the unexecutabilities were due to PIP's which would have been detected by the algorithms described here. The hand simulations further indicated that the analytic results produced by the algorithms described here could have been used to produce executable paths displaying the detected errors and anomalies in nearly all cases. Algorithms for using the results of the PIP algorithm for detecting the unexecutability of example paths and guiding the selection of executable counterparts appear in [9]. More efficient versions of these algorithms are currently being sought and studied.

The hand simulations seemed sufficiently encouraging that the implementation of the algorithms described here has been undertaken. These implementations are being incorporated into the DAVE system to enable experimentation and the gathering of statistics. Statistics about the relative numbers of PIP's, UIP's, and AUE's detected will be important measures of the significance of these algorithms and will be reported in [9].

The accurate and efficient determination of the INC vectors, embodying information about mutually inconsistent pairs, seems to be of crucial importance to the viability of these proposed algorithms. As already noted, it is not possible to write a procedure for infallibly determining when two predicates are mutually inconsistent. Hence the INC vectors cannot be expected to always be totally correct.* Moreover, because these are E INC vectors each of length E, the initialization of these vectors requires at least $O(E^2)$ time. Hence, comparing this with the time bounds stated at the end of section IV, it becomes clear that any test for mutual inconsistency must be quite straightforward, lest this phase of initialization become the dominant factor in the execution times of procedures PIP, UIP and AUE. Fortunately, the hand simulations carried out bore out the results to be expected from [21,22], namely

---

* A consequence of this is that PIP, UIP, and AUE cannot be expected to always accurately determine the executability of all paths in all programs. This is in agreement with the already-stated implications of the Halting Problem.

that mutually inconsistent predicates found in actual code usually involved no more than one variable, one constant, and one relational operator.

Because of these considerations a number of heuristics seem indicated in determining mutual incompatibility. First, two predicates cannot be mutually incompatible unless they are defined on identical sets of variables. Hence this test, easily made by comparing set-membership bit vectors, seems to be the indicated first step. If both predicates are defined in terms of the same single variable, they might then be cast into a canonical form, simplified, and compared. The comparison in such a situation could be done by consideration of a small, but exhaustive set of special cases dictated by the relational operators used in the two predicates in canonical form. Predicate pairs not conforming to these criteria would be declared not-incompatible. It is important to note, however, that although this might be an erroneous conclusion at times, it always errs on the safe side. This is because refusing to declare predicate pairs mutually incompatible unless that is certainly the case, assures that pairs will never be declared PIP's and UIP's incorrectly, and edges will never be declared AUE's incorrectly. PIP's, UIP's and AUE's may be missed because of these heuristics, but successively more will be correctly identified as successively stronger heuristics are brought to bear. Determination of appropriately powerful heuristics is another important goal of the experimental implementation currently in progress.

Other PIP's, UIP's and AUE's are also not detected by these procedures because mutual incompatibility of predicates is determined only on the basis of strict lexical incompatibility of the predicates. Other incompatibilities could be determined by carrying out a simulated execution of source text segments using the techniques of symbolic execution, and then using these results to determine the true relationships of predicate pairs to each other, where predicates would be considered functions of program input values only. In the extreme case this process would be tantamount to complete symbolic execution or verification of the program. The greater power of these approaches is achieved at correspondingly greater expenditures of time and money. Ideally these other methodologies should be considered, along with static analysis, to be components of an overall Integrated Analysis Strategy such as

outlined in [7] in which static analysis is carried to its logical conclusion, and unresolved questions, such as executability of paths bearing complicated predicates, are saved for subsequent higher powered modules such as symbolic executors. In the context of such a more grandiose scheme it is not unreasonable to consider employing limited amounts of symbolic execution in the determination of unexecutable paths during static analysis. Here too, however, experimentation should be used to guide the decision about how much symbolic execution is reasonably incorporated into the essentially static procedures described here.

Finally, work should be done on determining the applicability of these algorithms and procedures to the detection of unexecutable paths across procedure boundaries. Work such as [4,16] indicates that interprocedural data flow analysis is feasible. Interprocedural analysis applied to unexecutable path detection also warrants investigation.

VI. ACKNOWLEDGEMENTS

This work is the logical outgrowth of two research activities in which the author has been involved. The first activity is the work of the Software Validation Group at the University of Colorado and the production of the DAVE Static Analysis System. This work was supported by National Science Foundation Grants GJ-36461 and DCR75-09972. The second activity was work done for the Network Analysis Techniques Study carried out by TRW Defense and Space Systems Group at Houston for NASA/Johnson Space Center under NASA Contract No. NAS9-14853. The author is indebted to Bob Hoffman of TRW, Houston for his help in clarifying the Impossible Pairs concept which is central to this work, and to Lloyd Fosdick for his encouragement, helpful criticism, and perception about the applicability of the data flow analysis approach. The author is also deeply appreciative of the efforts of Lee Bollacker in asking challenging questions and working through the hand simulations which provided the impetus for carrying this work forward.

REFERENCES

[1] Schaeffer, M., A Mathematical Theory of Global Program Optimization, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[2] Allen, F. E., "Program optimization," Annual Review in Automatic Programming, Pergamon, New York, 1969, pp. 239-307.

[3] Lamport, L., "The Parallel Execution of DO Loops," CACM 17, (February 1974) pp. 83-93.

[4] Fosdick, L. D. and Osterweil, L. J., "Data Flow Analysis in Software Reliability," ACM Computing Surveys, 8 pp. 305-330, (September 1976).

[5] Osterweil, L. J. and Fosdick, L. D., "DAVE--A Validation, Error Detection, and Documentation System for FORTRAN Programs," Software--Practice and Experience, 6,pp. 473-486 (September 1976).

[6] Allen, F. E., "A basis for program optimization," Proceedings IFIP Congress 1971, North-Holland Publishing Co., Amsterdam, 1972, pp. 385-390.

[7] Osterweil, L. J., "A Proposal for an Integrated Testing System for Computer Programs," University of Colorado Department of Computer Science Technical Report No. CU-CS-093-76.

[8] Osterweil, L. J., "New Applications of Data Flow Analysis to Static Error Detection and Validation," University of Colorado Department of Computer Science Technical Report (to appear, 1977).

[9] Bollacker, L. "Some Experiments in Using Data Flow Analysis to Detect Unexecutable Paths Through Programs," University of Colorado Department of Computer Science Masters Thesis (to appear, 1977).

[10] Hopcroft, J. E. and Ullman, J. D., Formal Languages and Their Relation to Automata, Addison Wesley, Reading, Mass., 1969, pp. 108-109.

[11] King, J. C., "Symbolic Execution and Program Testing", CACM 19, pp. 385-394 (July 1976)

[12] Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, SE-2, pp. 215-222, (September 1976).

[13] Ullman, J. D., "Fast algorithms for the elimination of common subexpressions," Acta Informatica 2 (1973) pp. 191-213.

[14] Kennedy, K. W., "Node listings applied to data flow analysis," Proceedings of 2nd ACM Symposium on Principals of Programming Languages, Palo Alto, California (January 1975) pp. 10-21.

[15] Hecht, M. S. and Ullman, J. D., "A simple algorithm for global data flow analysis problems," SIAM J. Computing 4 (December 1975), pp. 519-532.

[16] F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure", CACM 19 pp. 137-147 (March 1976).

[17] Osterweil, L. J., "Data Flow Analysis in Detection of Uninitialized Variables and Editing of Impossible Pairs," TRW Defense and Space Systems Group, Houston, Texas; Contract #NAS9-14853 (21 January 1977).

[18] Krause, K. A., R. W. Smith and M. A. Goodwin, "Optimal Software Test Planning Through Automated Network Analysis," 1973 IEEE Symposium on Computer Software Reliability, IEEE Cat. #73C40741-9CSR New York, pp. 18-22 (June 1973).

[19] Hoffman, R. H., "ATDG Impossible Pairs Detection Capability Study Final Report," TRW 77:2511.3-17,(20 January 1977).

[20] Elspas, B., K. N. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness", ACM Computing Surveys 4, pp. 97-147 (June 1972).

[21] Knuth, D. E., "An Empirical Study of FORTRAN Programs," Software--Practice and Experience, 1, pp. 105-135 (1971).

[22] Elshoff, James L., "A Numerical Profile of Commerical PL/I Programs," Software--Practice and Experience, 6, pp. 505-526, (October, 1976).