DAVE-HAL/S: A SYSTEM FOR THE STATIC DATA FLOW ANALYSIS OF SINGLE-PROCESS HAL/S PROGRAMS

by

Carol Drey
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CS-CU-141-78 November 1978

This work was supported by grant NSG 1476 from NASA-Langley Research Center and NSF Grant Number MCS77-02194.
Abstract

The application of data flow analysis algorithms to improve the reliability of single-process HAL/S programs is described. These algorithms evolved from the development of a system, DAVE, designed to analyze Fortran but are basically language-independent and therefore adaptable to HAL/S. An overview of the DAVE-HAL/S system is presented and features of HAL/S requiring special attention are discussed. The anomalies detected by the system are listed, followed by a high-level description of the system components. Since concurrency is a major feature of HAL/S, the last section indicates work in progress to adapt the analysis to handle concurrent-process as well as single-process programs.
Introduction

DAVE-HAL/S is a tool for the static data flow analysis of HAL/S programs. The system described here is designed to detect the presence of data flow anomalies in single-process programs and is patterned after the DAVE system for FORTRAN [1]. DAVE-HAL/S, however, employs the faster, more efficient algorithms described in the Fosdick-Osterweil article, "Data Flow Analysis in Software Reliability" [2], which were developed after the completion of DAVE-FORTRAN. The term "DAVE" will be used here to refer to those elements of the analysis which are language-independent, while "DAVE-FORTRAN" and "DAVE-HAL/S" will be used to refer to specific DAVE systems.

This report first presents a summary of the ideas behind a DAVE system. For a more detailed description of the algorithms, the reader is referred to [2]. Section II discusses special features of HAL/S which affected the design of DAVE-HAL/S. The third section lists the anomalies detected by DAVE-HAL/S and their relationship to error conditions detected by the Intermetrics HAL/S compiler [3]. The fourth section consists of a high-level description of the data flow analysis system and the final section looks at work in progress to extend the analysis to concurrent process programs.

I. Concepts and Capabilities

DAVE-HAL/S is designed to detect anomalous data flow patterns, symptomatic of programming errors, not only along paths within function and procedure blocks but along paths which cross block boundaries as well. The algorithms used to detect these patterns of variable usage employ two types of graphs to represent execution sequences of a program. The first, a flow graph, is used to represent the flow of control from statement to statement within a subprogram unit. Since nodes in a flow graph reflect the control structure of a program, there may not be a one-to-one correspondence between nodes in the flow graph and the source statements in a unit. The correspondence between HAL/S statement and node types is discussed in [4]. Note that while a statement containing a procedure or function invocation is represented as a single node, that node actually represents all the data actions which occur in-
side the called block. Because of the order in which DAVE processes subprogram units, the data flow information in the called unit can be passed across the block boundary without placing its control structure at the point of invocation in the caller. To utilize the flow graph, DAVE's analysis requires the following information for each node: (1) its predecessors; (2) its successors; (3) its node type; (4) a representation of the statement or statement part associated with it; and (5) an index specifying the sequential number of that associated statement.

The other type of graph used is the call graph, which has the same form as a flow graph, but its nodes represent subprogram units and its edges indicate invocation of one unit by another. The call graph is used to guide the analysis of the units comprising a program in an order referred to as "leafs-up." The leaf subprograms, which invoke no others, are processed first; then those units which invoke only processed units are analyzed in a backward order with the main program block being processed last. In order to use this procedure, the call graph must be acyclic. Therefore, before DAVE-HAL/S begins analysis, it checks the call graph for cycles, indicating recursion, which is illegal in HAL/S. If it finds any cycles, it issues a message and terminates.

At the core of the data flow analysis is the idea of sets of variables called "path sets," which are associated with nodes in the flow graph. Membership of a variable in a path set for a node indicates that a particular sequence of data actions on that variable occurs at the node. The three possible actions are reference, define, and undefine. For statements containing no procedure or function invocations, determination of path set membership is straightforward. For instance, for the assignment statement, \( \alpha = \alpha + \beta \), associated with a node \( n \), \( \alpha \) and \( \beta \) will be placed in those path sets which represent a reference as the first data action at \( n \). \( \alpha \) will also be placed in those path sets representing an arbitrary sequence of actions followed by a definition. A variable \( \gamma \) appearing in the same subprogram, would be placed in the path set representing no action upon the variable at node \( n \).

Let us consider a leaf subprogram. Once the path sets have been
determined for the nodes in its flow graph, the path sets for the unit as a whole can be constructed using the algorithms described in [2]. The same procedures are followed whether analyzing variables declared in the unit or global to it. For formal parameters and global variables, the path sets are used for passing variable usage information across block boundaries and are saved in a master table as each unit is analyzed. At the same time as these path sets for the unit as a whole are created, additional path sets are formed for each node reflecting what sequences of data actions occur entering and leaving that node. By intersecting path sets representing sequences of actions entering (or leaving) the node and occurring at the node, anomalous data flow patterns are detected. The three types of anomalies found in this manner are:

1. a reference to an uninitialized variable
2. two definitions of a variable with no intervening reference
3. failure to subsequently reference a variable after defining it

When a non-leaf subprogram is analyzed, path set membership is determined as for a leaf with this exception: When a function or procedure invocation is encountered at a node, path set information must be passed from the invoked block to this node. First the path sets for the invoked block as a whole are retrieved from the master table. Then the actual arguments are placed in the same path sets as their corresponding formal parameters. This is also done for any global variables which are members of the path sets for the invoked block. Thus, the data actions which occur in the invoked procedure are reflected in the path sets for the node containing the invocation. Other than this, the analysis follows the same steps as outlined for a leaf unit.

In addition to the aforementioned anomalous path detection, the analysis performed by DAVE provides information which may be used for program documentation. This includes the order in which code blocks may be invoked, which variables need to be assigned values before entry to a block and which variables are assigned values there, as well as the side effect data flow of global variables as a result of an invocation of the block.
Although DAVE-HAL/S was designed for the HAL/S language, there should be no problem in employing it to analyze HALMAT code. Since HALMAT contains the same data and control flow information as HAL/S, it can be mapped onto the same type of flow graph as HAL/S and the same type of variable usage information can be gathered from it -- the two basic requirements for the operation of DAVE.

II. Special Considerations for HAL/S

Some special features of HAL/S and their impact on the design of DAVE-HAL/S are discussed in this section.

1. Data types: Three types of subscripting - array, structure, and component - may be applied to the HAL/S data types MATRIX, CHARACTER, VECTOR, ARRAY, STRUCTURE. As in DAVE-FORTRAN, any action performed upon one element of a subscripted data item is considered to be performed on the data item as an entity.

2. Temporary variables within DO...END statement groups: We assume that each data item contains a unique entry in a symbol table. Therefore, the name of a temporary data item may be treated as a local variable for the block containing the DO...END.

3. In-line functions: Since a data item declared in an in-line function will have a unique symbol table entry, it may be treated as local to the containing block.

4. % macros and REPLACE: We assume that DAVE-HAL/S works on the expanded text of a program.

5. SUBBIT pseudo-conversion function: This is the only conversion function which may appear in an assignment context as well as in an expression. When used in an assignment context, its argument is assigned a value as a result of its invocation. When used in a referencing context, its argument is considered referenced, as with the other conversion functions.
6. (a) NAME facility: When a NAME data item is used as an ordinary data item (indirect accessing), all one can say is that it is being referenced, and nothing at all about the data item it points to.

(b) NAME pseudo-function: This function is used to reference or modify pointer values. In a referencing context, if the argument is an ordinary data item, one or more pointers to it are created. For DAVE's purpose, this is not considered a reference to the ordinary data item. If the argument is a NAME data item, it is considered referenced. The appearance of a NAME pseudo-function in an assignment context causes the argument, which may only be a NAME data item, to be defined.

III. Anomalies Detected by DAVE-HAL/S

Of those anomalies detected by DAVE-FORTRAN that are not FORTRAN specific, there are two that are detected by the Intermetrics HAL/S compiler: (1) the mismatch of actual arguments and formal parameters in type or dimensionality, and (2) an indication of variables which are declared but never used.

Anomalous data flow which DAVE-FORTRAN does detect which the HAL/S compiler does not are, as described earlier:

(1) a reference to an uninitialized variable
(2) a definition of a variable followed by another definition of that variable with no intervening reference
(3) a definition of a variable with no subsequent reference to that variable

These three conditions are language-independent and their detection is useful for verifying HAL/S code.

The HAL/S compiler does detect an uninitialized variable reference but only if the variable never appeared in an assignment context. It does no path analysis. Neither does it do any analysis across procedure boundaries, as DAVE does. It will flag a formal input parameter which
is never referenced inside its block but it misses the fact that an actual input argument corresponding to that parameter is also not referenced at the point of invocation: Because it is an input argument, the compiler considers that a referencing context. For the same reasons, the compiler will also not detect several forms of side effects, e.g., an actual input argument being assigned a value via its global name inside an invoked procedure. A complete list of the anomalies detected by DAVE-HAL/S follows:

1. An input argument which is never referenced inside the called function or procedure is detected at the point of invocation.

2. An assign argument which is never assigned a value inside the called function or procedure is detected at the point of invocation.

3. A reference to a variable which has not been initialized on some or all paths to the reference. This anomaly is reported for all program variables.

4. Two definitions of a variable with no intervening reference on some or all paths between the definitions. This anomaly is reported only for simple local and global variables.

5. A definition of a variable occurring as the last action upon the variable in the program unit. This is reported for local variables declared AUTOMATIC in procedure or function blocks and for all local and COMPOOL variables at the main program level.

6. The non-usage of a variable is detected in the block containing the variable's declaration. Either the variable is never used or the variable name appears as an actual argument to a function or procedure but is never actually referenced or defined there. The HAL/S compiler cannot detect the second condition because it does no path analysis; it can only detect a variable which is declared but never appears again.

7. Possible side effect situations, which may occur when:

   NOTE: None of these situations is illegal in HAL/S, but the results obtained are either dependent upon the parameter passing mechanism of the implementation as in (a)
and (b), an unusual data usage (c), or the order of evaluation of an expression (d).

(a) the same data item appears both as an input argument and as an assign argument in a procedure invocation, e.g.,

```
CALL PROC(A) ASSIGN(A);  # PROC:PROCEDURE(X) ASSIGN(Y);
   DECLARE INTEGER,X,Y,Z;
   Y = 1;
   Z = X;
```

Since the HALS/S Language Specification [5] states that input arguments may be call-by-value or call-by-reference, the value that will be assigned to Z in this example is implementation dependent.

(b) a global variable, appearing as an actual input argument is also assigned a value as a global variable by the invoked procedure or function, e.g.,

```
P:PROGRAM;
DECLARE INTEGER,A,B;
   DECLARE X INTEGER;
   B = B+1;
   A = X;
```

Note that if a global variable is an actual assign argument and is referenced by that name inside the invoked block, the result is well-defined since assign arguments are specified as call-by-reference. However, it may still be useful to document this type of data flow.

(c) the same variable appears more than once in the list of assign arguments in a procedure invocation, e.g., CALL PROC ASSIGN (A,A). Again, the result is well-defined because assign arguments are call-by-reference, but the user should be made aware of the side effect.

(d) evaluation of a function alters the value of a data item which appears elsewhere in the expression, right-hand-side of an assignment statement, or CALL statement in which the function invocation appears, e.g.,

```
DECLARE INTEGER,A,B,X, FUN:FUNCTION INTEGER(X);
   DECLARE X INTEGER;
   A = 2;
   B = A+FUN(X);
   A = A+1;
   RETURN X+1;
   CLOSE FUN;
```
8. A statement which is not a RETURN but whose execution immediately precedes the CLOSE statement in a function block. This is only reported if there exists a path from the start node to this statement.

9. The presence of recursion in a HAL/S program.

IV. Design of DAVE-HAL/S

This section contains a high level description, written in PDL[6], of the flow segments comprising the DAVE-HAL/S data flow analysis system. These segments require access to certain data aggregates described in Section I: a call graph, a flow graph for each subprogram unit, and a master table containing an entry for each unit. In addition, these segments assume the availability of a symbol table containing a unique entry for each data item in the program and lists of local variables and global variables for each subprogram unit.
Tree of Inter-Segment References

Page Segment

11. Data flow analysis driver
12. Process call graph for cycles and leafs-up ordering
13. Depth first search(v)
14. Form path sets for template
15. Build path sets
19. Place (expression variables) in (X) path sets
22. Pass path set membership of (parameter) to (argument)
23. Report non-usage of (argument)
24. Report side effects involving (argument)
26. Pass path set membership of global variables over block boundary
27. Determine path sets for entire block
31. Number flow graph in postorder
32. Search and number(v)
33. Execute LIVE on graph
34. Execute AVAIL on graph
35. Report local anomalies
36. Determine anomalous paths of type ur, dd, du
38. Find a path that contains anomaly (direction, xy, frequency, node) for (variable)
SEGMENT Data Flow Analysis Driver

Process call graph for cycles and leafs-up ordering
IF cycles are present in call graph
    Output message ("Illegal recursion in program")
    Stop
ENDIF

DO for each block in leafs-up order
    IF block is specified by template
        Form path sets for template
        Make entry in master table
        Output message ("Template used for (block name)")
    ELSE
        Get flowgraph for block
        Build path sets
        Report local anomalies
        Report global anomalies
        Make entry in master table
    ENDIF
ENDDO
END
SEGMENT Process call graph for cycles and leafs-up ordering

*****************************************************************************************************************************************************************************************************************************
**  This segment determines the presence of cycles in the call graph (indicating recursion) and determines the postorder numbering of the call graph, which is the leafs-up order in which the subprogram units will be analyzed.
*****************************************************************************************************************************************************************************************************************************

Initialize TREE to Ø

DO for all nodes v in call graph
    PREORDER(v) = 0
ENDDO

i = 0
j = 0

Depth first search(entry node)

*** Check for backedges ***
cycles = false

DO for each node v in graph
    DO for each node w on v's exit list
        IF edge (v,w) ∈ TRFF
            IF w ≤ v < w + DESCENDANTS(w)
                cycles = true
        ENDDIF
    ENDDIF
ENDDO
ENDDO
END
SEGMENT Depth first search(v)

This segment performs a depth first search on a directed graph, numbers
the nodes in preorder and postorder and determines the depth
first spanning tree and the number of descendants for each node in
that tree.

\[
i = i + 1
\]
\[
\text{PREORDER}(v) = i
\]

DO for each node \( w \) on \( v \)'s list of exit nodes
\[
\text{IF} \ \text{PREORDER}(v) = 0
\]
\[
\text{Add} \ (v,w) \ \text{to TREE}
\]
\[
\text{Depth first search}(w)
\]
\[
\text{ENDIF}
\]
ENDDO

\[
\text{DESCENDANTS}(v) = i - \text{PREORDER}(v) + 1
\]
\[
j = j + 1
\]
\[
\text{POSTORDER}(v) = j
\]
END
SEGMENT  Form path sets for template

**************************************************************************************************************
* *
* This segment places all input parameters in referencing path sets
* and assign parameters in defining path sets for procedure and function
* blocks represented by templates.
* *
**************************************************************************************************************

Set up and initialize path sets: \( A_x, B_x, C_x, D_x, E_x, F_x, G, I \) for \( x = r, d, u \)

DO for each input formal parameter
    Enter (parameter) in \( (A_r) \)
    Enter (parameter) in \( (C_r) \)
ENDDO

DO for each assign formal parameter
    Enter (parameter) in \( (D_d) \)
    Enter (parameter) in \( (F_d) \)
    Enter (parameter) in \( G \)
ENDDO

END
SEGMENT Build path sets

Initialize $G(\text{graph}) = \emptyset$

* Path set $G$ has been added to the path sets described in [2] to be able to identify variables which are defined anywhere in the unit.

DO for each node in flow graph

Initialize $SIDEFCT = REF = DEF = \emptyset$

* Used for detection of side effects

Set up and initialize path sets: $A_x(\text{node})$, $B_x(\text{node})$, $B_x(\text{node})$, $C_x(\text{node})$, $D_x(\text{node})$, $E_x(\text{node})$, $F_x(\text{node})$, $I(\text{node})$ for $x = r, d, u$

DO case of node type

Assignment:

Place (right-hand-side) in (referencing) path sets
Place (left-hand-side) in (defining) path sets

CALL:

Place (arguments) in (referencing) path sets

DO initialization:

Place (loop variable) in (defining) path sets
Place (variables in initial value expression) in (referencing) path sets

DO successor:

Place (variables in successor expressions) in (referencing) path sets
Place (loop variable) in (referencing) path sets
Place (loop variable) in (defining) path sets
DO test:
  Place (variables in conditional expression) in (referencing) path sets

DO case:
  Place (variables in expression) in (referencing) path sets

IF:
  Place (variables in conditional expression) in (referencing) path sets

Program entry:
  DO for all variables declared in program block
    IF variable appeared with initialization attribute
      Place (variable) in (defining) path sets
    ELSE
      Place (variable) in (undefining) path sets
    ENDF
  ENDDO
  DO for all COMPOOL variables
    IF variable appeared with initialization attribute
      Place (variable) in (defining) path sets
    ELSE
      Place (variable) in (undefining) path sets
    ENDF
  ENNDFO

Procedure or function entry:
  DO for all variables declared in this block
    IF variable appeared with initialization attribute
      Place (variable) in (defining) path sets
    ELSE
      Place (variable) in (undefining) path sets
    ENDF
  ENDDO
CLOSE program:
    DO for all variables declared in program block
    Place (variable) in (undefining) path sets
ENDDO
    DO for all COMPOOL variables
    Place (variable) in (undefining) path sets
ENDDO

CLOSE procedure or function:
    DO for all variables declared AUTOMATIC in this block
    Place (variable) in (undefining) path sets
ENDDO

RETURN:
    Place (variables in expression) in (referencing) path sets

READ:
    Place (expression variables) in (defining) path sets

WRITE:
    Place (expression variables) in (referencing) path sets

FILE input:
    Place (variable on left-hand-side) in (defining) path sets
    Place (variables in right-hand-side file expression) in (referencing) path sets

FILE output:
    Place (variables in left-hand-side file expression) in (referencing) path sets
    Place (variables in right-hand-side expression) in (referencing) path sets
Other:
   Ignore
ENDDO

\( C_X (\text{node}) = C_X (\text{node}) \cup A_X (\text{node}) \)
\( F_X (\text{node}) = F_X (\text{node}) \cup D_X (\text{node}) \)
\( I (\text{node}) = \{ \text{all variables} \} - (A_X (\text{node}) \cup B_X (\text{node}) \)
\( \cup C_X (\text{node}) \cup D_X (\text{node}) \cup E_X (\text{node}) \cup F_X (\text{node}) \)
\( G (\text{graph}) = G (\text{graph}) \cup C_d (\text{Node}) \cup F_d (\text{node}) \)

IF SIDEFCT \( \neq \) empty
   DO for each variable in SIDEFCT
      Output message ("A possible side effect has been detected in this statement involving (variable)")
   ENDDO
ENDDO
ENDIF
ENDDO
ENDDO

Determine path sets for entire block

END
SEGMENT Place (expression variables) in (X) path sets

******************************************************************************
* * This recursive segment processes data item tokens -- which may
* * be variables or references to procedures or functions -- in expres-
* * sions and places the variables in the appropriate path sets.
* * X is either referencing, defining, or undefining
******************************************************************************

DO for each token in expression
   Initialize TEMPREF, TEMPDEF to Ø
   DO case of token type
      Built-in or conversion function (other than SUBBIT) name:
         DO for each argument
            Place (argument) in (referencing) path sets
         ENDDO

SUBBIT pseudo-conversion function:
   IF X is referencing
      Place (argument) in (referencing) path sets
   ELSEIF X is defining
      Place (argument) in (defining) path sets
   ENDDIF

NAME pseudo-function
   IF X is referencing
      IF argument is NAME data item
         Place (argument) in (referencing) path sets
      ENDDIF
   ELSEIF X is defining
      Place (argument) in (defining) path sets
   ENDDIF

User-defined function or procedure name:
   DO for each argument
      IF argument is an expression other than a single data item
         Place (argument) in (referencing) path sets
ELSE

* Argument is single data item -
  * subscripted or unsubscripted variable name
  *

IF argument is subscripted variable
Place (variables in subscript) in
(referencing) path sets
ENDIF

Get the formal parameter corresponding to argument
Pass path set membership of (parameter) to (argument)
Report non-usage of (argument) corresponding to (parameter)
Report side effects involving (argument)
ENDIF
ENDDO

Pass path set membership of global variables over block boundary
Output message (documentation information on global variable usage in
invoked block)

Unsubscripted or subscripted variable:

IF X is referencing or undefining
  Enter (variable) in \( (A_X) \) path set
  Enter (variable) in \( (D_X) \) path set
  IF X is referencing
    Enter (variable) in \( (TEMPREF) \) path set
ENDIF
ELSE
  Enter (variable) in \( D_d \) path set
  IF variable is already in \( A_r \) or \( D_r \)
    Remove variable from \( D_r \)
  ELSE
    Enter (variable) in \( A_d \) path set
  ENDIF
ENDIF

IF subscripted variable
Place (variables in subscript) in (referencing) path sets
ENDIF

Other:
Skip token
ENDDO
Check for side effects. A side effect occurs if evaluation of a function alters the value of any other element within the expression, right-hand-side of assignment statement, or CALL statement in which the function invocation appears. Sets TEMPREF and TEMPDEF are used to contain variables which were classified referenced or defined while processing this token (variable or procedure or function reference). Sets REF and DEF contain variables which were referenced or defined in that part of the statement analyzed up to this token.

\[
\text{SIDEFCT} = (\text{TEMPREF} \cap \text{DEF}) \cup \text{SIDEFCT} \\
\text{SIDEFCT} = (\text{TEMPDEF} \cap \text{REF}) \cup \text{SIDEFCT} \\
\text{REF} = \text{REF} \cup \text{TEMPREF} \\
\text{DEF} = \text{DEF} \cup \text{TEMPDEF}
\]

ENDDO
END
SEGMENT Pass path set membership of (parameter) to (argument)

*****************************************************************************
**
** This segment passes the path set membership of a formal parameter in an invoked procedure to the corresponding actual argument in order to reflect data flow across procedure boundaries.
**
*****************************************************************************

DO for PATHSET = A_x, B_x, C_x, D_x, E_x, F_x, and I for x = r, d, u
    IF parameter ∈ PATHSET (graph_{called})
        Enter (argument) in (PATHSET(node_{caller})) path set
    ENDIF
ENDDO

IF parameter ∈ G(graph_{called})
    Enter (argument) in (G(graph_{caller})) path set
    Enter (argument) in (TEMPDEF) path set
ENDIF

IF parameter ∈ C_r (graph_{called}) or F_r (graph_{called})
    Enter (argument) in (TEMPREF) path set
ENDIF
END

SEGMENT  Report non-usage of (argument) corresponding to (parameter)

IF argument is input argument

  IF parameter ∈ I(graph_{called})
    Output message "(Argument) specified as input argument is not referenced in (called)"
  ENDIF
ELSEIF parameter ∉ G (graph_{called})

  Output message "(Argument) specified as assign argument is not assigned a value in (called)"
ENDIF
END
SEGMENT Report side effects involving (argument)

(1) Detect side effect in which input argument is used by its global name and assigned a value in called block, as well as being associated with a formal input parameter.

IF argument is input argument
    IF argument ∈ G (graph called)
        Output message ("Side effect condition - actual input argument is used by its global name in (called block) and is defined there.")
    ENDIF
ENDIF

(2) Detect side effect in which assign argument is used by its global name in called block.

IF argument is assign argument
    IF argument ∈ A_X, B_X, C_X, D_X, E_X, F_X or G for X = r, d, u
        Output message ("Side effect condition - assign argument is used by its global name in (called block).")
    ENDIF
ENDIF

(3) Detect side effect in which an argument appears both as an input and an assign argument in the same call.

IF argument is assign argument and also appeared as an input argument
    Output message ("Side effect condition - same data item appears both as an input argument and an assign argument.")
ENDIF

(4) Detect side effect in which an assign argument appears more than once in the list of assign arguments.
IF argument is assign argument and appears elsewhere in assign argument list
    Output message ("Side effect condition - argument appears more than once in assign list.")
ENDIF
END
SEGMENT Pass path set membership of global variables over block boundary

DO for each variable var in list of global variables for called block
  DO for PATHSET = A_x, B_x, C_x, D_x, E_x, F_x, and I for x = r, d, u
    IF var ∈ PATHSET (graph called)
      Enter (var) in (PATHSET (nodeCaller))
    ENDF
  ENDDO

IF var ∈ G (graph called)
  Enter (var) in (G(graph caller)) path set
  Enter (var) in (TEMPDEF) path set
ENDIF

IF var ∈ C_r (graph called) or F_r (graph called)
  Enter (var) in (TEMPREF) path set
ENDIF

ENDDO
END
SEGMENT Determine path sets for entire block

Number flow graph in postorder
DO for each node in the flow graph
    Set up and initialize sets:  NULL, KILL, GEN, LIVE, AVAIL, A_\(x(n->)\),
    C_\(x(n->)\), D_\(x(--n)\), F_\(x(--n)\), x = r, d, u
ENDDO

Set up and initialize path sets:  A_\(x(graph)\), B_\(x(graph)\), C_\(x(graph)\),
    D_\(x(graph)\), E_\(x(graph)\), F_\(x(graph)\), I(graph), x = r, d, u
DO for x = r, d and u
    DO Case of x
    x=r:
        y = d
        z = u
    x=d:
        y = r
        z = u
    x=u:
        y = r
        z = d
ENDDO

DO for each node in the flow graph
    IF Node type is not exit
    NULL(node) = I(node) u B_\(x(node)\)
    KILL(node) = A_\(x(node)\)
    GEN(node) = \{all variables\} - (KILL(node) u NULL(node))
    ELSE
    NULL(node) = \ø
    KILL(node) = \ø
    GEN(node) = \{all variables\}
    ENDIF
ENDDO

Execute LIVE on graph
A_\(x(graph)\) = \{all variables\} - LIVE(entry node)
DO for each node in the flow graph
    A_\(x(n->)\) = \{all variables\} - LIVE(node)
ENDDO
DO for each node in the flow graph
   GEN(node) = C_x(node)
   KILL(node) = (A_y(node) u A_z(node))
   NULL(node) = {all variables} - (GEN(node) u KILL(node))
ENDDO

***** Determine C_x(graph) & C_x(n->) *****
***** Determine C_x(graph) & C_x(n->) *****

Execute LIVE on graph

C_x(graph) = LIVE(entry node)
DO for each node in the flow graph
   C_x(n->) = LIVE(node)
ENDDO

***************
* Determine B_x(graph) *
***************
***************
* Determine B_x(graph) *
***************
***************
* Determine B_x(graph) *
***************

DO for each node in the flow graph
   NULL(node) = I(node) u B_x(node)
   KILL(node) = A_x(node)
   GEN(node) = {all variables} - (KILL(node) u NULL(node))
ENDDO

Execute LIVE on graph

B_x(graph) = ({all variables} - LIVE(entry node)) n ({all variables} -
              A_x(graph)) n C_x(graph)

***************
* Determine D_x(graph) and *
* D_x(--->n) *
***************
***************
* Determine D_x(graph) and *
* D_x(--->n) *
***************

DO for each node in the flow graph
   GEN(node) = D_x(node)
   KILL(node) = (F_y(node) u F_z(node))
   NULL(node) = {all variables} - (GEN(node) u KILL(node))
ENDDO

Execute AVAIL on graph

D_x(graph) = AVAIL(exit node)
DO for each node in the flow graph
   D_x(--->n) = AVAIL(node)
ENDDO
DO for each node in the flow graph
  IF node type is → entry
    GEN(node) = D_y(node) u D_z(node)
    KILL(node) = F_x(node)
    NULL(node) ={all variables} - (KILL(node) u GEN(node))
  ELSE
    GEN(node) ={all variables}
    KILL(node) = Ø
    NULL(node) = Ø
  ENDP
ENDDO

Execute AVAIL on graph

F_x(graph) ={all variables} - AVAIL(exit node)
DO for each node in the flow graph
  F_x(--n) ={all variables} - AVAIL(node)
ENDDO

DO for each node in the flow graph
  IF node type is → entry
    GEN(node) = D_x(node)
    KILL(node) = F_y(node) u F_z(node)
    NULL(node) ={all variables} -(GEN(node) u KILL(node))
  ELSE
    GEN(node) ={all variables}
    KILL(node) = Ø
    NULL(node) = Ø
  ENDP
ENDDO

Execute AVAIL on graph

E_x(graph) = AVAIL(exit node) n (all variables) -(D_x(graph)) n F_x(graph)
ENDDO
Determine $I(\text{graph})$

$I(\text{graph}) \leftarrow \{\text{all variables}\}$

DO For each node in the flow graph

IF node type = (entry or exit)

$I(\text{graph}) = I(\text{graph}) \cap I(\text{node})$

ENDIF

ENDDO

END
SEGMENT Number flow graph in postorder

*****************************************************************************
* This segment performs a depth first search on a flow graph and numbers the nodes in postorder by invoking recursive segment "Search and number".
*****************************************************************************

DO for all nodes n in flow graph
     Indicate n "unmarked"
ENDDO
     i = 0
     Search and number (entry node)
END
SEGMENT Search and number (v)

******************************************************************************
* This recursive segment numbers nodes in a directed graph in postorder. *
******************************************************************************

DO for each node w on v's list of exit nodes
  IF w is unmarked
    Mark w
    Search and number(w)
  ENDDIF
ENDDO

i = i + 1
POSTORDER(v) = i

END
SEGMENT Execute LIVE on graph

n = number of nodes in flow graph

****** Graph is numbered in postorder ****************************

DO for j = 1 to n
    LIVE(j) = Ø
ENDDO

change = true

DO while change is true
    change = false
    DO for j = 1 to n
        PREVIOUS = LIVE(j)
        LIVE(j) = Ø
        DO for k = each of j's successors
            LIVE(j) = (LIVE(k) ∩ ([all variables] - KILL(k)))
                          ∪ GEN(k)) ∪ LIVE(j)
        ENDDO
        IF PREVIOUS ⊢= LIVE(j)
            change = true
        ENDF
    ENDDO
ENDDO
END
SEGMENT Execute AVAIL on graph

\[ n = \text{number of nodes in flow graph} \]

\[
\begin{align*}
\text{AVAIL}(n) &= \emptyset \\
\text{DO} & \text{ for } j = n-1 \text{ to } 1 \\
& \quad \text{AVAIL}(j) = \text{all variables} \\
\text{ENDDO} \\
\text{change} &= \text{true} \\
\text{DO} & \text{ while change is true} \\
& \quad \text{change} = \text{false} \\
& \quad \text{DO} \text{ for } j = n-1 \text{ to } 1 \\
& \quad \quad \text{PREVIOUS} = \text{AVAIL}(j) \\
& \quad \quad \text{AVAIL}(j) = \text{all variables} \\
& \quad \quad \text{DO} \text{ for } k \text{ each of node } j's \text{ predecessors} \\
& \quad \quad \quad \text{AVAIL}(j) = \text{AVAIL}(j) \cap ((\text{AVAIL}(k) \cap (\text{all variables}) \\
& \quad \quad \quad \quad \quad \quad \quad \text{ - KILL(k)}) \cup \text{GEN}(k)) \\
& \quad \text{ENDDO} \\
& \quad \text{IF PREVIOUS } \rightarrow \text{ AVAIL}(j) \\
& \quad \quad \text{change} = \text{true} \\
& \text{ENDIF} \\
& \text{ENDDO} \\
& \text{ENDDO} \\
& \text{END}
SEGMENT Report local anomalies

IF block is main program
   DO for each COMPOOL variable in I(graph\textsubscript{main})
      Output message ("COMPOOL variable (variable name) unused in entire program.")
   ENDDO
ENDIF

DO for each local variable in I (graph\textsubscript{block})
   Output message ("Variable (variable name) declared in block (block name) is never used.")
ENDDO

IF block is function block
   Get entry nodes for block's exit node
   DO for each entry node n
      IF n is not a RETURN node and ∃ a path from start node of function to n
         Output message ("Execution of function block possibly ends on statement (number corresponding to node) which is not a RETURN statement.")
      ENDF
   ENDDO
ENDIF

Determine anomalous paths of type ur, dd, du

END
SEGMENT Determine anomalous paths of type ur, dd, du

DO for FORM = 1, 2, and 3
  DO case of FORM
    FORM = 1:
      x = u
      y = r
    FORM = 2:
      x = d
      y = d
    FORM = 3:
      x = d
      y = u
  ENDDO

DO for each node n in flow graph
  Get path sets for n
  ANOM = F_x(n) \cap C_y(n+)
  IF ANOM \not\rightarrow empty
    DO for each variable in ANOM
      IF variable is simple variable or FORM is ur
        Find a path that contains anomaly (leaving, xy, some, node) for (variable)
        Output message ("On one or more paths leaving node anomaly of type (FORM) occurs for (variable). One such path is ...")
    ENDF
  ENDDO
ENDIF

ANOM = D_x(n) \cap A_y(n+)
IF ANOM \not\rightarrow empty
  DO for each variable in ANOM
    IF variable is simple variable or FORM is \not\rightarrow dd
      Find a path that contains anomaly (leaving xy, all, node) for (variable)
      Output message ("On all paths leaving node, anomaly of type (FORM) occurs for (variable). One such path is ...")
  ENDF
ENDDO
ENDIF
ANOM = F_x(\rightarrow n) \cap C_y(n)

IF ANOM \rightarrow empty

DO for each variable in ANOM

IF variable is simple variable or FORM is ur

Find a path that contains anomaly (entering, xy, some, node) for (variable)

Output message ("On one or more paths entering node, anomaly of type (FORM) occurs for (variable). One such path is ...")

ENDIF

ENDDO

ENDIF

ANOM = D_x(\rightarrow n) \cap A_y(n)

IF ANOM \rightarrow empty

DO for each variable in ANOM

IF variable is simple variable or FORM is \rightarrow dd

Find a path that contains anomaly (entering xy, all, node) for (variable)

Output message ("On all paths entering node, node, anomaly of type (FORM) occurs for (variable). One such path is ...")

ENDIF

ENDDO

ENDIF

ENDDO

ENDDO

END
SEGMENT  Find a path that contains anomaly (direction, xy, frequency, node) for (variable)

direction = "entering" or "leaving"
xy = "ur", "dd", "du", depending upon the type of the anomaly
frequency = "some paths" or "all paths"
node = node where anomaly was detected
variable = variable for which anomaly was detected

Although one solution for finding a path containing an anomaly is to perform a restricted depth first search for one variable at a time, this segment will not be specified here as work is in progress to find more efficient algorithms for localizing anomalous path expressions [7].
SEGMENT Report global anomalies

Determine anomalous paths of type ur, dd, du

END
V. Extension of Analytic Techniques to Concurrency

The data flow analysis system described in this report assumes that the HAL/S code to be analyzed consists of a single process. However, the facility for creating a multi-process program structure is a major feature of the HAL/S language. The algorithms described here cannot be applied directly to analyze such code, but it appears that they can be adapted to concurrent-process programs. The concurrent analysis employs precedence graphs in which the effect of real time statements is modelled by special edges joining the flow graphs of different processes [8]. Central to the analysis is the static determination for each node in the program flow graph of which other nodes must precede its execution, follow its execution, or possibly execute concurrently with it. A complete presentation of these analytic techniques will be contained in a forthcoming report [9].

Acknowledgments

I would like to acknowledge Rebecca Jones for significant help in creating the PDL specifications of the analysis algorithms.
References


4. B. Edwards, "Graph Representations for HAL/S Programs" (to appear).


