DESIGN OF A SYSTEM FOR ANOMALY DETECTION IN HAL/S PROGRAMS

G. Bristow, C. Drey
B. Edwards, W. Riddle

CU-CS-151-79 March 1979

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

DESIGN OF A SYSTEM FOR ANOMALY DETECTION IN HAL/S PROGRAMS†

by

G. Bristow, C. Drey,
B. Edwards, W. Riddle

Computer Science Department
University of Colorado at Boulder

CU-CS-151-79 March, 1979

†This work was supported by grant NSG 1476 from NASA Langley Research Center.

Abstract

An approach to the analysis of HAL/S software is discussed. The approach, called anomaly detection, involves the algorithmic derivation of information concerning potential errors and the subsequent, possibly non-algorithmic determination of whether or not the reported anomalies are actual errors. We give detailed designs for algorithms for detecting data-usage and synchronization anomalies and discuss how this technique may be integrated within a general software development support system.

Introduction

In developing software systems, especially large, complex ones, practitioners require analytic techniques to help them assess the validity of the system. In this paper, we explore an approach to providing these analytic techniques which we call anomaly detection and present anomaly detection algorithms useful for the assessment of HAL/S [Inte 76] programs.

In the anomaly detection approach, assessment is a two-step procedure. First, algorithms are employed to discover potential errors (anomalies) as evidenced by deviations from the developers' expectations. Second, non-algorithmic analysis, relying upon the experience, knowledge, and expertise of the developers themselves, is employed to determine whether or not a reported anomaly represents an actual error.

To focus our work, we have established the following criteria. First, our techniques must be applicable to programming language representations of the software system. Thus, they will not have to await the acceptance of some modeling representation by the system developers. Second, our techniques should be oriented toward expectations that arise from general concerns which pertain to a wide spectrum of programs. These concerns may reflect problem-domain considerations, the semantics of programming languages, or general rules of good practice. Thus, we do not have to develop techniques for specifying problem-specific expectations in order to have our techniques be applicable to a wide range of systems. Third, our techniques should not be restricted to sequential systems, but should apply also to systems with concurrency. This makes them applicable to those complex systems which involve either actual or apparent parallelism. Finally, our techniques should be of "reasonable" quality. We desire techniques that are considerably more effective than the trivial one which always, for all programs, announces "There's possibly an error somewhere in the program"; but we want techniques in which the algorithms have pleasing computational properties.

It should be stressed that we view anomaly detection as only one of the types of analytic techniques which should be made available to development practitioners. We feel that by not attempting to do complete analysis, we can find useful techniques which have reasonable computational requirements and are generally applicable over a broad range of software systems. We also feel that our current work gives rise to immediately usable techniques, but that it is preliminary in nature and many questions remain concerning its effectiveness and the degree to which anomaly detection techniques may be integrated into a full set of analytic techniques.

In the next section we give a brief overview of the anomaly detection system we envision, indicate how it may be incorporated as part of a more extensive development support system, and present a small example to convey an intuitive understanding of the purpose and functioning of the various parts of the anomaly detection system. The following sections address the various phases of our system in turn, covering the capabilities of the anomaly detection algorithms we have developed. In the concluding section, we discuss the implications of some of the constraints we have imposed in order to focus our work and indicate future directions we plan to pursue.

In the Appendices to this report, we give detailed pseudo-code descriptions for various segments of the anomaly detection system which we have developed. These algorithm specifications give the logic of the processing, but would have to be further refined during implementation to introduce data storage and processing efficiencies.

II. Anomaly Detection System Overview

We envision that the anomaly detection algorithms will be provided as tools within a software development support system. This support system would provide a variety of tools to development practitioners, supporting both management and bookkeeping activities as well as assessment activities. The support system would be organized as a set of modules, each of which augments and/or displays the information concerning the system under development which is stored in some central information repository. This organization is depicted in Figure 1.

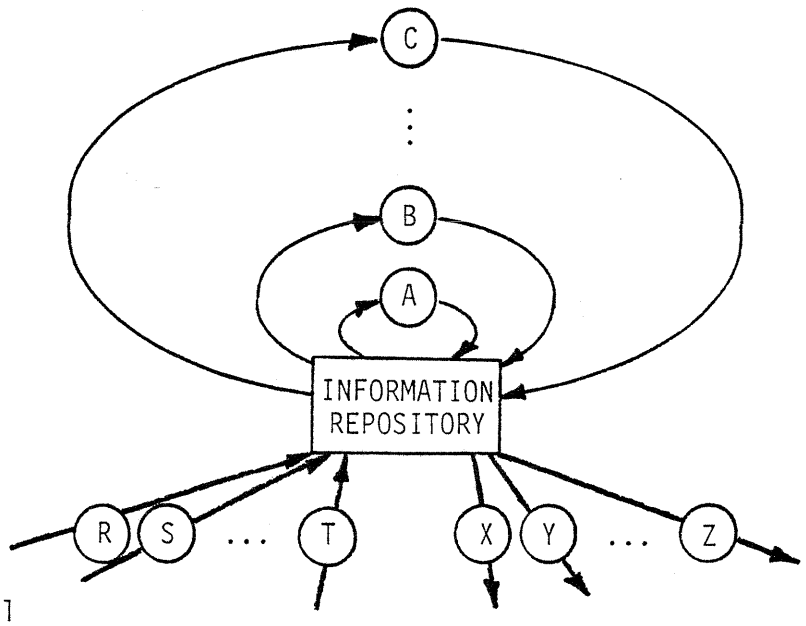


Figure 1
Organization of Development Support System

Guided by an overall methodology, practitioners would use the various modules, in sequence and in parallel, to gradually evolve a detailed description of the system under development. During this evolution process, progress and validity could be periodically and continuously assessed by employing those modules provided for this purpose. The anomaly detection modules would be among this set of assessment modules.

To represent specific ways in which use of the modules may be coordinated to achieve some overall information transformation, we use the graphical notations presented in Figures 2 and 3.

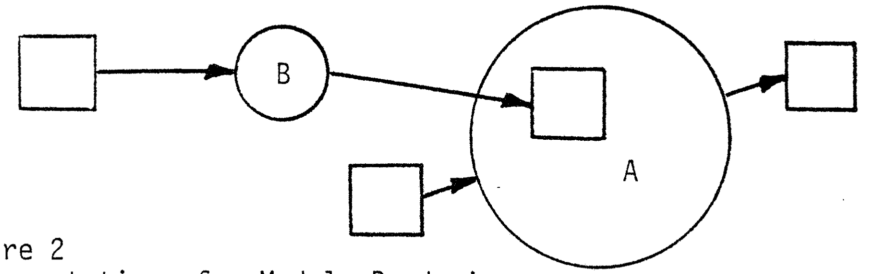


Figure 2
Representation of a Module Producing Information Used in Another Module's Processing

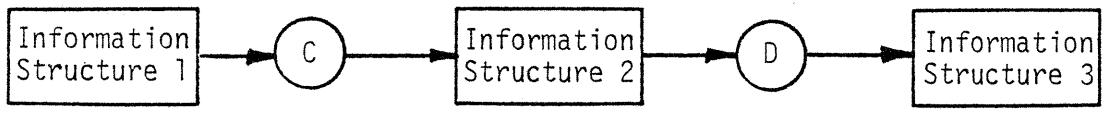


Figure 3
Representation of a Module Producing Information Used as Input by Another Module

The notation of Figure 2 is used to indicate that information in the central repository has been deposited by one module (B) specifically so that some other module (A) may perform its function. (The usual implication is that B's processing is done much less frequently than A's.) The notation of Figure 3 denotes that the information produced by one module (C) is subsequently transformed by another module (D).

Using these notations, the ideal anomaly detection subsystem may be depicted as in Figure 4. This system is language independent but can be particularized by information prepared by a language

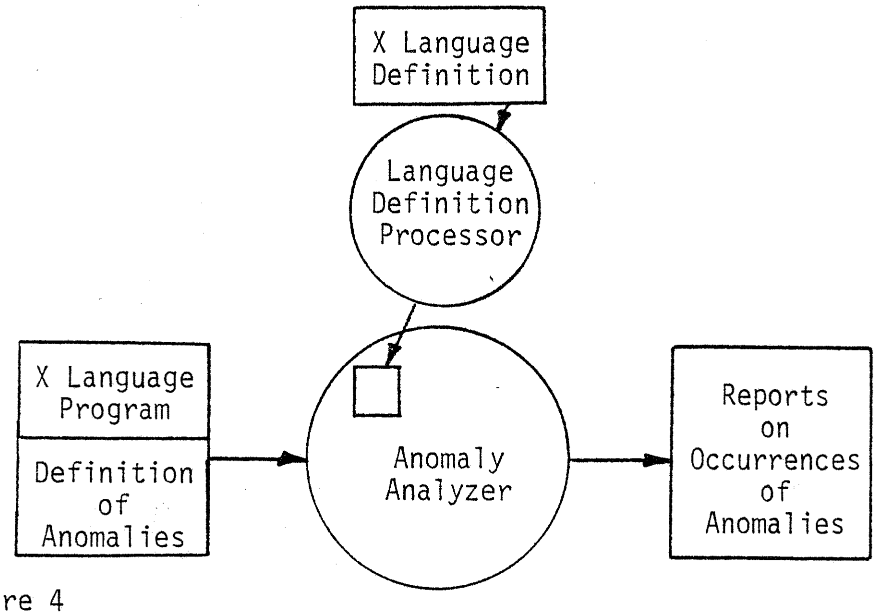


Figure 4
Ideal Anomaly Detection System

definition processor. This system would also be able to accept definitions of the anomalies to be detected.

We have reduced the scope of the problem by assuming that we are working with a particular language and by focusing specifically upon data-usage and synchronization anomalies. Therefore, the system for which we strive is that depicted in Figure 5. (We have been working with a particular language, HAL/S, but our techniques can be employed with other concurrent programming languages as they do not depend on the exact form of the language's constructs.)

The anomaly detection task may be decomposed into two major subtasks. The first is to derive a representation of the program under analysis which retains the information pertinent to the anomalies under detection and presents this information in a form which may be conveniently used by the anomaly detection algorithms. The second task is the anomaly detection itself. In Figures 6-9, we indicate the major components which perform these tasks.

So that the anomaly detection subsystem may easily be generalized to other languages, the initial processing module performs a program-to-parse-tree transformation (Figure 6). Identifying this as a separate module leads to two subsidiary benefits. First, it allows the use of existing scanner and parser generation systems in the preparation of the HAL/S Language Processor module. Second, the overall system may be easily modified to use representations of HAL/S programs other than the program text. In particular, the overall

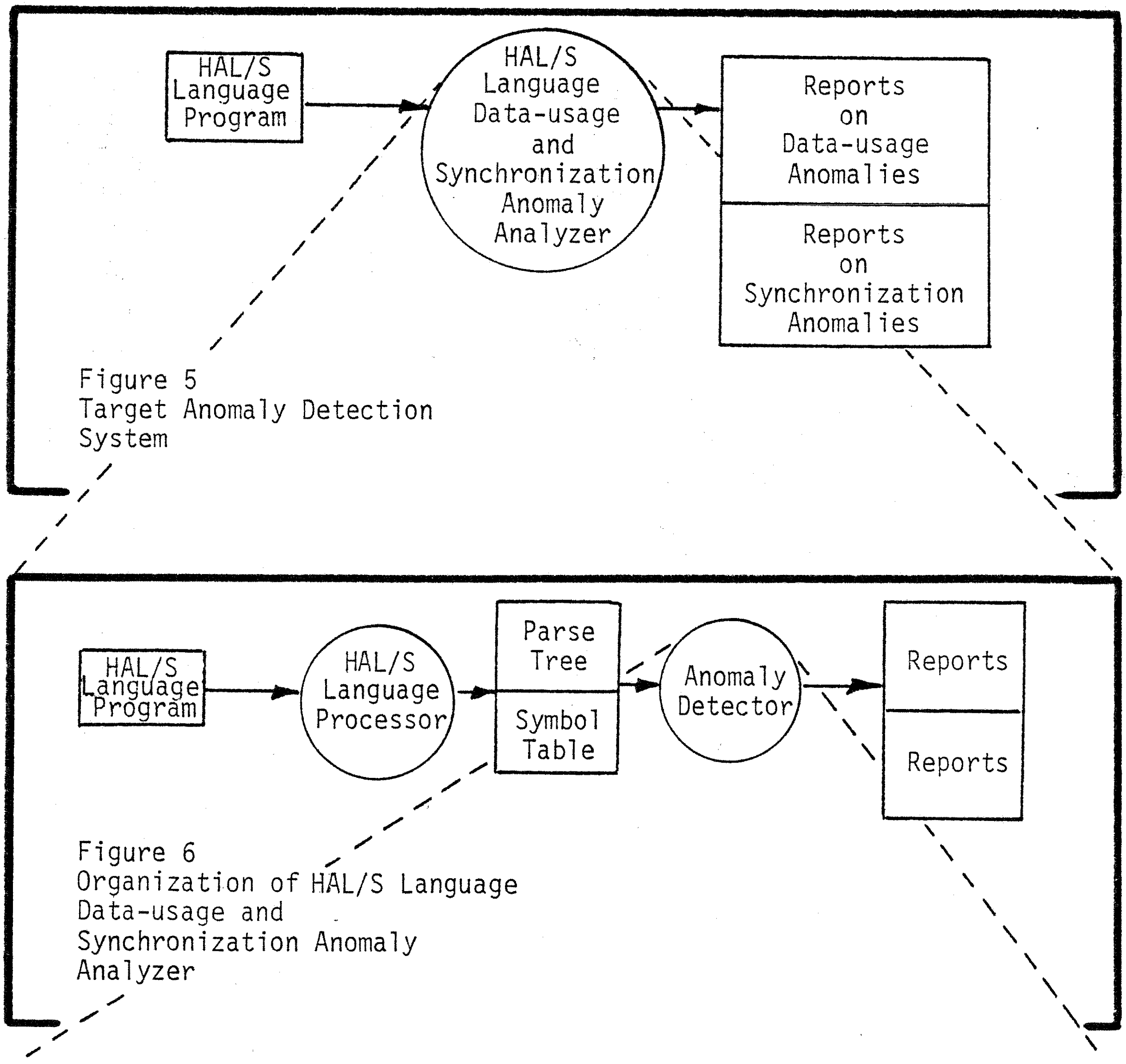

HAL/S Language Program
HAL/S Language Data-usage and Synchronization Anomaly Analyzer
Reports on Data-usage Anomalies
Reports on Synchronization Anomalies
HAL/S Language Program
HAL/S Language Processor
Parse Tree
Symbol Table
Anomaly Detector
Reports
Reports

Figure 5
Target Anomaly Detection System

Figure 6
Organization of HAL/S Language Data-usage and Synchronization Anomaly Analyzer

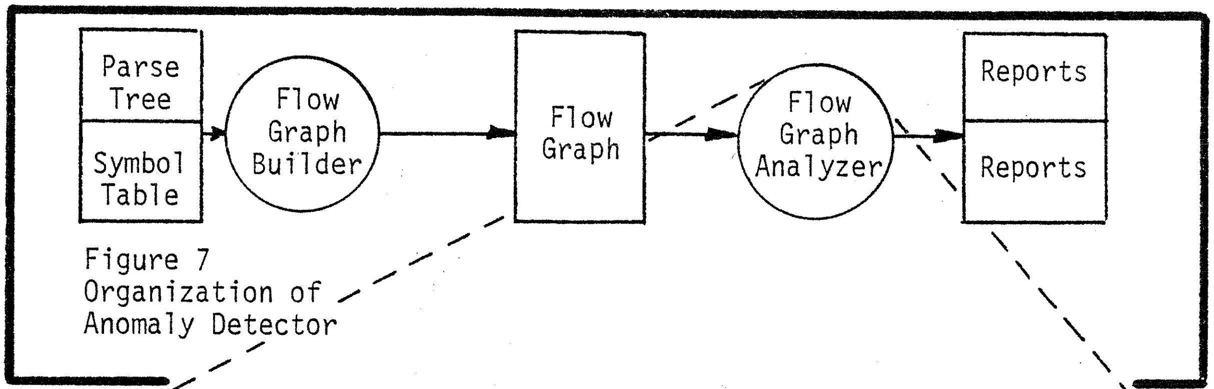


Figure 7
Organization of
Anomaly Detector

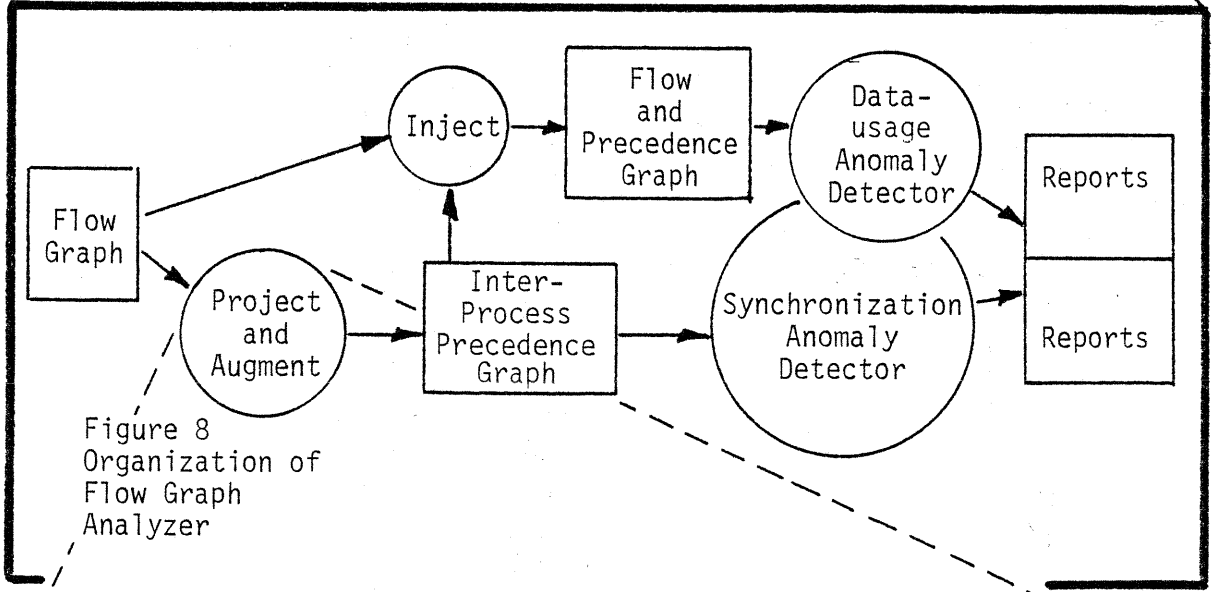

Flow Graph
Inject
Flow and Precedence Graph
Data-usage Anomaly Detector
Reports
Reports
Project and Augment
Inter-Process Precedence Graph
Synchronization Anomaly Detector

Figure 8
Organization of
Flow Graph
Analyzer

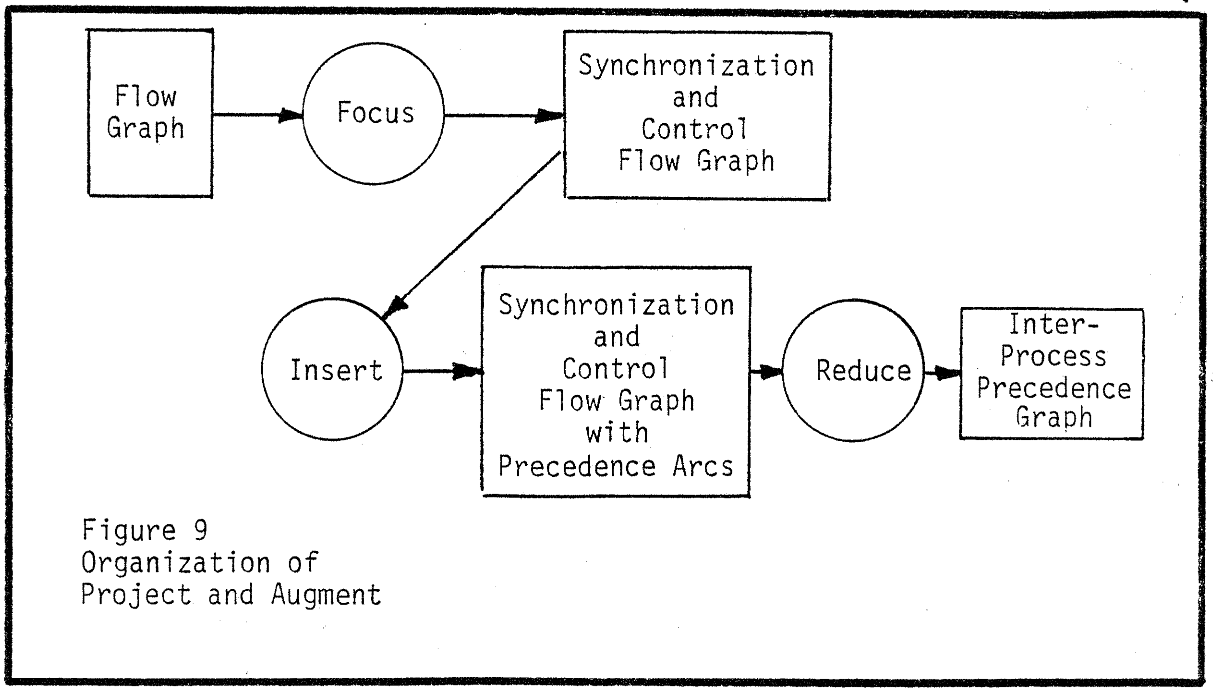

Flow Graph
Focus
Synchronization and Control Flow Graph
Insert
Synchronization and Control Flow Graph with Precedence Arcs
Reduce
Inter-Process Precedence Graph

Figure 9
Organization of
Project and Augment

system may be changed to use the internal, intermediate representation produced by the HAL/S language's compiler.

The next module which we identify has the task of building a flow graph representation of a program (see Figure 7). In a flow graph, nodes represent program statements (or perhaps fragments of statements) and arcs represent the flow of control within sequentially executed segments of the program (i.e., within program tasks). Further, for the purposes of our anomaly detection algorithms, cycles are removed by "unfolding" iteration loops. Identifying this as a separate module again eases our task since it is possible for us to consider using existing techniques in the design of the Flow Graph Builder module.

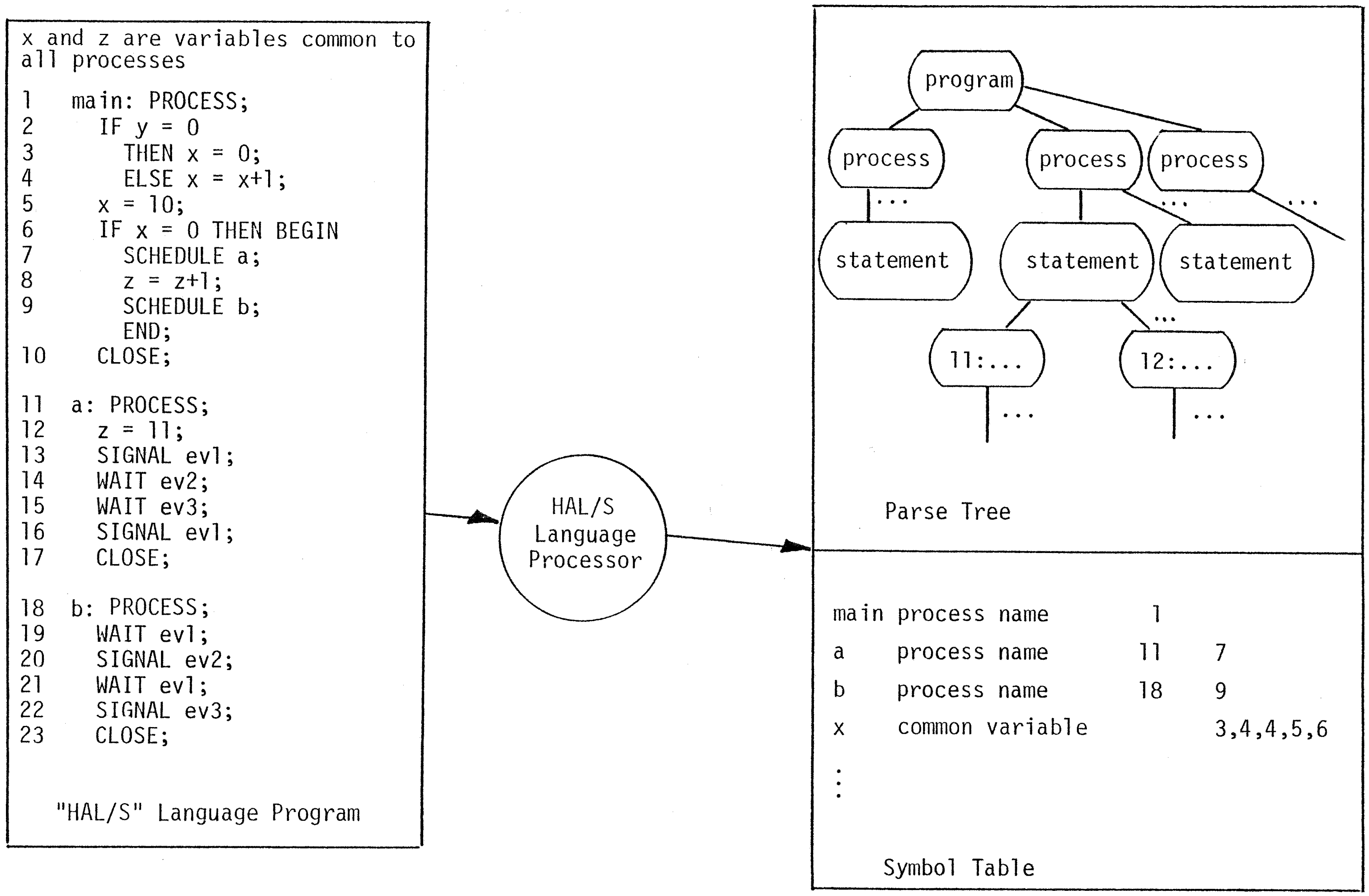
We decompose the Flow Graph Analyzer as indicated in Figure 8. In approaching the processing in this way, we separate out the task of constructing the Inter-Process Precedence Graph in which attention is focused upon process synchronization interactions and arcs are introduced to indicate the precedence of operations enforced by these interactions. This Inter-Process Precedence Graph may be used directly for the detection of synchronization anomalies, or the information contained in this graph may be injected into the flow graph to produce a combined Flow And Precedence Graph which may be used in the detection of data-usage anomalies.

Finally, we identify the modules depicted in Figure 9. These divide the task of constructing an Inter-Process Precedence Graph into three steps. First, the Flow Graph is processed to eliminate those nodes and arcs which do not pertain to the use of synchronization constructs or directly affect the flow of execution of synchronization operations. Then arcs reflecting the order of execution imposed by the synchronization operations are inserted into the graph. Finally, most (or ideally all) of the arcs which reflect impossible execution sequences are removed from the graph. The identification of these last two modules allows separate focus upon the simple task of obtaining _a_ representation of the effect of the synchronization operations and the much more difficult task of obtaining a representation which reflects the actual run-time behavior of the program under analysis.

Before giving the details of the individual components in this decomposition of the system, we first give a small example and discuss the relationship of our work to the work of others and to our previous work.

III. An Example

In Figure 10 we present an example program in the HAL/S language and show the various information structures produced during processing. The program is a meaningless one except for the purpose of indicating the major types of anomalies which our algorithms will detect.



x and z are variables common to all processes

```
1    main: PROCESS;
2      IF y = 0
3        THEN x = 0;
4        ELSE x = x+1;
5      x = 10;
6      IF x = 0 THEN BEGIN
7        SCHEDULE a;
8        z = z+1;
9        SCHEDULE b;
         END;
10     CLOSE;

11   a: PROCESS;
12     z = 11;
13     SIGNAL ev1;
14     WAIT ev2;
15     WAIT ev3;
16     SIGNAL ev1;
17     CLOSE;

18   b: PROCESS;
19     WAIT ev1;
20     SIGNAL ev2;
21     WAIT ev1;
22     SIGNAL ev3;
23     CLOSE;
```

"HAL/S" Language Program

main	process name	1	
a	process name	11	7
b	process name	18	9
x	common variable		3,4,4,5,6
⋮			

Symbol Table

Figure 10
Example Showing Program Representations

continued on next page

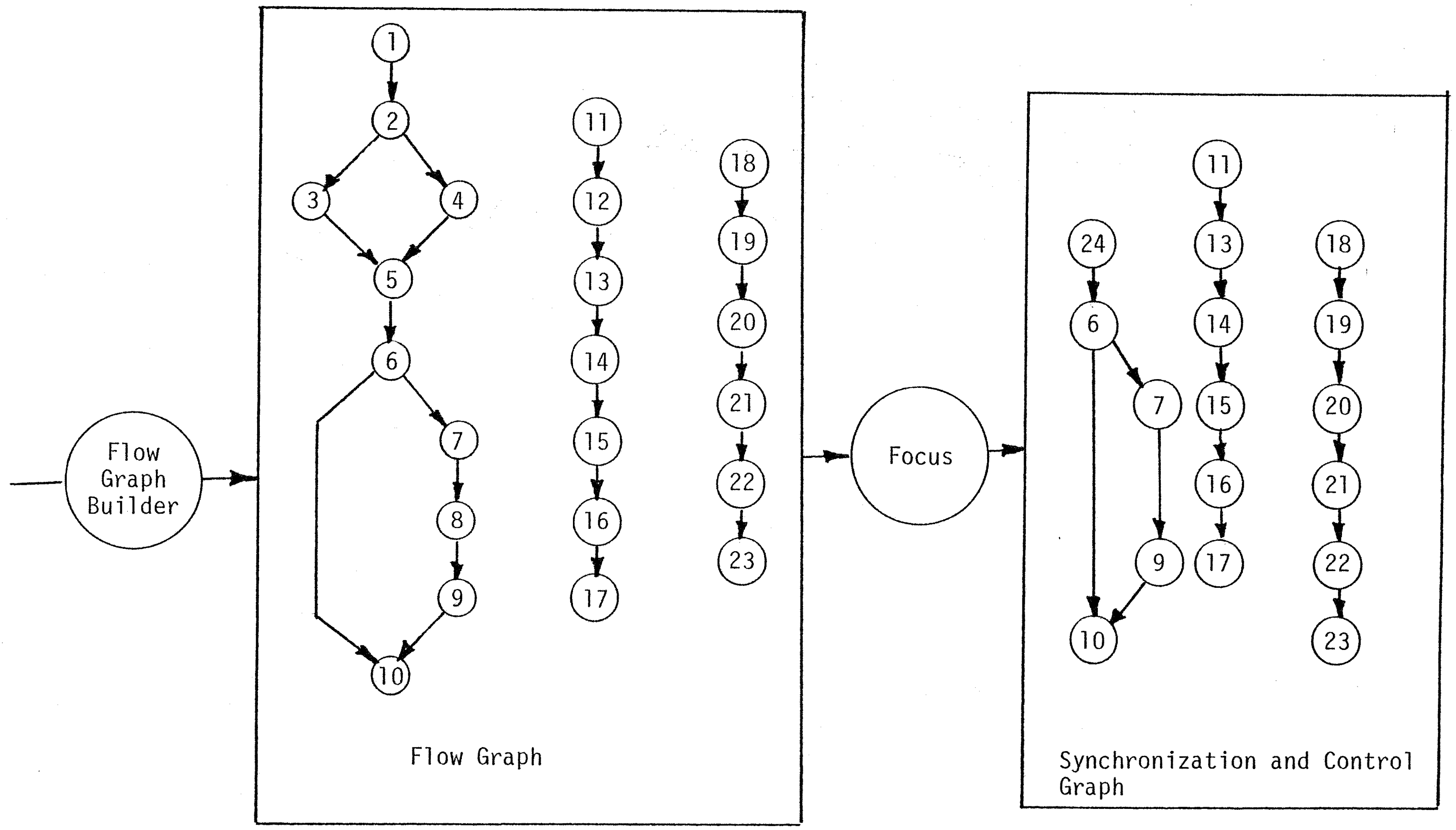


Figure 10
... continued

continued on next page

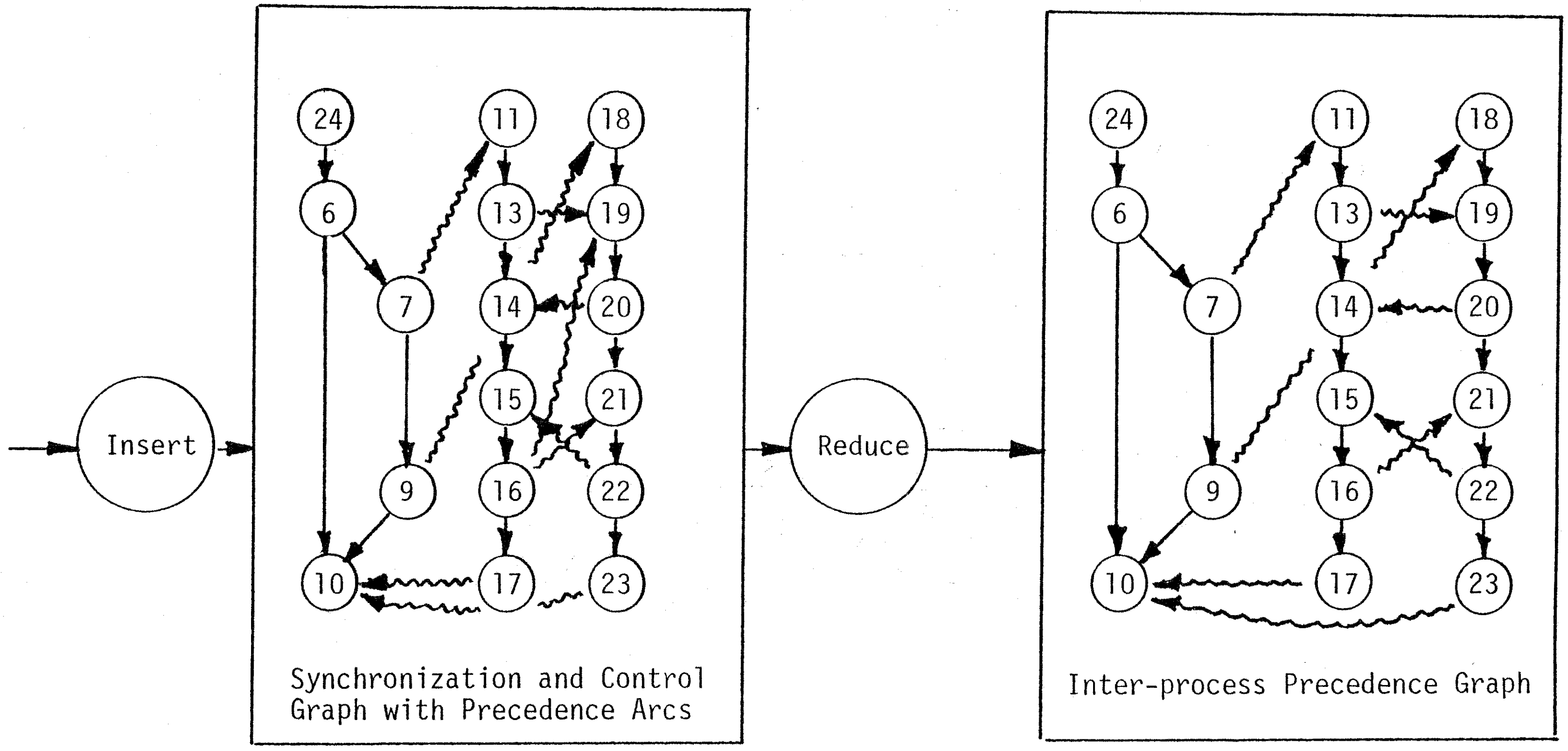
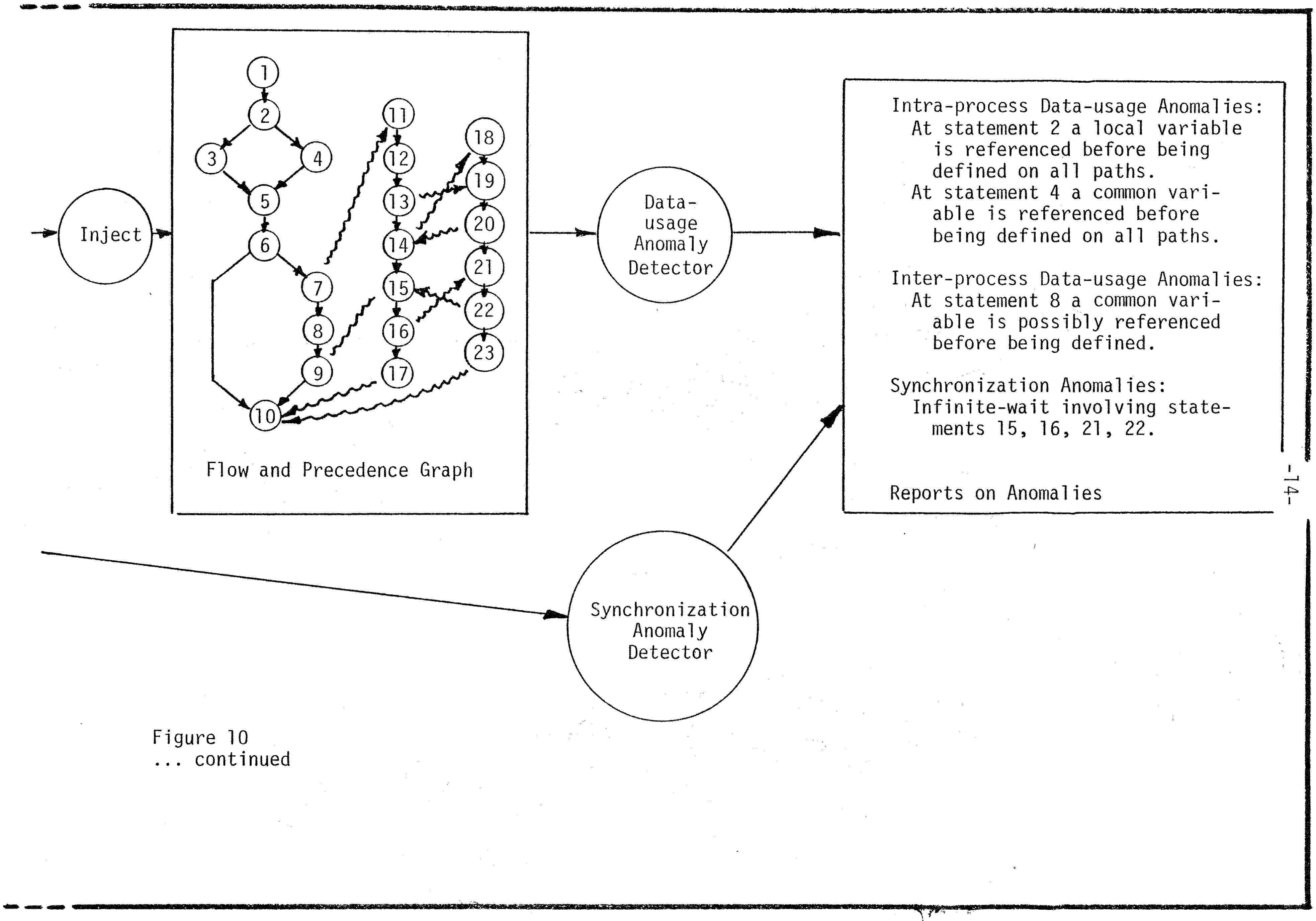

Insert
24
6
7
9
10
11
13
14
15
16
17
18
19
20
21
22
23
Synchronization and Control Graph with Precedence Arcs
Reduce
Inter-process Precedence Graph

Figure 10
... continued

continued
on next
page


Inject
Flow and Precedence Graph
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
Data-usage Anomaly Detector
Synchronization Anomaly Detector
Reports on Anomalies

Intra-process Data-usage Anomalies:
At statement 2 a local variable is referenced before being defined on all paths.
At statement 4 a common variable is referenced before being defined on all paths.

Inter-process Data-usage Anomalies:
At statement 8 a common variable is possibly referenced before being defined.

Synchronization Anomalies:
Infinite-wait involving statements 15, 16, 21, 22.

Figure 10
... continued

IV. Related Work

Closely related to our own work is that of Taylor and Osterweil [TayR 78]. They share an interest in producing a general software development support system, and Osterweil has been actively involved in the DAVE data-usage anomaly detection system [OstL 76]. Although our paths of development differ, we have arrived at essentially the same point, except for relatively minor differences in capabilities and algorithms.

Reif's recent work [ReiJ 78] on the analysis of interacting processes deals with formal models of concurrent systems and decidability. It relates most directly to the formal foundational work which is a basis for the work reported here ([GreI 77], [OgdW 78], [PetJ 74], [PetJ 76], [PetJ 78], [RidW 72], [RidW 73], [RidW 74], [RidW 78a], [ShaA 78], [WilJ 78]).

The Inter-Process Precedence Graph is an intermediate representation for describing the partial ordering of synchronization events within concurrent systems. Thus, it is closely related to other techniques that have recently been developed for this purpose ([CamR 74], [GreI 77], [HabA 75], [RidW 78b], [ShaA 78]). Its representational power is equivalent to that of event expressions, defined in [RidW 78b].

Our synchronization anomaly detection algorithms were developed after initially attempting to employ the static deadlock detection algorithms developed by Saxena [SaxA 77]. However, we found the requirements for use of those algorithms to be too strict for our purposes.

The data-usage anomaly detection phase of our system is derived from the DAVE system for analyzing FORTRAN programs [OstL 76]. Faster, more efficient algorithms [FosL 76] evolved from the original system and the elements of the analysis performed by them are essentially language independent. These algorithms have been applied to the HAL/S language for single-process programs to design a DAVE-HAL/S system [DreC 78]. This work has been extended to include analysis of multi-process HAL/S programs as well and will be described here in relation to the HAL/S language.

V.1 Flow Graph

The flow graph is derived from the parse tree and symbol table for a program specified in the HAL/S language. The flow graph is an abstraction of the control structure of the program and is used to detect anomalous data flow patterns.

The flow graph is composed of a subgraph for each subprogram unit in the program under analysis. Each subgraph contains:

1. N, a set of nodes, $\{n_1, n_2, n_3, \ldots, n_k\}$
2. E, a set of ordered pairs of nodes (edges), $\{(n_{j_1}, n_{j_2}), (n_{j_3}, n_{j_4}), (n_{j_5}, n_{j_6}), \ldots, (n_{j_{m-1}}, n_{j_m})\}$, where the n_{j_i} s are not necessarily distinct.
3. n_e, the unique entry node, $n_e \varepsilon N$.
4. n_x, the unique exit node, $n_x \varepsilon N$.

The nodes in the graph roughly correspond to statements in the program. The edges in the graph indicate flow of control from one node to the next. The flow graph is acyclic. Each loop in the program is expanded to reflect zero-or-more or one-or-more repetitions of the loop. Each node, $n_j \varepsilon N$, has the following information associated with it:

1. P, the set of predecessor nodes. $n_i \varepsilon P$ if the edge $(n_i,n_j) \varepsilon E$.
2. S, the set of successor nodes. $n_i \varepsilon S$ if the edge $(n_j,n_i) \varepsilon E$.
3. t, the type of the node, indicating the type of statement in the language HAL/S which the node represents.
4. r, a representation of the actual statement or statement fragment which the node represents.
5. m, the sequential number of the statement which the node represents.

V.2 Flow Graph Builder

The flow graph is produced by the flow graph builder from the parse tree and symbol table for a program. The major tasks in the production of the flow graph are to expand loops and to model the synchronization constructs with SETs and WAITs. (See Appendix A.)

V.3 Inter-Process Precedence Graph

The Inter-Process Precedence Graph, derived from the Flow Graph, is an abstraction of the synchronization constructs and the control structures which directly affect the flow of execution of the synchronization operations. The Inter-Process Precedence Graph is used to detect anomalous patterns of synchronization operations.

The graph is composed of subgraphs for each process in the program. Each subgraph is a flow graph, representing the synchronization operations and the pertinent control structures in the program.

The synchronization constructs are modeled by combinations of SET, RESET, and WAIT operations applied to event variables. Event variables are binary valued variables. They may be set to true (SET), set to false (RESET), or a process may be suspended until a logical expression over event variables is true (WAIT). For the languages we have considered, SET, RESET, and WAIT appear to be sufficient to model all synchronization constructs.

The subgraphs are linked together by inter-process precedence edges (IPPEs) as shown in Figure 11. An IPPE is an edge

$$(n_{i_k}, n_j) \varepsilon \{(n_{i_1}, n_j), (n_{i_2}, n_j), \ldots, (n_{i_m}, n_j)\}$$

such that at least one of the n_{i_k}s must execute before n_j can execute. Thus the IPPEs indicate inter-process time orderings.

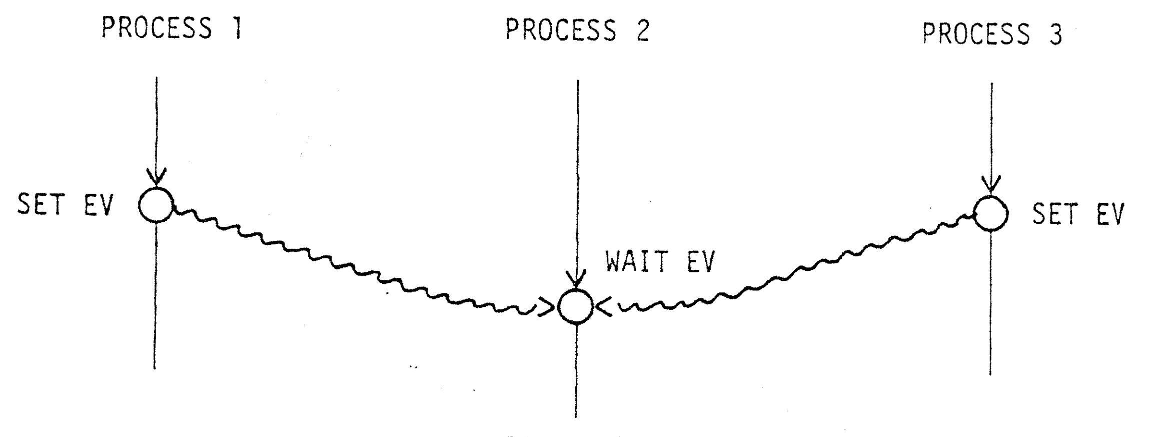


Figure 11

Since a WAIT on an event variable cannot be satisfied until a SET for that event variable has been executed, the IPPEs may be viewed as linking all SETs for a particular event variable to all of the WAITs for that same event variable.

The object of a WAIT can be a logical expression over event variables, and many SETs can occur for any particular event variable. This results in multiple IPPEs which lead to the same WAIT node. These IPPEs are grouped in conjunctive normal form. Thus, before the WAIT can be satisfied, at least one term in each conjunct must be true. Therefore, for each conjunct, at least one node having an IPPE in that conjunct must execute before the WAIT can be satisfied.

In the HAL/S language, the synchronization constructs can be modeled as follows:

1. SCHEDULE - a schedule is treated as a SET on an event variable representing permission for a process to execute.
2. PROCESS - a process is treated as having a WAIT on the event variable representing permission to execute.
3. CLOSE - a close is treated as a RESET on the event variable representing permission for the process to execute.
4. SIGNAL - a signal is treated as a SET, followed immediately by a RESET.

V.4 Precedence Graph Construction

The inter-process precedence graph is produced from the flow graph in three steps. First, Focus removes all nodes in the graph which do not represent synchronization constructs or control structures which directly affect the flow of execution of the synchronization constructs. Next, IPPEs are Inserted into the graph. (See Appendix B.) Finally, Reduce removes all spurious IPPEs from the graph. (See Section V.6.)

The flow and precedence graph is produced by Injecting the IPPEs, from the inter-process precedence graph, into the flow graph. (See Appendix B.)

V.5 Execution Sequence Sets

The inter-process precedence graph is used in the generation of a number of sets of nodes, collectively known as the execution sequence sets, at each node in the graph. These execution sequence sets give information about the possible and forced orders of execution of the nodes in the graph.

Before we discuss the contents of the individual sets, we shall first explain how we use the terms 'execution path' and 'execution sequence' when referring to concurrent programs.

An execution sequence through a program is the ordered set of statements that would be executed during a run of the program. It is assumed that where sections of two or more processes execute concurrently, there will still be an order in time of the execution of the individual statements; i.e. at some (possibly atomic) level, no two actions can occur simultaneously.

An execution path through a program can be regarded as the set of all possible execution sequences for the particular set of statements that constitute a run of the program. An execution path is a set of paths through each individual process in a run of the program, together with the partial orderings that are enforced by the synchronization statements. For a program with no potential concurrency, the set of execution sequences in an execution path contains only one element -- i.e. the two terms refer to the same thing. For a program with potential concurrency, there will be several execution sequences in an execution path, as the execution path contains no information concerning the actual order of execution of the individual statements in sections of two or more processes running concurrently.

At a node n, the following execution sequence sets will be generated:

ALWAYS(n)	contains all nodes which must execute if n is to execute; i.e. those nodes that are always present in every execution path containing n.
NEVER(n)	contains all nodes which cannot execute if n is to execute; i.e. those nodes that do not appear in any execution path containing n.

BEFORE(n) contains all nodes which, if they execute at all, will execute before n; i.e. those nodes that occur before n on at least one execution sequence containing n, but do not occur after n in any execution sequences containing n.

ALWAYS_BEFORE(n) is the subset of BEFORE(n) containing those nodes which always execute before n on all execution sequences containing n.

POSSIBLY_BEFORE(n) is the superset of BEFORE(n) containing those nodes which, for at least one execution path containing n, will execute before n on all execution sequences in that execution path.

CONCURRENT(n) contains every node e which satisfies the following conditions:

— e and n co-occur in at least one execution path

— in every execution path in which both nodes occur, there is no forced ordering of the two nodes, i.e. in every execution path in which both nodes occur, e will occur before n in at least one execution sequence in the execution path, and n will occur before e in at least one execution sequence in the execution path.

ALWAYS_CONCURRENT(n) is the subset of CONCURRENT(n) containing those nodes which satisfy the conditions for belonging to CONCURRENT(n) but also co-occur with n in every execution path containing n.

POSSIBLY_CONCURRENT(n) contains every node e which co-occurs with n on at least one execution path in which there is no forced orderings of the two nodes.

AFTER(n) contains those nodes which, if they execute at all, will execute after n, i.e. those nodes that occur after n in at least one execution sequence, but do not occur before n in any execution sequences.

ALWAYS_AFTER(n) is the subset of AFTER(n) containing those nodes which always execute after n in every execution sequence containing n.

POSSIBLY_AFTER(n) is the superset of AFTER(n) containing those nodes which, for at least one execution path containing n, will execute after n in all execution sequences in the execution path.

There is a certain amount of symmetry which is implied by the above definitions of the execution sequence sets. Firstly, if node a is in the NEVER set of node b, then node b will be in the NEVER set of node a. If node a is in the AFTER set of node b, then node b will be in the BEFORE set of node a, and vice versa. Similarly if node a is in the POSSIBLY_BEFORE set of node b, then node b will be in the POSSIBLY_AFTER set of node a, and vice versa.

To calculate the ALWAYS sets it is necessary to generate some intermediate sets. ALWAYS_BEFORE_WITHIN_PROCESS(n) contains those nodes, within the same process as n, that must always execute before n on all execution sequences containing n. Similarly for ALWAYS_AFTER_WITHIN_PROCESS(n). BEFORE_WITHIN_PROCESS(n) contains those nodes within the same process as n, that will execute either before n, or not at all, in all execution sequences containing n. Similarly for AFTER_WITHIN_PROCESS(n). The procedure to generate these sets takes advantage of the fact that loops within processes will have been expanded out before this stage is reached.

From these sets are generated the ALWAYS_SUBPROCESS_SETs and SUBPROCESS_SETs for each process. For a process p, SUBPROCESS_SET(p) contains all subprocesses of p; i.e. those processes which can be scheduled by p, or can be scheduled by a process scheduled by p, etc. ALWAYS_SUBPROCESS_SET(p) is the subset of SUBPROCESS_SET(p) containing those subprocesses of p which must always execute if p executes.

The first step in the generation of the final ALWAYS sets is to generate the ALWAYS set for the open node of the main program. It contains those nodes which must execute every time the program runs. It consists of all nodes that always execute in all subprocesses of MAIN that always execute, together with all nodes in MAIN that always execute. For each other process, the ALWAYS set of the open node is the same as the ALWAYS set of the schedule node.

For a node n, ALWAYS(n) contains the union of the following sets:

- ALWAYS of the open node of the process containing n
- ALWAYS_BEFORE_WITHIN_PROCESS(n)
- ALWAYS_AFTER_WITHIN_PROCESS(n)
- Those nodes that must execute in any subprocess p that must execute if n executes, but does not necessarily have to execute whenever the process containing n executes.

For the generation of the NEVER sets, it is necessary to generate NEVER_WITHIN_PROCESS sets. NEVER_WITHIN_PROCESS(n) contains those nodes within the same process as n that cannot execute if n executes. For the open node of a process other than the main program, the NEVER set is the same as the NEVER set of the schedule node. For a general node n, NEVER(n) is the union of the following sets:

- NEVER set of the open node of the process containing n
- NEVER_WITHIN_PROCESS(n)
- All nodes in all subprocesses that cannot execute if n executes.

CALCULATE_BEFORE(m,p) is a recursive function which returns as its value the BEFORE set of a node m in process p. Nodes in BEFORE_WITHIN_PROCESS(m) are members of BEFORE(m). Nodes in the intersection of the BEFORE and NEVER sets of nodes within p that have edges to m are placed in BEFORE(m) as well as nodes in the intersection of the BEFORE and NEVER sets of nodes within other processes which are the tails of IPPEs to m.

Because of the possibility of loops in the graph involving IPPEs, a stack, SAVE, is kept containing an entry (m,p) each time CALCULATE_BEFORE is entered. In the event that the recursive process leads to the calculation of the BEFORE set of a node m which occurs within the same process as a node n on SAVE and m is a descendant of n or an element of NEVER(n), then the identity is returned as the result of the call. Also, all entries on SAVE from the one after (n,p) to the top are placed in REDO, indicating that their BEFORE set calculations are valid only from the standpoint of node n and their true BEFORE sets need to be calculated separately.

CALCULATE_AFTER(m,p) employs the BEFORE sets, reasoning that if $n \varepsilon$ BEFORE(m), then $m \varepsilon$ AFTER(n).

The POSSIBLY_BEFORE set of each node m contains each entry node e to m plus BEFORE(e). The POSSIBLY_AFTER sets are formed from the POSSIBLY_BEFORE sets in the same manner as above.

The pseudo-code for the generation of the execution sequence sets appears in Appendix C.

V.6 Spurious IPPE Elimination

The last step in the construction of the Inter-Process Precedence Graph is to remove arcs which reflect impossible execution sequences.

The presence of each IPPE in this graph should indicate that three conditions have been satisfied.

1. The predecessor node causes a term in the wait expression of the successor node to become true.
2. The predecessor node will execute before the successor node in at least one legal execution sequence.
3. In at least one of the execution sequences in 2) the term will not become false again before the wait has completed.

During the building of the graph, however, all IPPEs are inserted which satisfy only condition 1) above, and it is possible for some of these to violate conditions 2) or 3) above. Those that do are spurious, and for more accurate results, should be removed prior to performing any analysis.

For example, Figure 12 contains a section of an Inter-Process Precedence Graph, as it would appear immediately following IPPE insertion. The section corresponds to parts of two parallel processes, synchronizing themselves using one event variable, *ev*. Originally, *ev* has the value false, and no other processes are using it. The node numbering is chosen arbitrarily. The presence of an IPPE from node 3 to node 4 should indicate that in some sequences it is the execution of node 3 that allows for the completion of the wait at node 4. However, inspection of the code reveals that node 5 must execute before the wait at node 2 can complete, preventing node 3 from being reached until after the wait at node 4 is completed. The IPPE therefore violates condition 2, and should be removed. In addition, the IPPE from node 1 to node 6 should indicate that the wait at node 6 can complete at any time after node 1 has executed. However, node 1 must always execute before the wait at node 4 can complete, and hence its effect will be negated by node 5 before node 6 can be reached. This IPPE should also be removed, as it violates condition 3. Figure 13 contains the section of the Inter-Process Precedence Graph as it should appear, and inspection of the code will reveal that the execution sequencing enforced by the remaining IPPEs is genuine.

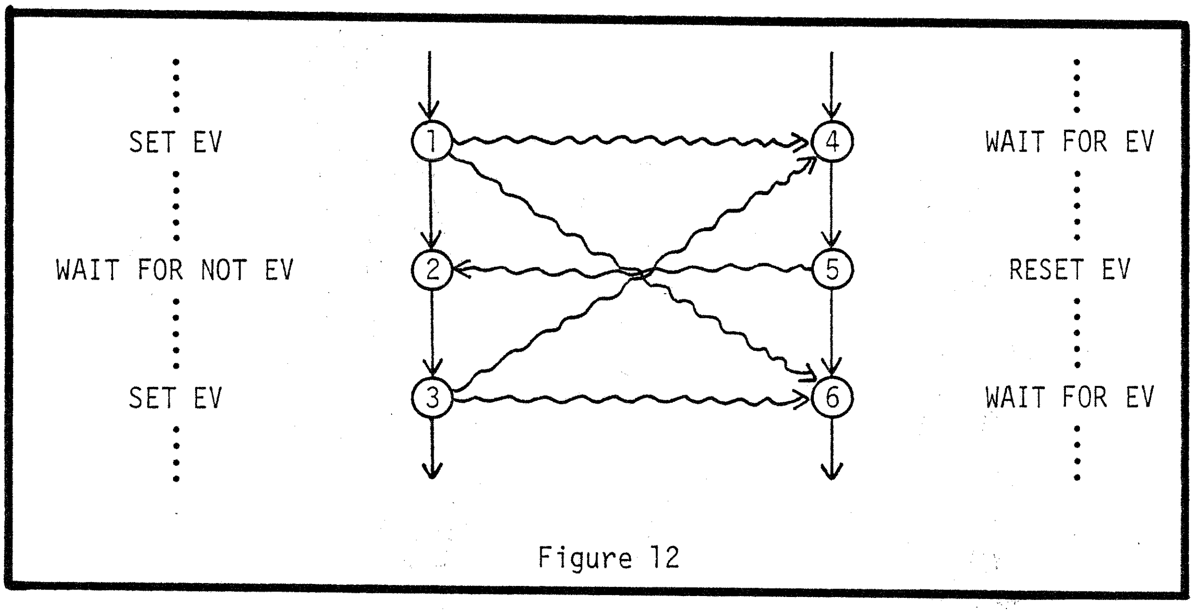


Figure 12

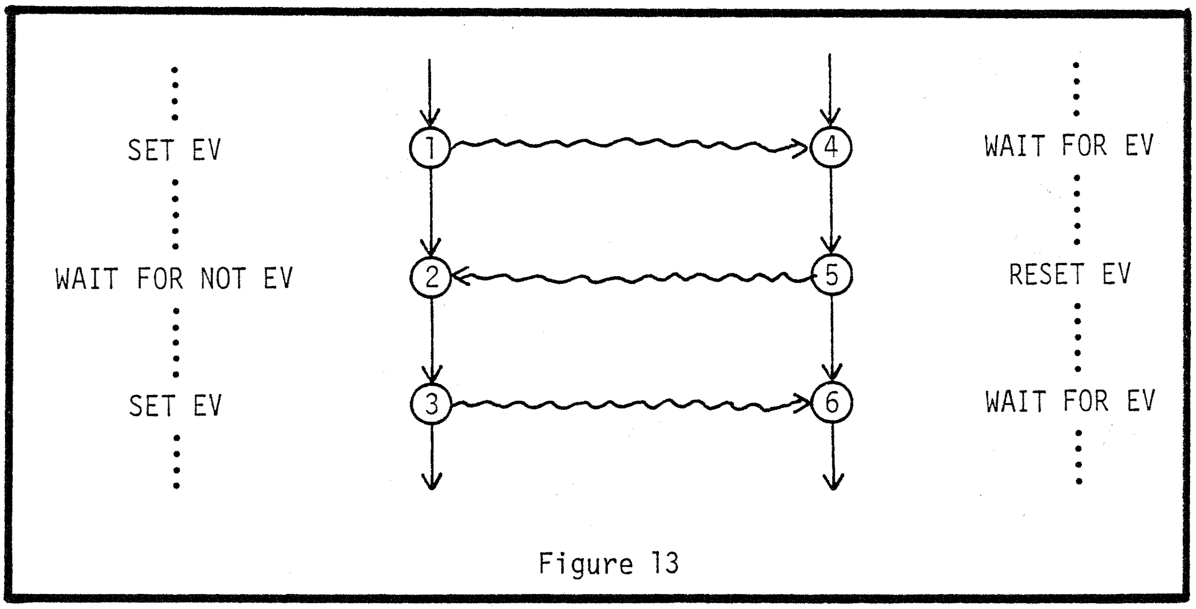


Figure 13

Spurious IPPEs can be removed by generating the execution sequence sets for the nodes in the graph. If, for any IPPE, the predecessor node is in the AFTER set or the NEVER set of the successor node, condition 2 is violated, and the IPPE is removed. If for any IPPE, a node negating the effect of the predecessor node occurs in the intersection of the ALWAYS_AFTER set of the predecessor node and the BEFORE set of the successor node, the IPPE violates condition 3, and is removed. The removal of an IPPE may alter the generated execution sequence sets, so these must be regenerated after an IPPE is removed. The process iterates until no more spurious IPPEs can be found. Note that the presence of spurious IPPEs acts to increase potential concurrency, so that the generated BEFORE, AFTER, and ALWAYS_AFTER sets will be subsets of the actual BEFORE, AFTER and ALWAYS_AFTER sets. This implies that the relative orderings we use are genuine, and only spurious IPPEs can be removed.

If, at any time during the spurious IPPE elimination, a node is found to be in its own BEFORE or AFTER set, this indicates the presence of a guaranteed deadlock in the code. The effect of the deadlock may permeate throughout the entire graph in an unpredictable manner, so the analysis should terminate at this point.

The pseudo-code for the spurious IPPE elimination phase appears in Appendix D.

VI. Data-Usage Anomaly Detection

The data-usage anomaly detection system will first be described in relation to the detection of intra-procedural and inter-procedural anomalies in programs containing no synchronization constructs. This will be followed by a discussion of the modifications made to incorporate concurrency into the analysis to enable detection of inter-process data-usage anomalies.

The single-process analysis system is designed to detect anomalous data flow patterns, symptomatic of programming errors, not only along paths within subprogram units but also along paths which cross unit boundaries. 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. Note that while a statement containing a subprogram invocation is represented as a single node, that node actually represents all the data actions which occur inside the called unit. Because of the order in which subprogram units are processed, the data flow information in the called unit can be passed across the boundary without placing its control structure at the point of invocation in the calling unit.

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 other, are processed first; then those units which invoke only processed units are analyzed in a backward order with the main program being processed last. In order to use this procedure, the call graph must be acyclic. If the call graph contains cycles, indicating recursion, analysis is terminated.

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, α and β will be placed in those path sets which represent a reference as the first data action at n. α will also be placed in those path sets representing an arbitrary sequence of actions followed by a definition. A variable γ 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 [FosL 76]. 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 subprogram 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 the 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 subprogram invocation is encountered at a node, path set information must be passed from the invoked unit to this node. First the path sets for the invoked routine 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 unit. Thus, the data actions which occur in the invoked subprogram 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 provides information which may be used for program documentation. This includes the order in which subprograms may be invoked, which variables must be assigned values before entry to a unit and which variables are actually assigned values there, as well as the subprogram's side effects with respect to global variables.

Now let us consider the effect of the inclusion of synchronization constructs upon the analysis. To analyze the usage of variables global to more than one process, we must consider the entire Flow And Precedence Graph at once. We cannot use the leafs-up ordering technique as we did for subprogram units in single-process programs since now the subgraphs for the units may contain IPPEs connecting them to other processes' subgraphs. Although that technique could still be used for those variables not participating in the concurrency, it would be preferable to be able to process all variables in parallel. This can be done by performing the analysis for all variables over the entire Flow And Precedence Graph, in which case the call graph would not be needed. However, it appears advantageous to integrate the leafs-up technique where possible to enable variable usage information gathered about subprograms to be compressed and inserted at each invocation point.

When performing data flow analysis on concurrent processes, paths through the flow graph give information on sequential patterns of references and definitions, but it is also necessary to know what other nodes in the graph could be executing concurrently with a given node. Therefore preliminary analysis must be performed upon the Flow and Precedence Graph to find the node sets CONCURRENT(n), ALWAYS_CONCURRENT(n), and POSSIBLY_CONCURRENT(n), as described in Section V. 5., for each node n.

The form of the path sets and the anomaly detection techniques are basically the same for multi-process as for single-process programs, but the data flow algorithms must be modified to work on the expanded process flow graph containing precedence edges. Now, predecessors and successors of a node may be in different processes.

Consider the graph segment in Figure 14. Assume that no usage of

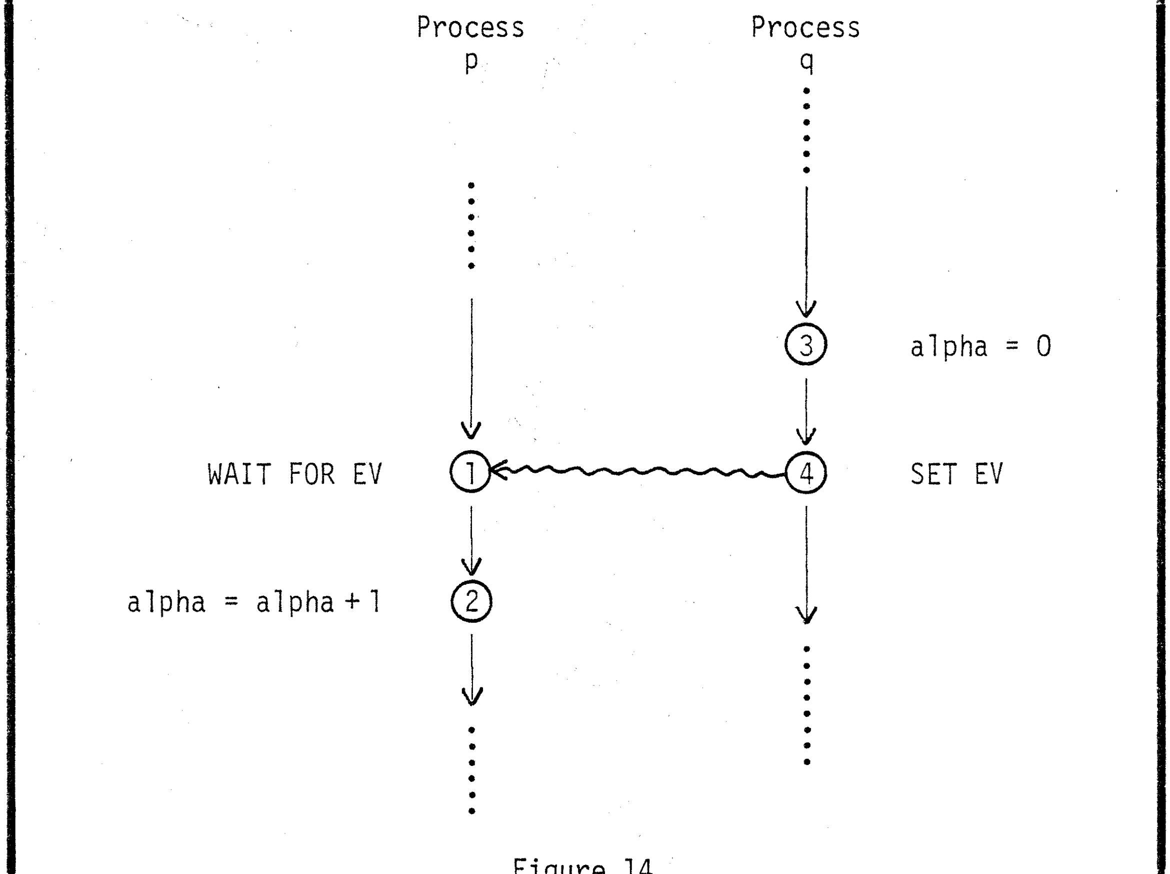


Figure 14

alpha has appeared prior to this segment. Node 1 must execute before node 2 and is the head of an IPPE originating in process q. Since a definition of *alpha* occurs on all paths into node 4, it will occur before the execution of node 1, and thus node 2. Therefore, the data flow along the IPPE (4,1) is treated differently from that along a regular flow graph edge. Similarly, in Figure 15, although it appears that *beta* is

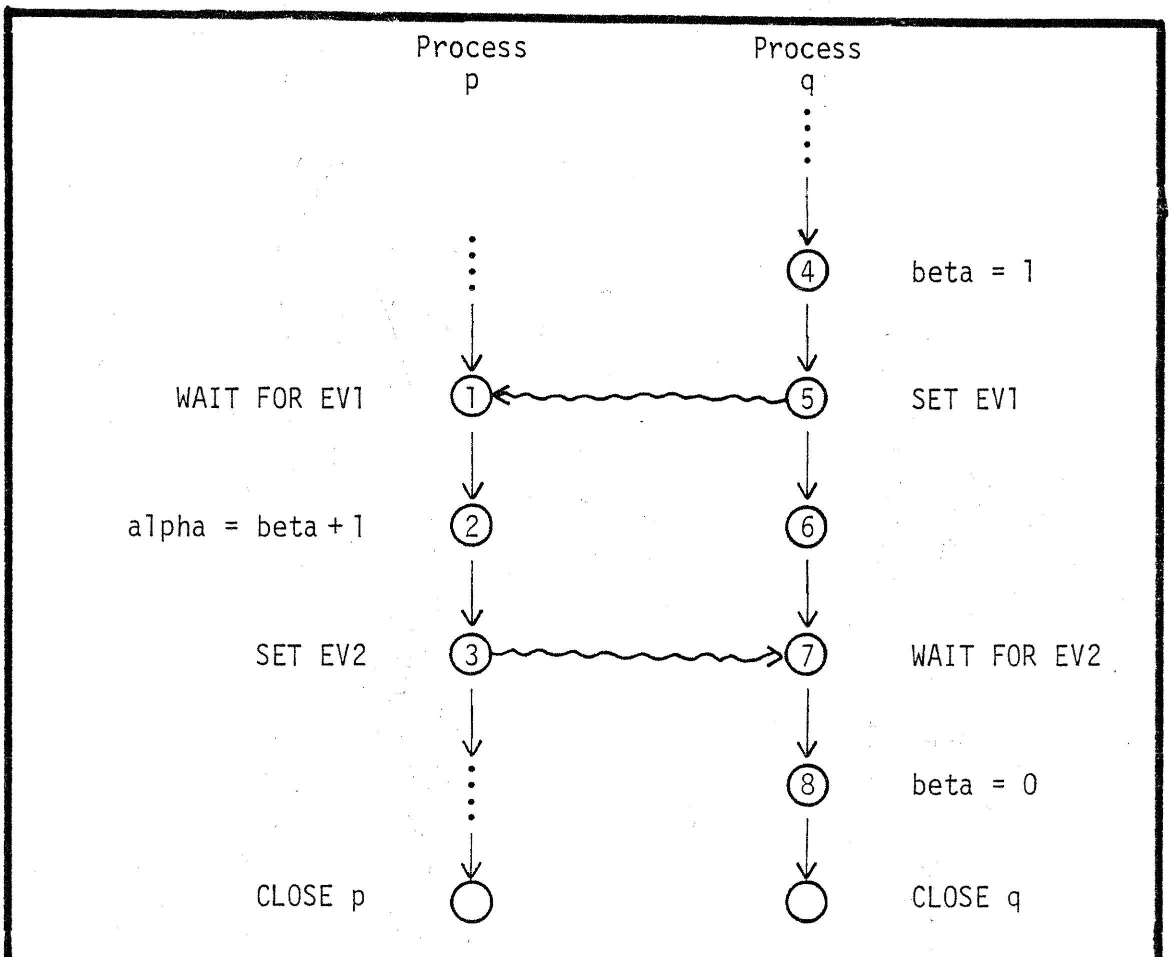


Figure 15

defined twice within q on the path through nodes 4, 5, 6, 7, 8 with no intervening reference, because of the IPPEs (5,1) and (3,7) the analysis will indicate that *beta* will always be referenced between the definitions.

The pseudo-code description of the data flow analysis phase in Appendix E is an expanded and modified version of that for single-process HAL/S programs[DreC 78]. It is designed to handle the leafs-up order analysis of a single-process program or the analysis of a multi-process program using the entire flow graph. The anomalies detected reflect the role of concurrency since we now consider actions which occur at nodes concurrent, always concurrent, or possibly concurrent to a given node as well as those actions occurring at the node.

The types of anomalies detected in single-process programs [DreC 78] are also applicable to multi-process programs. In addition, the concurrent node sets enable the detection of the possibility of references and definitions of variables at these nodes occurring in an unspecified order. Consider, for example, the situation in Figure 16.

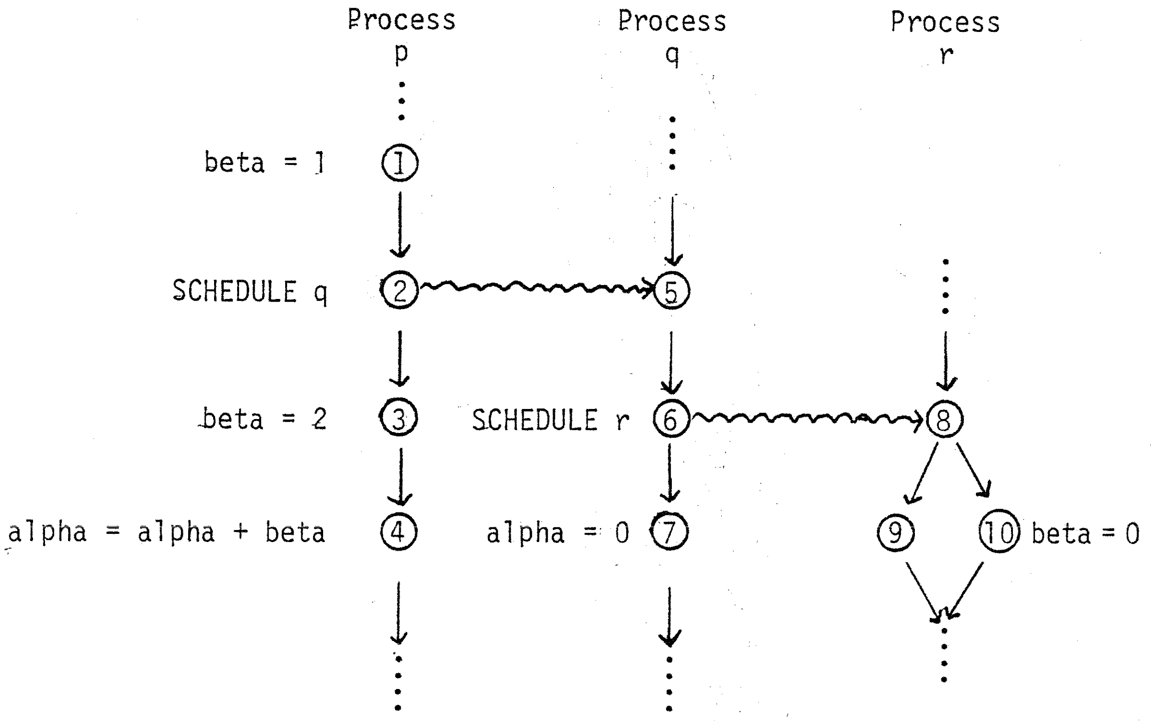


Figure 16

Here, *alpha* may or may not be defined when node 4 is executed since node 7 ε ALWAYS_CONCURRENT(4). This is a possible case of anomaly type (1), a reference to an uninitialized variable. Without considering what actions occur at concurrent nodes, we have a definite case of *alpha's* being uninitialized at node 4. Incorporating the ALWAYS_CONCURRENT node sets and making the assumption, for the sake of analysis, that actions occurring at concurrent nodes happen at the same time as the node's actions, we find that *alpha* will be defined before node 4 executes. Combining this with the previous information indicates the presence of the type (1) anomaly with the restriction that due to concurrency it may not actually occur during the program's execution.

This is also an example of a reference and a definition of a variable occurring at nodes whose execution is always unordered.

An example of anomaly type (2) also appears in Figure 16. Two definitions of a variable, with no intervening reference, occurs on the path 1, 2, 3 involving the variable *beta*. Another case of double definition, this time involving concurrency and representing a race condition, concerns *beta* at the concurrent nodes 3 and 10: the value of *beta* used in the computation at node 4 depends upon the order of execution of these nodes. Finally, anomaly type (3), the failure to subsequently reference a variable after defining it, is exemplified in Figure 15 by *beta*, last assigned a value at node 8.

VII. Synchronization Anomaly Detection

In addition to aiding in the search for data-usage anomalies, the execution sequence sets at each node can be used in the detection of potential synchronization anomalies. These are anomalies arising directly from potential concurrencies in the programs. It was encouraging to discover that all the synchronization anomalies we originally set out to detect can be found using these sets.

The first such anomaly is the potential for infinite waits, which includes deadlock as a subset. A process will wait indefinitely at a WAIT statement if the wait condition is false when the WAIT is reached during execution, and either no combination of statements will be executed in other processes that would set the wait condition to true, or all such combinations will be prevented from executing while waiting on this process (a deadlock).

The detection method involves considering each WAIT node in turn for legal execution sequences resulting in an infinite wait. Note that in order to produce reasonable time and space bounds for the algorithm, all possible combinations of loops and branches in a process are treated as legal execution sequences, even though some of these may constitute unexecutable paths due to the particular branch and loop conditions. This implies that all potential anomalies of this type will be discovered, but in addition some potential anomalies may be flagged where they do not exist.

The wait condition will have already been converted to conjunctive normal form during the insertion of IPPEs: For a potentially infinite wait, there must be at least one conjunct which can remain indefinitely false from the time the wait is started. This would require all the terms in that conjunct to remain indefinitely false. Therefore each conjunct, and each term in the conjunct, is checked for the potential to remain indefinitely false. If it cannot be proved that the wait is always finite, an anomaly is assumed.

The worst possible case is assumed while checking a term. It is assumed that a legal execution sequence exists in which the only nodes that occur setting the term to true must always execute if the WAIT is to be reached; i.e. those nodes from the ALWAYS set at the WAIT node. It is assumed that of those nodes, any that may not be able to execute until after the WAIT has completed, i.e. those from the POSSIBLY_AFTER set at the WAIT node, will have to wait until after the WAIT is completed. Further, it is assumed that all nodes setting the term to false not belonging to the AFTER set at the WAIT node will execute before the WAIT node is reached. Given these assumptions, and considering only those nodes which now will execute before the WAIT is reached, the term can be false for the duration of the WAIT if either no node remains setting it to true, or all such nodes can be followed by a node setting it back to false. One restriction is that the node setting the term back to false is not always subsequently followed by a node setting it to true.

Thus the wait at WAIT node w can be infinite if there exists a conjunct c in the wait expression such that

```
for each term t in c and
for each node $n_t$ setting t to true
    $n_T \varepsilon \{\text{ALWAYS}(w) - \text{POSSIBLY\_AFTER}(w)\}$
there exists a node $n_f$ setting t to false
    $n_f \varepsilon \{\text{everything} - \{\text{NEVER}(n_t) \cup \text{BEFORE}(n_t)\}\}$
        $\cap \{\text{everything} - \{\text{NEVER}(w) \cup \text{AFTER}(w)\}\}$
    and such that no node $n_t$ exists
    $n_t \varepsilon \{\text{ALWAYS\_AFTER}(n_f)\}$
        $\cap \{\text{everything} - \{\text{NEVER}(w) \cup \text{POSSIBLY\_AFTER}(w)\}\}$
```

The algorithm itself is a direct implementation of the above set expression.

Two other types of synchronization anomalies proved readily detectable from the execution sequence sets. The first of these is the possibility that a process may be rescheduled while it is still active. This involves checking the execution sequence sets at each schedule or close node. If at any of these nodes, a different schedule or close on the same process exists, but does not belong to the NEVER, BEFORE, or AFTER sets of the node being considered, then there is a potential anomaly. Further, if a different schedule for the same process appears in the BEFORE set of the open node, and the corresponding close is not also in the BEFORE set, this also signifies a potential anomaly. Due to the symmetry of the BEFORE and AFTER sets, the possibility that the same process must start after the open of the one being considered, but not after it has closed, will be discovered when considering the schedule and close for that instance of the process itself.

The second type of anomaly is the possibility of the premature termination of a process. Although this is not a violation of the rules of HAL/S, it may be indicative of a programming error. For instance, if a process which updates a database is terminated prematurely, it may leave the database in an inconsistent state. Checking for premature termination requires looking at the execution sequence sets at each terminate node. If the close node of any process that could be terminated by a particular terminate statement is in either the CONCURRENT set or the AFTER set of the terminate node, that process could be terminated prematurely.

The pseudo-code for these algorithms appears in Appendix F.

We anticipate that anomalies specific to other concurrent languages will also prove to be readily detectable from the execution sequence sets, although this work still remains to be done.

An additional area that we have been exploring is the checking of assertions. It is likely that, in certain circumstances, the system developer will wish to obtain information about the system that is unrelated to any specific anomaly, e.g., whether a particular execution ordering is forced, possible, or impossible.

The execution sequence sets may be used to test assertions about the time orderings of individual nodes. By examining the sets at the open and close nodes we can readily test assertions about whole processes. Other time-ordering assertions must be ultimately reducible to combinations of assertions about individual nodes.

VIII. Conclusion

The anomaly detection technique appears to provide an approach to software system analysis that does not suffer from many of the traditional problems of decidability and computational complexity. Its value is highly dependent on the ability to derive high-quality information concerning anomalies. However, the dual aims of obtaining high-quality information and using algorithms with pleasant computational complexity characteristics are sometimes in conflict. We have been successful so far in obtaining algorithms, but more, formal work is needed to determine the limits of this approach with respect to specific analysis problems.

We plan to expand the scope of our results by considering other languages within the class we have roughly delineated here. We expect this will bring us to considering the question of how best, with respect to specific language constructs and specific behavioral properties, to determine an abstract representation (akin to our present Flow And Precedence Graphs) which contains the information required for analysis.

We also plan to broaden the scope of the anomalies we can detect and enhance our system by the addition of anomaly definition capabilities.

IX. References

CamR 74	Campbell, R.A., and Habermann, A.N. The specification of process synchronization by path expressions. In _Lecture Notes in Computer Science_, _16_, Springer-Verlag, Heidelberg, 1974.
DreC 78	Drey, C. "DAVE-HAL/S: A System for the Static Data Flow Analysis of Single-Process HAL/S Programs," University of Colorado Technical Report CU-CS-141-78, November 1978.
FosL 76	Fosdick, L.D., and Osterweil, L.J. Data flow analysis in software reliability. _Computing Surveys_, _8_, 3 (September 1976), 305-330.
GreI 77	Greif, I. A language for formal problem specification. _Comm. ACM_, _20_, 12 (December 1977), 931-935.
HabA 75	Habermann, A.N., Path expressions. Computer Sci. Dept., Carnegie-Mellon Univ., Pittsburgh, June 1975.
Inte 76	_HAL/S Language Specification_. Intermetrics, Inc., Cambridge, Massachusetts, June 1976.
OdgW 78	Ogden, W.F., Riddle, W.E., and Rounds, W.C. Complexity of expressions allowing concurrency. _Proc. Fifth ACM Symp. on Prin. of Programming Languages_, Tucson, January 1978, pp. 185-194.
OstL 76	Osterweil, L.J., and Fosdick, L.D., "DAVE - A Validation Error Detection and Documentation System for Fortran Programs," _Software - Practice and Experience_, _6_, (1976), 473-486.
PetJ 74	Peterson, J.L., and Bredt, T.H. A comparison of models of parallel computation. _Proc. IFIP Congress 74_, Stockholm, August 1974, pp. 466-470.
PetJ 76	Peterson, J.L. Computation sequence sets. _J. of Comp. and Sys. Sci._, _13_, 1 (August 1976), 1-24.
PetJ 78	Peterson, J.L. Petri Nets. Dept. of Computer Sci., Univ. of Texas, Austin, August 1978.
ReiJ 78	Reif, J.H. Analysis of communicating processes. TR 30, Computer Sci. Dept., Univ. of Rochester, New York, May 1978.
RidW 72	Riddle, W.E. The hierarchical modelling of operating system structure and behavior. _Proc. ACM 72 National Conf._, Boston, August 1972, pp. 1105-1127.
RidW 73	Riddle, W.E. A design methodology for complex software systems. _Proc. Second Texas Conf. on Computing Systems_, Austin, November 1973, pp. 22.1-22.8.

RidW 74	Riddle, W.E. The equivalence of Petri nets and message transmission models. SRM/97, Computing Lab., Univ. of Newcastle upon Tyne, England, August 1974.
RidW 78a	Riddle, W.E. An approach to software system modelling and analysis. To appear: _J. of Computer Languages_.
RidW 78b	Riddle, W.E. An approach to software system behavior description. To appear: _J. of Computer Languages_.
SaxA 77	Saxena, A. The static detection of deadlocks. CU-CS-122-77, Dept. of Computer Sci., Univ. of Colorado at Boulder, November 1977.
ShaA 78	Shaw, A.C. Software descriptions with flow expressions. _IEEE Trans. on Software Engineering_, _SE-4_, 3 (May 1978), 242-254.
TayR 78	Taylor, R.N., and Osterweil, L.J. A facility for verification, testing and documentation of concurrent process software. _Proc. Compsac 78_, Chicago, November 1978, pp. 36-41.
WilJ 78	Wileden, J.C. Modelling parallel systems with dynamic structure. RSSM/71 (Ph.D. Thesis), Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, January 1978.

APPENDIX A

Pseudo-code for Building Flow Graph

The pseudo-code in this appendix uses the syntax defined in Caine and Gordon, "PDL - A tool for software Design," Proc, 1975 National Computer Conference, June 1975, pp. 271-276.

Tree of Inter-Segment References

```
SEGMENT flow graph builder

****************************************************************
*                                                              *
*      This segment builds a flow graph from the parse tree    *
*                                                              *
*      and symbol table for a given program.                   *
*                                                              *
****************************************************************

   DO   for each program unit

        create (subgraph) named (program unit name)
        create (entry) node ($n_e$)
        create (exit) node ($n_x$)
        end nodes = { }
        build flow graph segment (program unit body,{$n_e$}, end nodes)
        create edges from (end nodes) to ($n_x$)

   ENDDO

END
```

```
SEGMENT build flow graph segment (statement, start nodes, end nodes)

*****************************************************************
*                                                               *
*     This recursive segment produces a flow graph segment     *
*                                                               *
*     for a single statement at a time.                         *
*                                                               *
*****************************************************************

    IF  statement type is simple statement

        create (statement type) node (n)
        create edges from (start nodes) to (n)
        end nodes = {n}
    ELSE

        DO  case of statement type

            if statement:
                create (if) node (i)
                create edges from (start nodes) to (i)
                end then = { }
                end else = { }
                build flow graph segment (then part, {i}, end then)
                IF  there is an "ELSE" part
                   build flow graph segment (else part, {i}, end else)
                ENDIF
                end nodes = end then ∪ end else

            do case statement:
                create (do case) node (dc)
                create edges from (start nodes) to (dc)
                end nodes = { }
                DO  for each case part
                   end case = { }
                   build flow graph segment (case part, {dc}, end case)
                   end nodes = end nodes ∪ end case

                ENDDO
                IF  there is an "ELSE" part
                   end else = { }
                   build flow graph segment (else part, {dc}, end else)
                   end nodes = end nodes ∪ end else
    ENDIF
do statement:
    S = start nodes
    DO    for each statement
          end statement = { }
          build flow graph segment (statement, s, end state-
            ment)
          S = end statement
    ENDDO
    end nodes = end statement
do while statement:
    create (do test) node (dt)
    create edges from (start node) to (dt)
    end while = { }
    build flow graph segment (do part, {dt}, end while)
    end nodes = end while ∪ {dt}
do until statement:
    end until = { }
    build flow graph segment (do part, start nodes, end until)
    create (do test) node (dt)
    create edges from (end until) to (dt)
    end until = { }
    build flow graph segment (do part, {dt}, end until)
    end nodes = end until ∪ {dt}
do for statement:
    create (do initialization) node (di)
    create edges from (start nodes) to (di)
    create (do test) node (dt)
    create edges from ({di}) to (dt)
    end for = { }
    build flow graph segment (do part, {dt}, end for)
    create (do successor) node (ds)
    create edges from (end for) to (ds)
    end nodes = {dt, ds}
```

```
            ENDDO
        ENDIF
    END
```

APPENDIX B

Pseudo-code for Building the Inter-Process Precedence Graph and the Flow and Precedence Graph

The pseudo-code in this appendix uses the syntax defined in Caine and Gordon, "PDL - A tool for software Design," _Proc_, _1975 National Computer Conference_, June 1975, pp. 271-276.

Tree of Inter-Segment References

Page	Segment
49	focus
50	insert
50	inject

```
SEGMENT focus
****************************************************************
*   This segment builds the Synchronization and Control Flow   *
*   Graph from the Flow Graph, by removing nodes which do not  *
*   represent synchronization constructs, or control struc-    *
*   tures which affect their flow of execution.                *
****************************************************************
    DO   for every node $n_i \not\in \{n_e, n_x\}$
         IF   $n_i$ is not a structured node or a synchronization node
              create edge from (predecessor of $n_i$) to (successor of $n_i$)
              delete edge from (predecessor of $n_i$) to ($n_i$)
              delete edge from ($n_i$) to (successor of $n_i$)
              delete node ($n_i$)
         ENDIF
    ENDDO
    DO   for every node $n_i \not\in \{n_e, n_x\}$
         IF   $n_i$ is a structured node
              IF  $n_i$ has exactly one successor node
                  create edge from (predecessor of $n_i$) to (successor of $n_i$)
                  delete edge from (predecessor of $n_i$) to ($n_i$)
                  delete edge from ($n_i$) to (successor of $n_i$)
                  delete node ($n_i$)
              ENDIF
         ENDIF
    ENDDO
  END
```

```
SEGMENT insert

******************************************************************
*                                                                *
*   This segment builds the Inter-Process Precedence Graph, by   *
*   inserting IPPEs into the Synchronization and Control Flow    *
*   Graph.                                                       *
*                                                                *
******************************************************************

  DO   for every node $n_i \not\in \{n_e, n_x\}$
       IF   $n_i$ is a SET or RESET node
            DO   for every node $n_j \not\in \{n_e, n_x\}$
                 IF   $n_j$ is a WAIT node involving the event SET (RESET)
                      by $n_i$ create IPPE from ($n_i$) to ($n_j$)
                 ENDIF
            ENDDO
       ENDIF
  ENDDO
END
```

```
SEGMENT inject

*****************************************************************
*                                                               *
*        This segment injects the correct IPPEs from the        *
*        Inter-Process Precedence Graph into the Flow Graph,    *
*        producing the Flow and Precedence Graph.               *
*                                                               *
*****************************************************************

  DO   for each IPPE = ($n_i$,$n_j$) in IPPG
       create IPPE from $n_i$ to $n_j$ in the Flow Graph
  ENDDO
END
```

APPENDIX C

Pseudo-code for
Determination of the
Execution Sequence Sets

The pseudo-code in this appendix uses the syntax defined in Caine and Gordon, "PDL - A tool for software Design," Proc, 1975 National Computer Conference, June 1975, pp. 271-276.

Tree of Inter-Segment References

Page	Segment
53	calculate execution sequence sets
53	calculate ALWAYS sets
54	calculate ALWAYS_BEFORE_WITHIN_PROCESS and BEFORE_WITHIN_PROCESS sets
55	calculate ALWAYS_AFTER_WITHIN_PROCESS and AFTER_WITHIN_PROCESS sets
56	calculate ALWAYS_SUBPROCESS and SUBPROCESS sets
57	calculate final ALWAYS sets
53	calculate NEVER sets
58	calculate NEVER_WITHIN_PROCESS sets
59	calculate final NEVER sets
60	CALCULATE_BEFORE (node,process)
62	calculate AFTER sets
62	calculate ALWAYS_BEFORE and ALWAYS_AFTER sets
63	calculate POSSIBLY_BEFORE and POSSIBLY_AFTER sets
63	calculate CONCURRENT, ALWAYS_CONCURRENT and POSSIBLY_CONCURRENT sets

```
SEGMENT calculate execution sequence sets

calculate ALWAYS sets
calculate NEVER sets
DO   for all nodes n in flow graph
     BEFORE(n) = Ø

ENDDO

DO   for all nodes n in flow graph in BFS (breadth first search order)
     BEFORE(n) = CALCULATE_BEFORE(n,p)
ENDDO

calculate AFTER sets

calculate ALWAYS_BEFORE and ALWAYS_AFTER sets
calculate POSSIBLY_BEFORE and POSSIBLY_AFTER sets
calculate CONCURRENT, ALWAYS_CONCURRENT, and POSSIBLY_CONCURRENT sets

END
```

```
SEGMENT  calculate ALWAYS sets

calculate  ALWAYS_BEFORE_WITHIN_PROCESS and BEFORE_WITHIN_PROCESS sets
calculate  ALWAYS_AFTER_WITHIN_PROCESS and AFTER_WITHIN_PROCESS sets
calculate  ALWAYS_SUBPROCESS and SUBPROCESS sets
calculate  final ALWAYS sets

END
```

```
SEGMENT  calculate NEVER sets

calculate  NEVER_WITHIN PROCESS sets
calculate  final NEVER sets

END
```

```
SEGMENT calculate ALWAYS_BEFORE_WITHIN_PROCESS and
                         BEFORE_WITHIN_PROCESS sets

DO   for all processes p in the flow graph

     DO   for all nodes n in p in breadth-first order
          ABWP = ALWAYS_BEFORE_WITHIN_PROCESS (e) for any node
               e such that there exists a flow graph edge (e,n)
          BWP = BEFORE_WITHIN_PROCESS (e)

          DO   for all nodes f ≠ e such that there exists a
               flow graph edge (f,n)
               ABWP = ABWP intersection ALWAYS_BEFORE_WITHIN_PROCESS(f)
               BWP = BWP union BEFORE_WITHIN_PROCESS (f)
          ENDDO

          ALWAYS_BEFORE_WITHIN_PROCESS (n) = ABWP
          BEFORE_WITHIN_PROCESS (n) = BWP
     ENDDO

ENDDO

END
```

```
SEGMENT calculate ALWAYS_AFTER_WITHIN_PROCESS and
                          AFTER_WITHIN_PROCESS sets

DO for all processes p in the flow graph

     DO for all nodes n in p in bottom-up breadth-first order
          AAWP = ALWAYS_AFTER_WITHIN_PROCESS (e) for any node e
               such that there exists a flow graph edge (e,n)
          AWP = AFTER_WITHIN_PROCESS (e)

          DO for all nodes f ≠ e such that there exists a flow graph
             edge (f,n)
             AAWP = AAWP intersection ALWAYS_AFTER_WITHIN_PROCESS (f)
             AWP = AWP union AFTER_WITHIN_PROCESS (f)
          ENDDO

          ALWAYS_AFTER_WITHIN_PROCESS (n) = AAWP
          AFTER_WITHIN_PROCESS (n) = AWP
     ENDDO

ENDDO

END
```

```
SEGMENT calculate ALWAYS_SUBPROCESS and SUBPROCESS sets

DO  for all processes p in the flowgraph in reverse breadth-first-
    search order
    ASS = { }

    DO for all processes q such that SCHEDULE (q) belongs to
       ALWAYS_AFTER_WITHIN_PROCESS (open node of p)
       ASS = ASS union ALWAYS_SUBPROCESS_SET (q)
       ASS = ASS union q
    ENDDO

    ALWAYS_SUBPROCESS_SET (p) = ASS
    SS = { }

    DO for all processes q such that SCHEDULE (q) belongs to
       AFTER_WITHIN_PROCESS (open node of p)
       SS = SS union SUBPROCESS_SET (q)
       SS = SS union q
    ENDDO

    SUBPROCESS_SET (p) = SS
ENDDO

END
```

```
SEGMENT calculate final ALWAYS sets
DO for each process p in the flow graph in breadth-first order
     IF   P = main program
          A = OPEN node of main
          A = A union ALWAYS_AFTER_WITHIN_PROCESS (open node of main)
          DO for all processes q belonging to ALWAYS_SUBPROCESS_SET(main)
             A = A union (open node of q)
             A = A union ALWAYS_AFTER_WITHIN_PROCESS (open node of q)
          ENDDO
          ALWAYS (open node of main) = A
     ELSE ALWAYS (open node of p) = ALWAYS (SCHEDULE(p) node)
     ENDIF
     DO   for all nodes n in p apart from open node of p
          A = n
          A = A union ALWAYS_BEFORE_WITHIN_PROCESS (n)
          A = A union ALWAYS_AFTER_WITHIN_PROCESS (n)
          A = A union ALWAYS (open node of p)
          DO  for each process q such that SCHEDULE (q) belongs to
              ALWAYS_BEFORE_WITHIN_PROCESS (n) union n union
              ALWAYS_AFTER_WITHIN_PROCESS (n) minus
              ALWAYS_AFTER_WITHIN_PROCESS (open node of p)
              DO for each process r belonging to ALWAYS_SUBPROCESS_SET (q)
                   A = A union (open node of r)
                   A = A union ALWAYS_AFTER_WITHIN_PROCESS (open node of r)
              ENDDO
              A = A union (open node of q)
              A = A union ALWAYS_AFTER_WITHIN_PROCESS (open node of q)
          ENDDO
          ALWAYS (n) = A
     ENDDO
ENDDO
END
```

```
SEGMENT calculate NEVER_WITHIN_PROCESS sets
DO   for all processes p in the flow graph
     DO   for all nodes n in p
          NWP = all nodes in p
          NWP = NWP minus BEFORE_WITHIN_PROCESS (n)
          NWP = NWP minus n
          NWP = NWP minus AFTER_WITHIN_PROCESS (n)
          NEVER_WITHIN_PROCESS (n) = NWP
     ENDDO
ENDDO
END
```

```
SEGMENT calculate final NEVER sets
DO for each process p in the flow graph in breadth-first order
     IF   p not the main program
          NEVER (open node of p) = NEVER (SCHEDULE(p)node)
     ENDIF
     DO   for each node n in p apart from open node of p
          N = NEVER (open node of p)
          N = N union NEVER_WITHIN_PROCESS (n)
          DO   for each process q such that SCHEDULE (q) belongs to
               NEVER_WITHIN_PROCESS (n)
               N = N union all nodes in q
               DO   for each process r belonging to SUBPROCESS_SET (q)
                    N = N union all nodes in r
               ENDDO
          ENDDO
          NEVER (n) = N
     ENDDO
ENDDO
END
```

```
SEGMENT CALCULATE_BEFORE(m,p)

     ***************************
     *                         *
     * m is a node in process p *
     *                         *
     ***************************

N =  {all nodes} = identity
IF   $\exists(n,p) \in$ SAVE $\rightarrow$ [$n \in$ NEVER(m) $\cup$ {ancestors of m}]
     REDO = SAVE from entry after (n,p) to top
     CALCULATE_BEFORE = N
     RETURN
ENDIF
IF BEFORE(m) = { }
     CALCULATE_BEFORE=BEFORE(m)
     RETURN
ENDIF
Push (m,p) on SAVE
BEFORE_SET = BEFORE_WITHIN_PROCESS(m)
E = {$e \in$ process p $|\exists$ edge(e,n)}
IF E $\neq$ { }
     TEMP = N
     DO $\forall e \in E$
          TEMP = TEMP $\cap$ (CALCULATE_BEFORE(e,p) $\cup$ NEVER(e))
     ENDDO
     I = {$k \in$ process q $|\exists$ IPPE(k,m)}
     IF   I $\neq$ { }
          TEMPX = N
          DO $\forall k \in I$
               TEMPX = TEMPX $\cap$ (CALCULATE_BEFORE(k,q) $\cup$ {k}
                                        $\cup$ NEVER(k))
          ENDDO
     ENDIF
ENDIF
```

```
BEFORE_SET = BEFORE_SET ∪ TEMP ∪ TEMPX
IF m ε REDO
     BEFORE(m) = BEFORE_SET
ELSE remove m from REDO
ENDIF
Pop SAVE
END
```

```
SEGMENT calculate AFTER sets
DO   for all nodes n in flow graph
     DO for all nodes m $\in$ BEFORE(n)
          AFTER(m) = AFTER(m) $\dot{\cup}$ {n}
     ENDDO
ENDDO
END
```

```
SEGMENT calculate ALWAYS_BEFORE and ALWAYS_AFTER sets
DO   for all nodes n in flow graph
     ALWAYS_BEFORE(n) = ALWAYS(n) $\cap$ BEFORE(n)
     ALWAYS_AFTER(n) = ALWAYS(n) $\cap$ AFTER(n)
ENDDO
END
```

```
SEGMENT  calculate POSSIBLY_BEFORE and POSSIBLY_AFTER sets

DO   for all nodes n in flow graph
     POSSIBLY_BEFORE(n) = BEFORE(n)
     POSSIBLY_AFTER(n) = { }
ENDDO

DO   for all nodes n in flow graph in BFS order
     DO for all entry nodes e to n
          POSSIBLY_BEFORE(n) = POSSIBLY_BEFORE(n) ∪ {e} ∪ POSSIBLY_
                                                        BEFORE(e)
     ENDDO
ENDDO

DO   for all nodes n in flow graph
     DO for all nodes m ε POSSIBLY_BEFORE(n)
          POSSIBLY_AFTER(m) = POSSIBLY_AFTER(m) ∪ {n}
     ENDDO
ENDDO

END


SEGMENT calculate CONCURRENT, ALWAYS_CONCURRENT, and POSSIBLY_CONCURRENT
        sets

DO   for all nodes n in flow graph

     CONCURRENT(n) = {all nodes} - {NEVER(n) ∪ POSSIBLY_BEFORE(n)
                          ∪ POSSIBLY_AFTER(n)}

     ALWAYS_CONCURRENT(n) = ALWAYS(n) ∩ CONCURRENT(n)

     POSSIBLY_CONCURRENT(n) = {(POSSIBLY_BEFORE(n) - BEFORE(n))
                               ∪ (POSSIBLY_AFTER(n) - AFTER(n))}

ENDDO

END
```

APPENDIX D

Pseudo-code for Removal of Spurious IPPEs

The pseudo-code in this appendix uses the syntax defined in Caine and Gordon, "PDL - A tool for software Design," Proc, 1975 National Computer Conference, June 1975, pp. 271-276.

```
SEGMENT remove spurious IPPEs

DO   until no more IPPEs can be removed
     calculate BEFORE sets
     calculate AFTER sets
     calculate ALWAYS_AFTER sets
     calculate NEVER sets

     DO   for each IPPE in the flow graph

          IF   predecessor node belongs to AFTER (successor node)
          OR   predecessor node belongs to NEVER (successor node)
          OR   there exists a node negating the predecessor node
               belonging to ALWAYS_AFTER (predecessor node) union
               BEFORE (successor node)
               remove IPPE from flow graph
          ENDIF

     ENDDO

ENDDO

END
```

APPENDIX E

Pseudo-code for Data-usage Anomaly Detection

The pseudo-code in this appendix uses the syntax defined in Caine and Gordon, "PDL - A tool for software Design," _Proc_, _1975 National Computer Conference_, June 1975 pp. 271-276.

Tree of Inter-Segment References

```
SEGMENT  Data Flow Analysis Driver

Process call graph for cycles (and leafs-up ordering if single-process
program)

IF cycles are present in call graph
    Output message ("Illegal recursion in program")
    STOP

ENDIF

DO for all blocks specified by template

    Form path sets for template
    Make entry in master table
    Output message ("Template used for (block name)")

ENDDO

IF single-process program
    DO for each block in leafs-up order
        Get flow graph for block
        Build path sets (flow graph)
        Report anomalies (flow graph)
        Make entry in master table
    ENDDO
ELSEIF multi-process program
    Get flow graph for program
    Build path sets (flow graph)
    Report anomalies (flow graph)
ENDIF

END
```

<u>SEGMENT</u> 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 { }
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) ∉ TREE
             IF w ≤ v < w + DESCENDANTS(w)
                 cycles = true
             ENDIF
          ENDIF
     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
PREORDER(v) = i

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

DESCENDANTS(v) = i - PREORDER(v) + 1
j = j + 1
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 (flow graph)
DO for each subprogram unit in flow graph
   $G(graph_{unit}) = \{ \}$
   $I(graph_{unit}) = \{all\ variables\}$
ENDDO
DO for each node in flow graph
   Initialize SIDEFCT = REF = DEF = { }
   Set up an initialize path sets:
   $A_x(node)$, $B_x(node)$, $C_x(node)$,
   $D_x(node)$, $E_x(node)$, $F_x(node)$,
   I(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
```

```
***********
* SIDEFCT *
* used for *
* detection*
* of side  *
* effects. *
***********
```

```
*********************
* Path set G has been *
* added to the path   *
* sets described in   *
* [FosL 76] to be able*
* to identify varia-  *
* bles which are de-  *
* fined anywhere in   *
* the unit.           *
*********************
```

```
  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
    ENDIF
  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
    ENDIF
  ENNDO

Procedure, function or process 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
    ENDIF
  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, function or process:
  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 (referenc-
  ing) path sets

FILE output:
  Place (variables in left-hand-side file expression) in (referenc-
  ing) path sets
  Place (variables in right-hand-side expression) in (referencing)
  path sets
```

```
        Other:
          Ignore
        ENDDO
$C_X$ (node) = $C_X$ (node) $\cup$ $A_X$ (node)
$F_X$ (node) = $F_X$ (node) $\cup$ $D_X$ (node)
I  (node) = {all variables} - ($A_X$ (node) $\cup$ $B_X$ (node) $\cup$ $C_X$ (node) $\cup$ $D_X$ (node) $\cup$ $F_X$ (node))
G $\left(\text{graph}_{\text{unit}}\right)$ = G $\left(\text{graph}_{\text{unit}}\right)$ $\cup$ $C_d$ (node) $\cup$ $F_d$ (node), where node $\in$ unit
        IF node $\neg$ entry or exit of unit
          I $\left(\text{graph}_{\text{unit}}\right)$ = I $\left(\text{graph}_{\text{unit}}\right)$ $\cap$ I (node)
        ENDIF
        IF SIDEFCT $\neg$ empty
          DO for each variable in SIDEFCT
            Output message ("A possible side effect has been detected in
            this statement involving (variable)")
          ENDDO
        ENDIF
ENDDO
Determine path sets for (flow graph)

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 { }
    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
            ENDIF


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

        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
                Associate formal parameter to argument
                IF single-process program
                    Pass path set membership of (parameter)
                    to (argument)
                    Report non-usage of (argument) corresponding
                    to (parameter)
                    Report side effects involving (argument)
                ENDIF
            ENDIF
        ENDDO

        IF single-process program
            Pass path set membership of global variables over
            block boundary
            Output message (documentation information on global
            variable usage in invoked block)
        ENDIF

    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
            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 expres-  *
*  sion, 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.                                                   *
*                                                                      *
************************************************************************
    SIDEFCT = (TEMPREF ∩ DEF) ∪ SIDEFCT
    SIDEFCT = (TEMPDEF ∩ REF) ∪ SIDEFCT
    REF = REF ∪ TEMPREF
    DEF = DEF ∪ TEMPDEF
ENDDO
END
```

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

```
**********************************************************************
*                                                                    *
*      This segment passes the path set membership of a formal para-*
*                                                                    *
*  meter in an invoked procedure to the corresponding actual argu-   *
*                                                                    *
*  ment 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 $\varepsilon$ PATHSET (graph$_{called}$)
        Enter (argument) in (PATHSET(node$_{caller}$)) path set
    ENDIF
ENDDO
IF parameter $\varepsilon$ G(graph$_{called}$)
    Enter (argument) in (G(graph$_{caller}$)) path set
    Enter (argument) in (TEMPDEF) path set
ENDIF
IF parameter $\varepsilon$ $C_r$ (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 $\varepsilon$ I(graph$_{called}$)
          Output message ("(Argument) specified as input argument is
          not referenced in (called)")
     ENDIF
ELSEIF parameter $\not\varepsilon$ 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 ε G (graph_called)
        Output message ("Side effect condition - actual input argu-
        ment is used by its global name in (called block) and is de-
        fined 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 ε $A_X$, $B_X$, $C_X$, $D_X$, $F_X$ or G for X = r, d, u for
       nodes ε AFTER($n_0$) ∩ BEFORE($n_{exit}$) in called block
        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 argu-
ment 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 $\varepsilon$ PATHSET (graph $_{called}$ )
            Enter (var) in (PATHSET (node$_{caller}$))
        ENDIF
    ENDDO
    IF var $\varepsilon$ G (graph $_{called}$)
        Enter (var) in (G(graph $_{caller}$)) path set
        Enter (var) in (TEMPDEF) path set
    ENDIF
    IF var $\varepsilon$ $C_r$ (graph $_{called}$) or $F_r$ (graph$_{called}$)
        Enter (var) in (TEMPREF) path set
    ENDIF
ENDDO
END
```

```
SEGMENT  Determine path sets  (flow graph)

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
                                        ****************************************
                                        * The (graph) path sets are calculated *
                                        * without considering concurrency      *
                                        * since they are primarily used for    *
                                        * single process analysis.             *
                                        ****************************************
Set up and initialize path sets:
      $A_x(graph)$, $B_x(graph)$, $C_x(graph)$,
      $D_x(graph)$, $E_x(graph)$, $F_x(graph)$,
      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                             *    Determine $A_x(graph)$ and   *
                                        *                               *
      DO  for each node in the flow graph  * $A'_x(n-->)$, $A''_x(n-->)$, $A'''_x(n-->)$ *
                                        *********************************

            IF Node type is not exit
                 NULL(node)  =  I(node) ∪ $B_x(node)$
                 KILL(node)  =  $A_x(node)$
                 GEN(node) = {all variables} - (KILL(node) ∪ NULL(node))
            ELSE
                 NULL(node) = { }
                 KILL(node) = { }
                 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 n in flow graph except exit
    DO for each node m $\varepsilon$ ALWAYS_CONCURRENT(n)
        KILL(n) = KILL(n) $\cup$ KILL(m)
        GEN(n) = GEN(n) $\cup$ GEN(m)
    ENDDO
ENDDO

Execute LIVE on graph

DO for each node in flow graph
    $A''_x$(n-->) = {all variables}-LIVE(node)
ENDDO

DO for each node n in flow graph except exit
    DO for each node m $\varepsilon$ {CONCURRENT(n) - ALWAYS_CONCURRENT(n)}
        KILL(n) = KILL(n) $\cup$ KILL(m)
        GEN(n)  = GEN(n)  $\cup$ GEN(m)
    ENDDO
ENDDO

Execute LIVE on graph

DO for each node in 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) $\cup$ $A_z$(node))
        NULL(node) ={all variables}- (GEN(node) $\cup$ KILL(node))
ENDDO
                                  ********************************
                                  *                              *
                                  *  Determine $C_x$(graph), and    *
                                  *  $C'_x$(n-->), $C''_x$(n-->), $C'''_x$(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
```

```
DO for each node n in flow graph
    DO for each node m $\varepsilon$ ALWAYS_CONCURRENT(n)
        GEN(n) = GEN(n) $\cup$ GEN(m)
        KILL(n) = KILL(n) $\cup$ KILL(m)
    ENDDO
ENDDO

Execute LIVE on graph

DO for each node in flow graph
    $C''_X$(n-->) = LIVE(node)
ENDDO

DO for each node n in flow graph
    DO for each node m $\varepsilon$ {CONCURRENT(n) - ALWAYS_CONCURRENT(n)}
        GEN(n) = GEN(n) $\cup$ GEN(m)
        KILL(n) = KILL(n) $\cup$ KILL(m)
    ENDDO
ENDDO

Execute LIVE on graph

DO for each node in flow graph
    $C'''_X$(n-->) = LIVE(node)
ENDDO
                                              ************************
                                              *                      *
                                              *  Determine $B_X$(graph)  *
                                              *                      *
DO for each node in the flow graph            ************************
    NULL(node) = I(node) $\cup$ $B_X$(node)
    KILL(node) = $A_X$(node)
    GEN(node) ={all variables}- (KILL(node) $\cup$ NULL(node))
ENDDO
Execute LIVE on graph
$B_X$(graph) = ({all variables}- LIVE(entry node)) $\cap$ ({all variables}-
               $A_X$(graph)) $\cap$ $C_X$(graph)
```

```
                                    ***********************************
                                    *     Determine $D_x$(graph) and       *
                                    *$D'_x$ (--> n), $D''_x$ (--> n), $D'''_x$(--> n)*
                                    ***********************************
DO for each node in the flow graph
     GEN(node) = $D_x$(node)
     KILL(node) = ($F_y$(node) $\cup$ $F_z$(node))
     NULL(node) = {all variables}- (GEN(node) $\cup$ 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 flow graph
     DO for each node m $\varepsilon$ ALWAYS_CONCURRENT(n)
        GEN(n) = GEN(n) $\cup$ GEN(m)
        KILL(n) = KILL(n) $\cup$ KILL(m)
   ENDDO
ENDDO

Execute LIVE on flow graph
DO for each node in flow graph
     $D''_x$(--> n) = AVAIL(node)
ENDDO

DO for each node in flow graph
     DO for each node m $\varepsilon$ {CONCURRENT(n) - ALWAYS_CONCURRENT(n)}
        GEN(n) = GEN(n) $\cup$ GEN(m)
        KILL(n) = KILL(n) $\cup$ KILL(m)
     ENDDO
ENDDO

Execute LIVE on flow graph

DO for each node in flow graph
     $D'''_x$(-->n) = AVAIL(node)
ENDDO
```

```
DO for each node in the flow graph          ********************************
                                            *    Determine $F_x$(graph) and   *
                                            *$F'_x$(-->n), $F''_x$ (-->n), $F'''_x$(-->n)*
                                            ********************************
     IF node type is $\neg$ entry
          GEN(node) = $D_y$(node) $\cup$ $D_z$(node)
          KILL(node) = $F_x$(node)
          NULL(node) = {all variables}- (KILL(node) $\cup$ GEN(node))

     ELSE
          GEN(node) = {all variables}
          KILL(node) = { }
          NULL(node) = { }

     ENDIF

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 n in flow graph except entry
     DO for each node m $\varepsilon$ ALWAYS_CONCURRENT(n)
          GEN(n) = GEN(n) $\cup$ GEN(m)
          KILL(n) = KILL(n) $\cup$ KILL(m)

     ENDDO

ENDDO

Execute AVAIL on flow graph

DO for each node in flow graph
     $F''_x$(--> n) = {all variables}- AVAIL(node)
ENDDO

DO for each node n in flow graph except entry
     DO for each node m $\varepsilon$ {CONCURRENT(n)- ALWAYS_CONCURRENT(n)}
          GEN(n) = GEN(n) $\cup$ GEN(m)
          KILL(n) = KILL(n) $\cup$ KILL(m)

     ENDDO

ENDDO
```

```
        Execute AVAIL on flow graph
        DO for each node in flow graph
            $F'''_x$(-->n) = {all variables}- AVAIL(node)
        ENDDO
    ENDDO
    END
```

```
SEGMENT Number flow graph in postorder

**********************************************************************
*                                                                    *
*  This segment performs a depth first search on a flow graph and num-*
*  bers 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)
     ENDIF
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) = { }
ENDDO
change = true
DO while change is true
     change = false
     DO for j = 1 to n
          PREVIOUS = LIVE(j)
          LIVE(j) = { }
          DO for all K ε {successors of node j}
               LIVE(j) = LIVE(j) ∪ ((LIVE(K) ∩ {all variables}
                         - KILL(K))) ∪ GEN(K))
          ENDDO
          DO for all K ε {successors of node j which are heads of
                         precedence edges}
               DO for all v ε LIVE(j) ∩ $\overline{\text{LIVE}}$(K)
                    DO for all m ε {descendants of j | v ε GEN(m)}
                         IF ∃ path from K to m
                              LIVE(j) = LIVE(j) - {v}
                         ENDIF
                    ENDDO
               ENDDO
          ENDDO
          DO for all K ε {predecessors of node j which are tails of
                         precedence edges}
               DO for all v ε $\overline{\text{LIVE}}$(j) ∩ LIVE(K)
                    DO for all m ε {descendants of K | v ε GEN(m)}
                         IF ∃ path from j to m
                              LIVE(K) = LIVE(K) - {v}
                         ENDIF
                    ENDDO
               ENDDO
          ENDDO
```

```
            IF PREVIOUS ≠ LIVE(j)
                change = true
            ENDIF
        ENDDO
    ENDDO
    END
```

```
SEGMENT Execute AVAIL on graph

n = number of nodes in flow graph

    ******************************************************
    *                                                    *
    *  Assume graph is numbered from 1 to n in postorder. *
    *                                                    *
    ******************************************************

AVAIL(n) = { }
DO for j = n - 1 to 1
     AVAIL(j) = {all variables}
ENDDO
change = true
DO while change is true
     change = false
     DO for j = n - 1 to 1
          PREVIOUS = AVAIL(j)
          AVAIL(j) = {all variables}
          DO for all K ε {predecessors of node j which are not tails
                          of IPPE's}
               AVAIL(j) = AVAIL(j) ∩ ((AVAIL(K) ∩ ({all-variables}
                          - KILL(K))) ∪ GEN(K))
          ENDDO
          TEMP = {all variables}
          DO for all K ε {predecessors of node j which are tails of IPPE's}
               TEMP = TEMP ∩ AVAIL(K)
          ENDDO
          AVAIL(j) = AVAIL(j) ∪ TEMP
          IF PREVIOUS ≠ AVAIL(j)
               change = true
          ENDIF
     ENDDO
ENDDO

END
```

```
SEGMENT Report anomalies (flow graph)

    ***************************************************************
    *                                                             *
    * Flow graph may be for one code block or for entire program  *
    *                                                             *
    ***************************************************************

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

DO for each local variable in I (graph$_{block}$)
        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.")
        ENDIF
    ENDDO
ENDIF

IF flowgraph is for entire program
    Report invocation anomalies (block)
ENDIF

Determine anomalous paths of type ur, dd, du, r-d (flow graph)

END
```

```
SEGMENT Determine anomalous sequences of type ur, dd, du, r-d (flow graph)

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
        Report concurrent references and definitions (r-d) at (n)
        ANOM = $F_x(n) \cap C_y'(n\rightarrow)$
        IF ANOM $\neg$ 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 ...")
                ENDIF
            ENDDO
        ENDIF
        ANOM = $F_x(n) \cap C_y''(N\rightarrow) \cap \overline{C_y'}(n\rightarrow)$
        IF ANOM $\neg$ 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).  This anomaly involves nodes which
                    will always execute concurrently with nodes on
                    the path.  One such path is ...")
                ENDIF
            ENDDO
        ENDIF
```

```
ANOM = $F'''_y(\rightarrow n) \cap C_y(n) \cap \overline{F}'_y(\rightarrow n) \cap \overline{F}''_y(\rightarrow n)$
IF ANOM $\neg$ 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).  This anomaly involves nodes which
            may execute concurrently with nodes on the
            path.   One such path is ...")
        ENDIF
    ENDDO
ENDDO

ANOM = $D'_x(\rightarrow n) \cap A_y(n)$
IF ANOM $\neg$ empty
    DO for each variable in ANOM
        IF variable is simple variable or FORM IS $\neg$ dd
            Find a path that contains anomaly (entering
            xy, all, node) for (variable)
            Output message ("On all paths entering (node),
            anomaly of type (FORM) occurs for (variable).
            One such path is ...")
        ENDIF
    ENDDO
ENDIF

ANOM = $D''_x(\rightarrow n) \cap A_y(n) \cap \overline{D}'_x(\rightarrow n)$
IF ANOM $\neg$ empty
    DO for each variable in ANOM
        IF variable is simple variable or FORM IS $\neg$ dd
            Find a path that contains anomaly (entering,
            xy, all, node) for (variable)
            Output message ("On all paths entering (node),
            anomaly of type (FORM) occurs for (variable).
            This anomaly involves nodes which will always
            execute concurrently with nodes on the paths.
            One such path is ...")
        ENDIF
    ENDDO
ENDIF
```

```
ANOM = $D_x(n) \cap A'''_y(n\rightarrow) \cap \overline{A}'_y(n\rightarrow) \cap \overline{A}'_y(n\rightarrow)$
IF ANOM $\neg$ empty
    DO for each variable in ANOM
        IF variable is simple variable or FORM IS $\neg$ 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).
            This anomaly involves nodes which may execute
            concurrently with nodes on the paths.  One
            such path is ...")
        ENDIF
    ENDDO
ENDIF

ANOM = $F'_x(\rightarrow n) \cap C_y(n)$
IF ANOM $\neg$ 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 enter-
            ing (node), anomaly of type (FORM) occurs for
            (variable).  One such path is ...")
        ENDIF
    ENDDO
ENDIF

ANOM = $F''_x(\rightarrow n) \cap C_y(n) \cap \overline{F}'_x(\rightarrow n)$
IF ANOM $\neg$ 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).  This anomaly involves nodes which
            will always execute concurrently with nodes on
            the path.   One such path is ...")
        ENDIF
    ENDDO
ENDIF
```

```
ANOM = $F_x(n) \cap C_y'''(n\rightarrow) \cap \overline{C}_y'(n\rightarrow) \cap \overline{C}_y''(n\rightarrow)$
IF ANOM $\neg$ 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). This anomaly involves nodes
            which may execute concurrently with nodes on
            the path. One such path is ...")
        ENDIF
    ENDDO
ENDIF

ANOM = $D_x(n) \cap A_y'(n\rightarrow)$
IF ANOM $\neg$ empty
    DO for each variable in ANOM
        IF variable is simple variable or FORM is $\neg$ 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 ...")
        ENDIF
    ENDDO
ENDIF

ANOM = $D_x(n) \cap A_y''(n\rightarrow) \cap \overline{A}_y'(n\rightarrow)$
IF ANOM $\neg$ empty
    DO for each variable in ANOM
        IF variable is simple variable or FORM IS $\neg$ 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).
            This anomaly invovles nodes which will always
            execute concurrently with nodes on the paths.
            One such path is ...")
        ENDIF
    ENDDO
ENDIF
```

```
        ANOM = $D'''_x(\rightarrow n) \cap A_y(n) \cap \overline{D}'_x(\rightarrow n) \cap \overline{D}''_x(\rightarrow n)$
        IF ANOM $\neg$ empty
             DO for each variable in ANOM
                  IF variable is simple variable or FORM IS $\neg$ dd
                       Find a path that contains anomaly (entering,
                       xy, all, node) for (variable)
                       Output message ("On all paths entering (node),
                       anomaly of type (FORM) occurs for (variable).
                       This anomaly involves nodes which may execute
                       concurrently with nodes on the path.  One such
                       path is ...")
                  ENDIF
             ENDDO
        ENDIF
   ENDDO
```

SEGMENT Report concurrent references and definitions (r-d) at (n)

```
REFN = $A_r(n) \cup D_r(n)$
DEFN = $A_d(n) \cup D_d(n)$
DO  For SET = CONCURRENT(n), ALWAYS_CONCURRENT(n), POSSIBLY_CONCURRENT(n)
    DO  for all m $\varepsilon$ SET
        REFM = $A_r(m) \cup D_r(m)$
        DEFM = $A_d(m) \cup D_d(m)$

        ANOM = REFN $\cap$ DEFM
        DO for all variables $\varepsilon$ ANOM
            Output message ("(Variable) is referenced at node(n) and
            defined at node (m).  Node (m) is a member of (SET(n)).")
        ENDDO
        ANOM = DEFN $\cap$ REFM
        DO for all variables $\varepsilon$ ANOM
            Output message ("(Variable) is referenced at node (m)
            and defined at node (n).  Node (m) is a member of
            (SET(n)).")
        ENDDO
    ENDDO
ENDDO
END
```

<u>SEGMENT</u> 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 pro- *
*  gress to find more efficient algorithms for localizing anoma-    *
*  lous path expressions.                                           *
*                                                                   *
*********************************************************************
```

<u>END</u>

```
SEGMENT Report invocation anomalies (unit flow graph)

DO for each subprogram invocation in unit
     DO for each argument in invocation
          Report non-usage of (argument) corresponding
          to parameter
          Report side effects involving (argument)
     ENDDO
ENDDO
END
```

APPENDIX F

Pseudo-code for Synchronization Anomaly Detection

The pseudo-code in this appendix uses the syntax defined in Caine and Gordon, "PDL - A tool for software Design," Proc, 1975 National Computer Conference, June 1975, pp. 271-276.

Tree of Inter-Segment References

Page	Segment
107	find potentially infinite waits
109	find instances of p process being scheduled while still running
110	find potential premature terminations

```
SEGMENT find potentially infinite waits
DO   for each wait node w in the graph
     POSSIBLY_INFINITE = FALSE
     DO   for each conjunct c in w
          CONJUNCT_POSSIBLY_INFINITE = TRUE
          DO   for each term t in c
               TERM_POSSIBLY_INFINITE = TRUE
               DO   for each node $n_t$ making t true belonging to
                    ALWAYS (w) minus POSSIBLY_AFTER (w)
                    NODE_POSSIBLY_NEGATED = FALSE
                    DO   for each node $n_f$ making t false belonging to
                         {all nodes minus {BEFORE($n_t$) union NEVER($n_t$)}}
                         intersection {all nodes minus{NEVER(w) union
                         AFTER (w)}}
                         IF   there does not exist a node $n_t$ making t
                              true belonging to {ALWAYS_AFTER(w)}
                              intersection {all nodes minus {NEVER (w)
                              union POSSIBLY_AFTER (w)}}
                              NODE_POSSIBLY NEGATED = TRUE
                         ENDIF
                    ENDDO
                    IF   NODE_POSSIBLY_NEGATED = FALSE
                         TERM_POSSIBLY_INFINITE = FALSE
                    ENDIF
               ENDDO
               IF   TERM_POSSIBLY_INFINITE = FALSE
                    CONJUNCT_POSSIBLY_INFINITE = FALSE
               ENDIF
          ENDDO
          IF   CONJUNCT_POSSIBLY_INFINITE = TRUE
               POSSIBLY_INFINITE = TRUE
          ENDIF
     ENDDO
```

```
    IF   POSSIBLY_INFINITE = TRUE
         WRITE error message
    ENDIF
ENDDO
END
```

```
SEGMENT find instances of p process being scheduled while still running
DO   for each SCHEDULE (p) in the flow graph
     IF   there exists a different SCHEDULE (p) or CLOSE (p) belonging
          to {all nodes minus {BEFORE union AFTER union NEVER for
          this SCHEDULE (p)}}
          WRITE error message
     ENDIF
     IF   there exists a different SCHEDULE (p) belonging to
          BEFORE (this SCHEDULE(p))
     AND  the corresponding CLOSE (p) does not belong to
          BEFORE (this SCHEDULE(p))
          WRITE error message
     ENDIF
ENDDO
END
```

```
SEGMENT find potential premature terminations
DO   for each TERMINATE statement
     IF   TERMINATE statement is followed by process name list
          DO   for each process p on process name list
               IF   the TERMINATE does not belong to AFTER (CLOSE node of p)
                    WRITE error message
                    DO   for each process q dependent on p
                         IF   the terminate does not belong to AFTER
                              (CLOSE node of q)
                              WRITE error message
                         ENDIF
                    ENDDO
               ENDIF
          ENDDO
     ELSE DO   for each process p dependent on process containing
               TERMINATE statement
               IF   the TERMINATE does not belong to AFTER (CLOSE node of P)
                    WRITE error message
               ENDIF
          ENDDO
     ENDIF
ENDDO
END
```