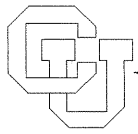
Anomaly Detection in Concurrent Programs

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

CU-CS-147-79 January 1979



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This research was supported by grant #NSG-1476 from NASA Langley Research Center.

ANOMALY DETECTION IN CONCURRENT PROGRAMS†

by

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

Computer Science Department
University of Colorado at Boulder

CU-CS-147-79 January, 1979

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

Abstract

An approach to the analysis of concurrent 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 overviews of 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.

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 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.

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 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.

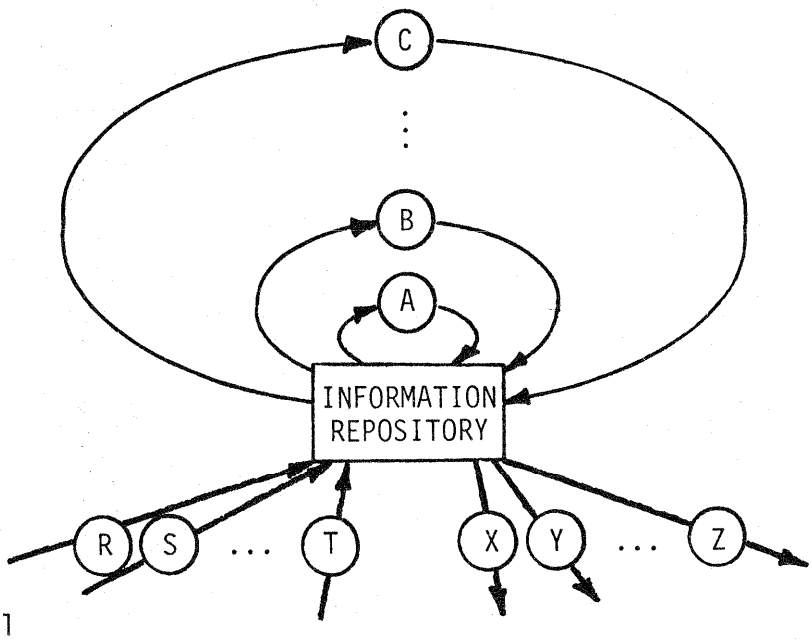


Figure 1
Organization of Development Support System

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 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.

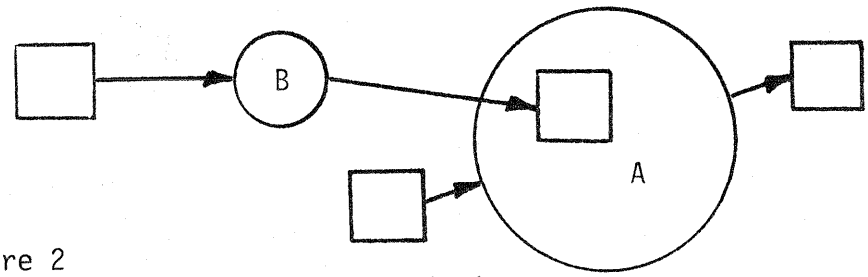


Figure 2
Representation of a Module Producing Information Used in Another Module's Processing

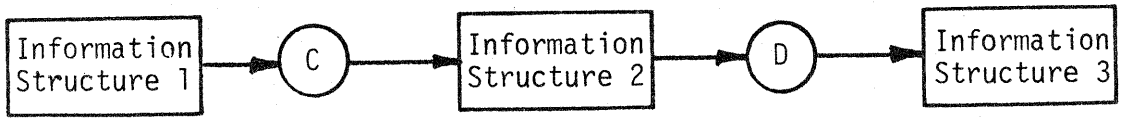


Figure 3
Representation of a Module Producing Information Used as Input by Another Module

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

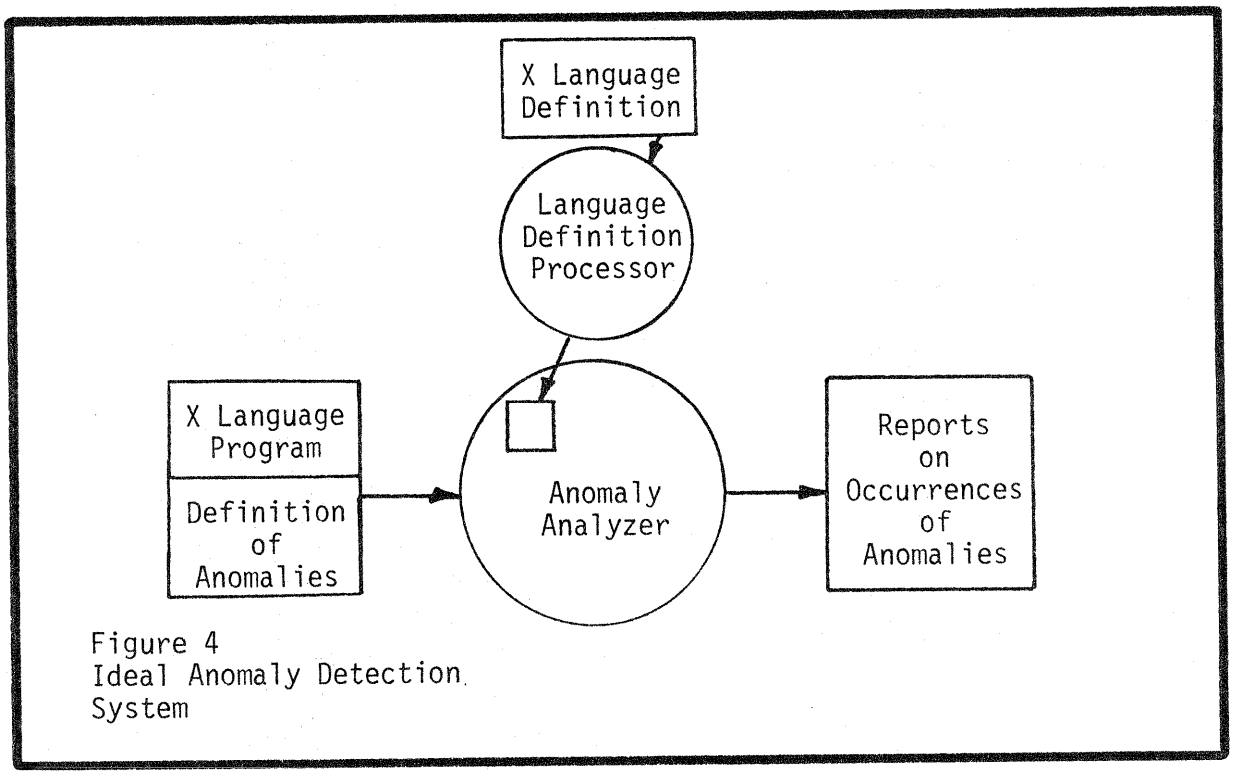


Figure 4
Ideal Anomaly Detection System

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 [Inte 76], but for the purposes of this paper we will consider a general language, which we call X, having concurrent programming facilities. As will be indicated later, our techniques do not depend on the exact form of the language's constructs. Thus we do not give an explicit description of the X language but rather uncover its capabilities gradually throughout the exposition.)

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 X Language Processor module. Second, the overall system may be easily modified to use representations of X programs other than the program text. In particular, the overall

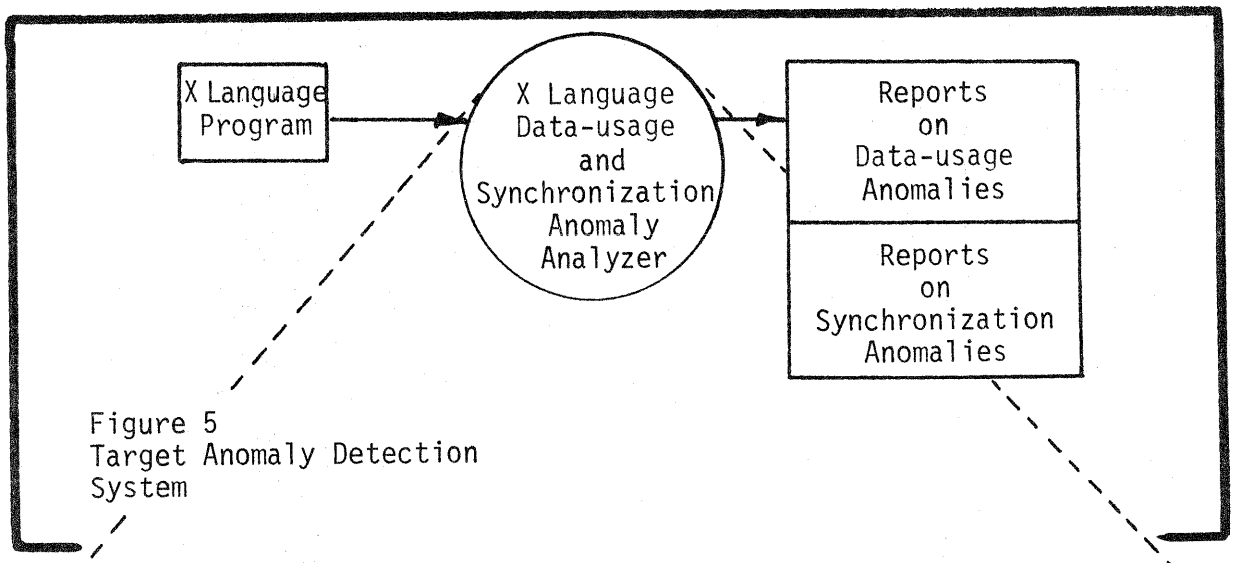


Figure 5
Target Anomaly Detection System

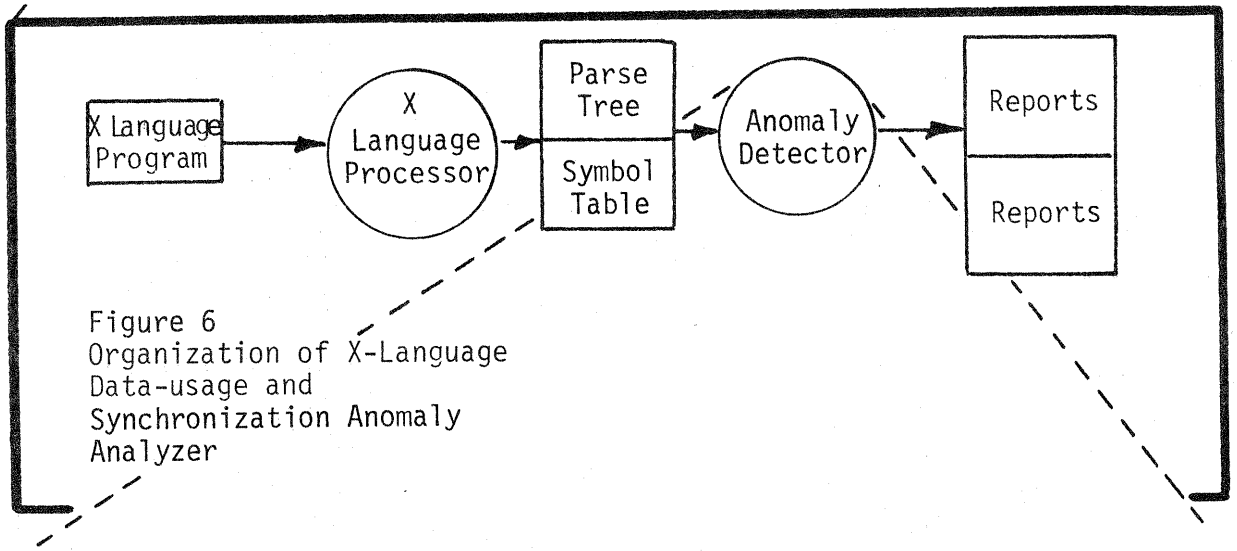


Figure 6
Organization of X-Language Data-usage and Synchronization Anomaly Analyzer

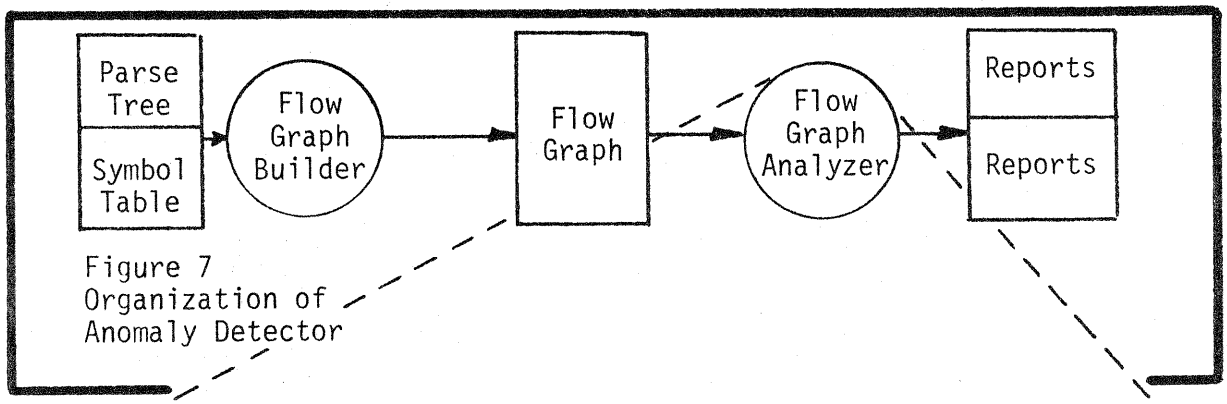


Figure 7
Organization of
Anomaly Detector

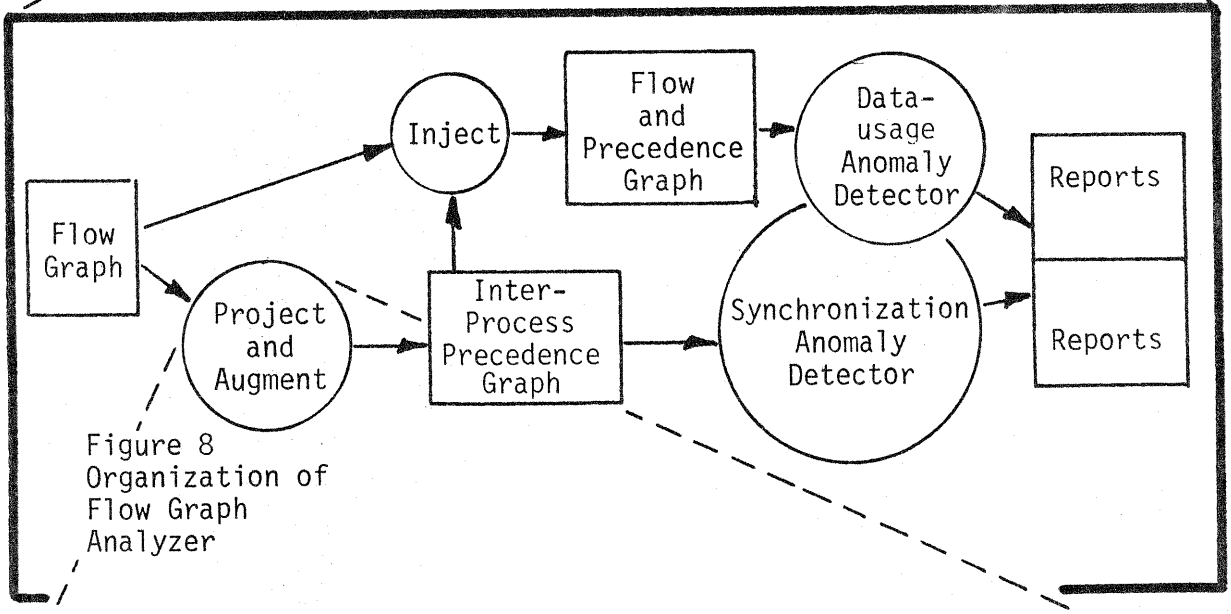


Figure 8
Organization of
Flow Graph
Analyzer

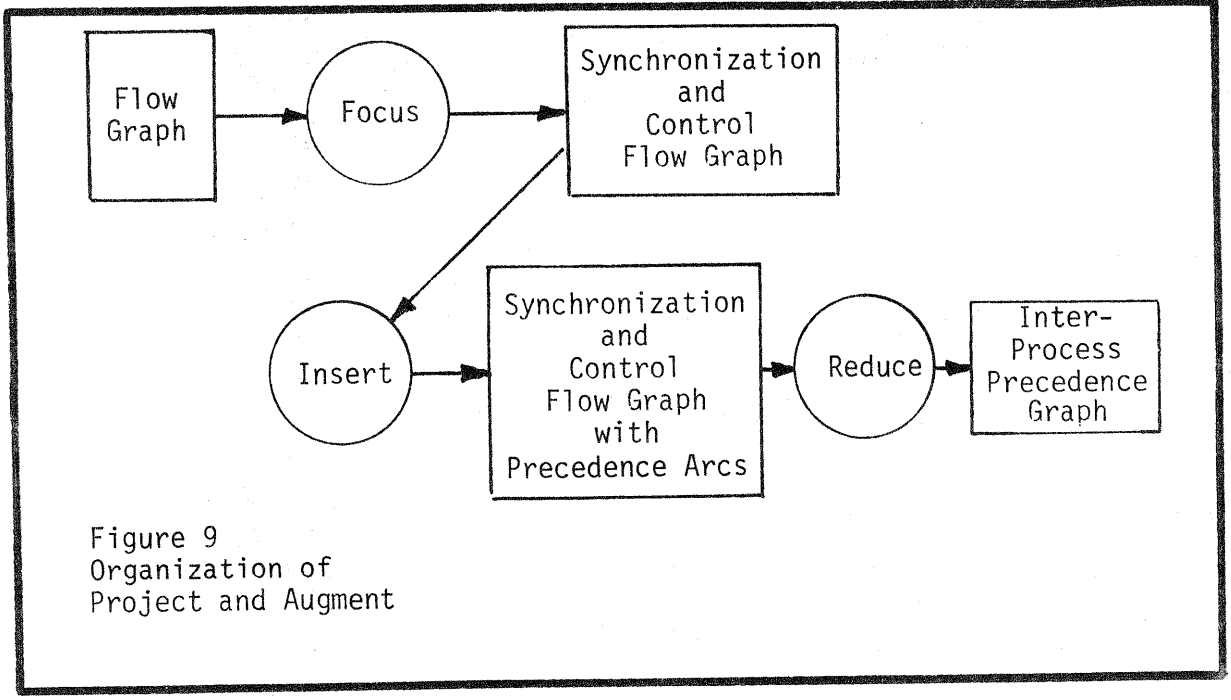


Figure 9
Organization of
Project and Augment

system may be changed to use the representation produced by the X 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 programs and tasks). 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.

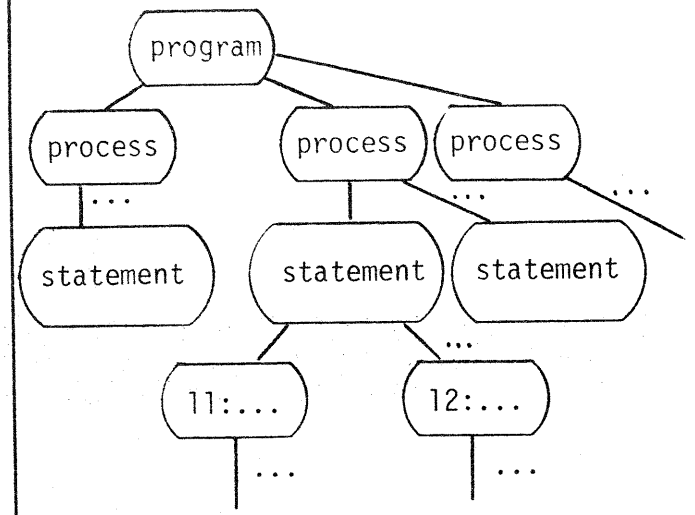
III. An Example

In Figure 10 we present a hypothetical program in the X 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;
```

X Language Program

X Language Processor



Parse Tree

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

Symbol Table

continued on next page

Figure 10
Example Showing Program Representations

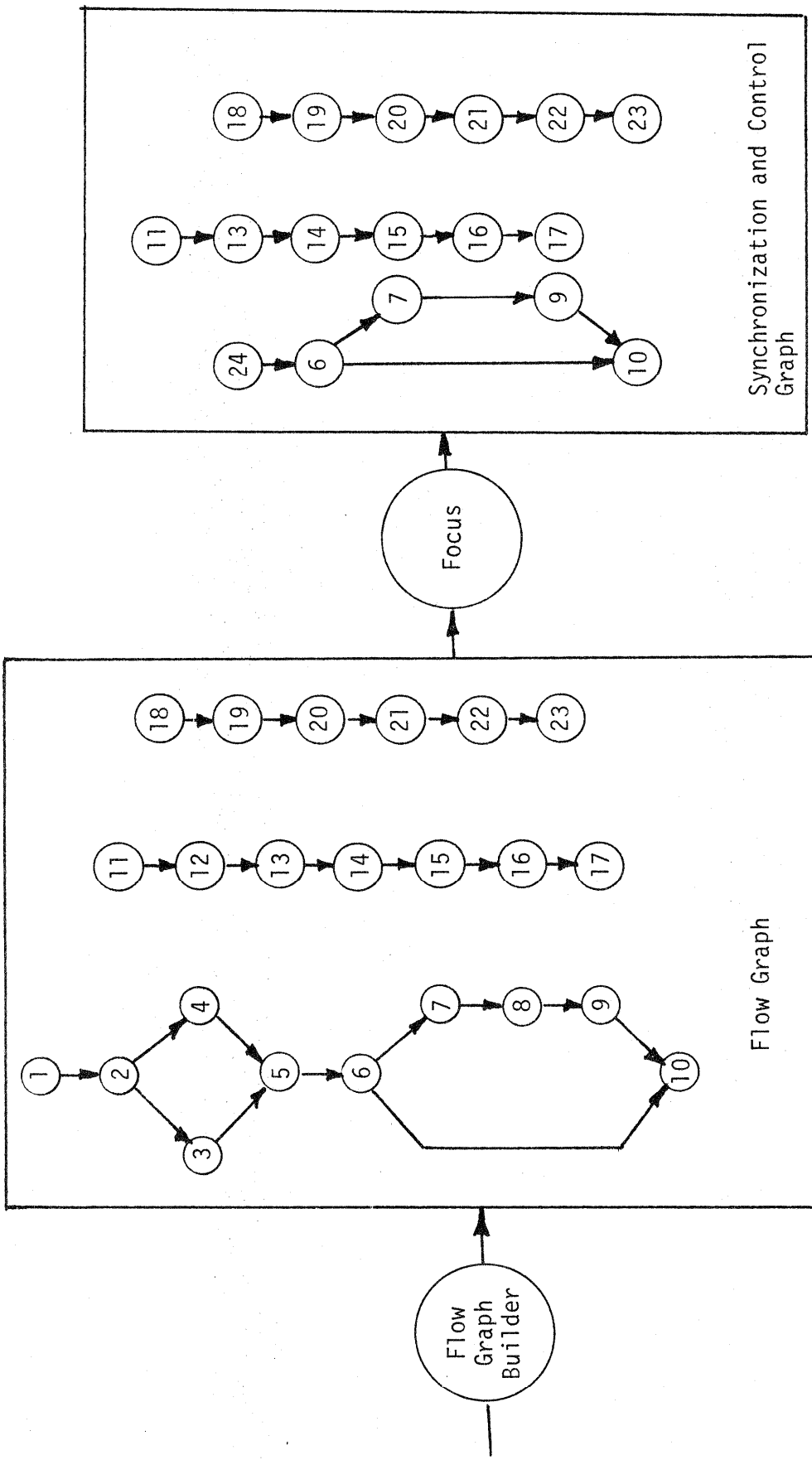


Figure 10
... continued

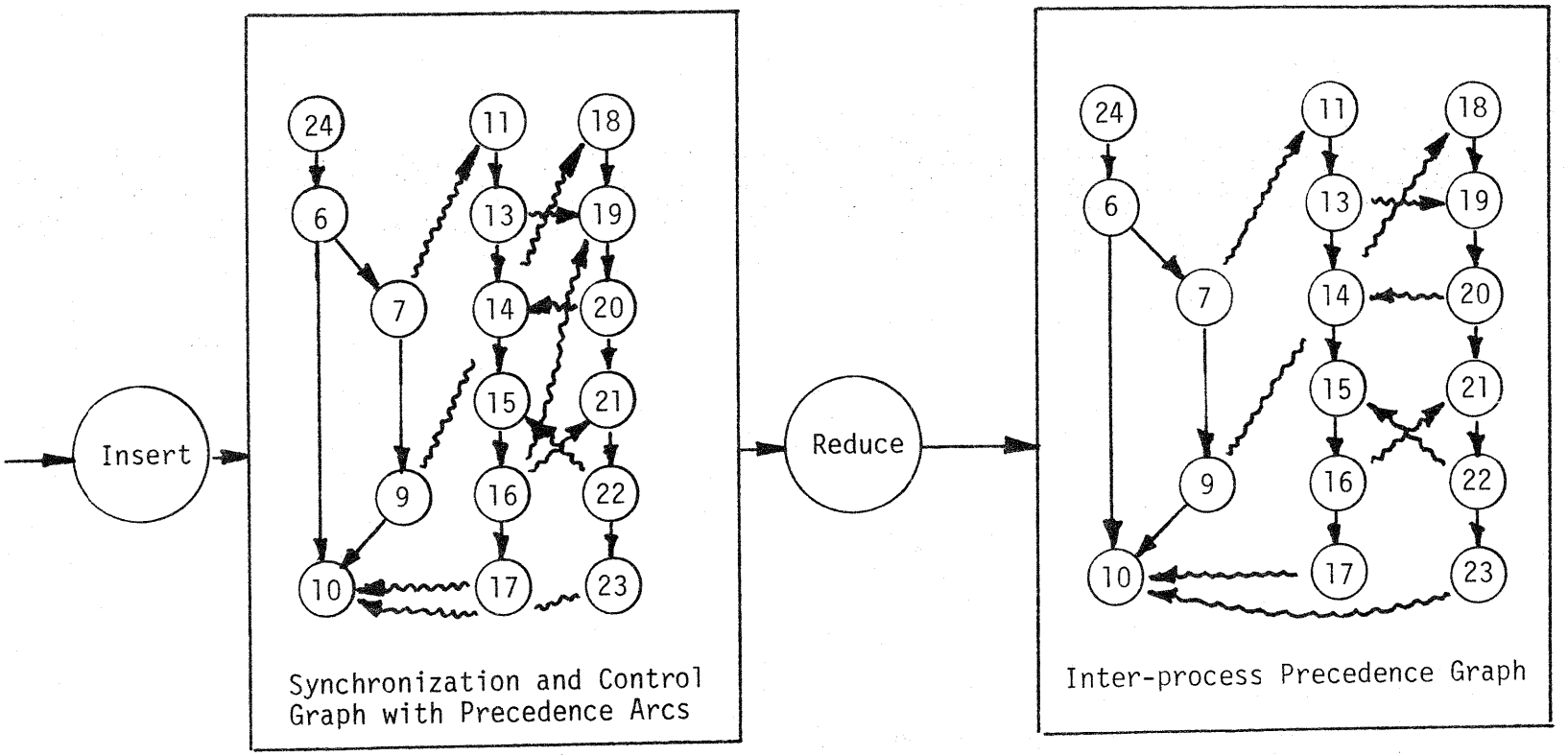


Figure 10
... continued

continued on next page

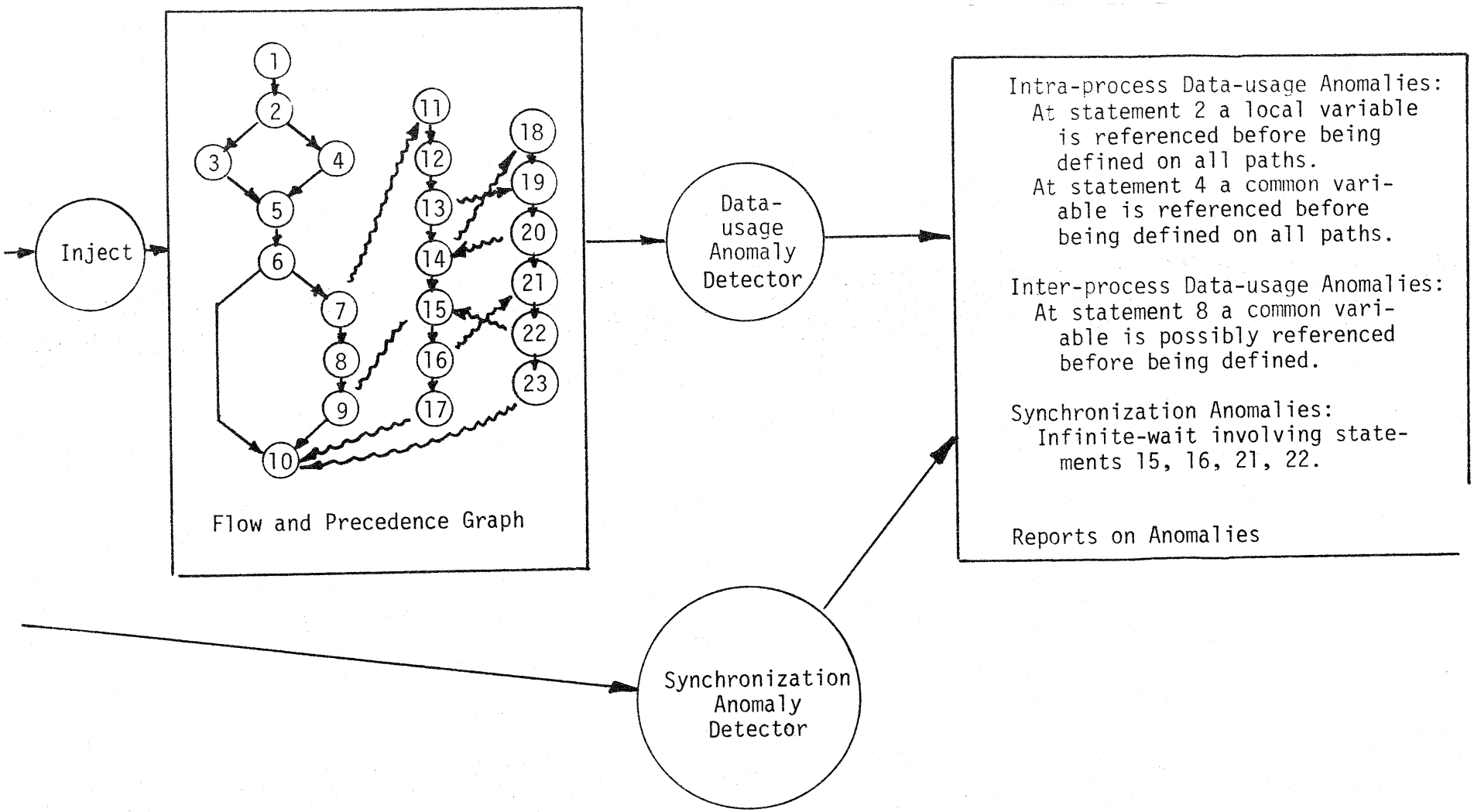


Figure 10
... continued

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 X language.

V. Graph Building

The first major task of the anomaly detection system is to derive an abstraction of the program being analyzed. This abstraction must contain all information which is pertinent to the anomalies under detection and must be in a form which is conveniently used by the anomaly detection algorithms.

V.1 Flow Graph

The flow graph is derived from the parse tree and symbol table for a program specified in the X 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, $n_e \in N$.
4. n_x, the unique exit node, $n_x \in N$.

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. Each node, $n_j \in N$, has the following information associated with it:

1. P, the set of predecessor nodes. $n_i \in P$ if the edge $(n_i, n_j) \in E$.
2. S, the set of successor nodes. $n_i \in S$ if the edge $(n_j, n_i) \in E$.
3. t, the type of the node, indicating the type of statement in the language X 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 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

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

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.

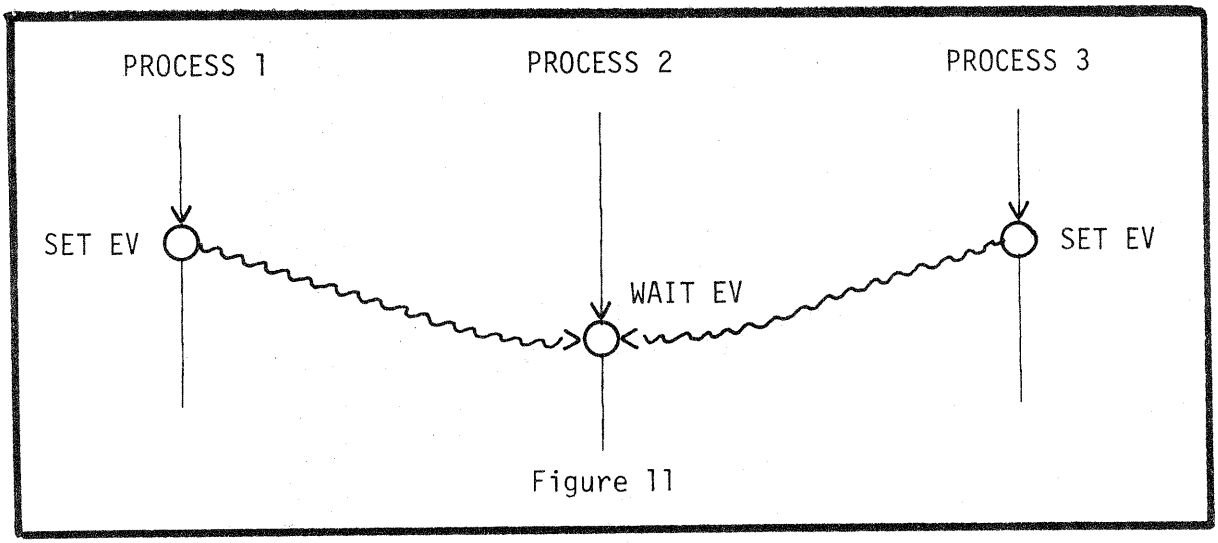


Figure 11

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.

In the X 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.3 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.

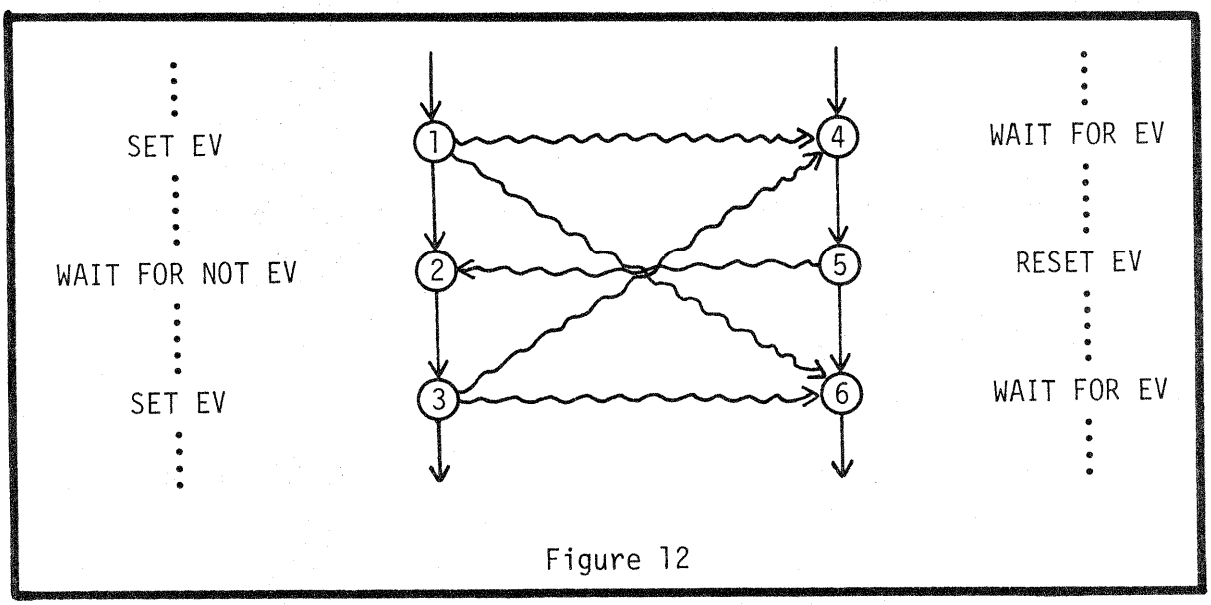


Figure 12

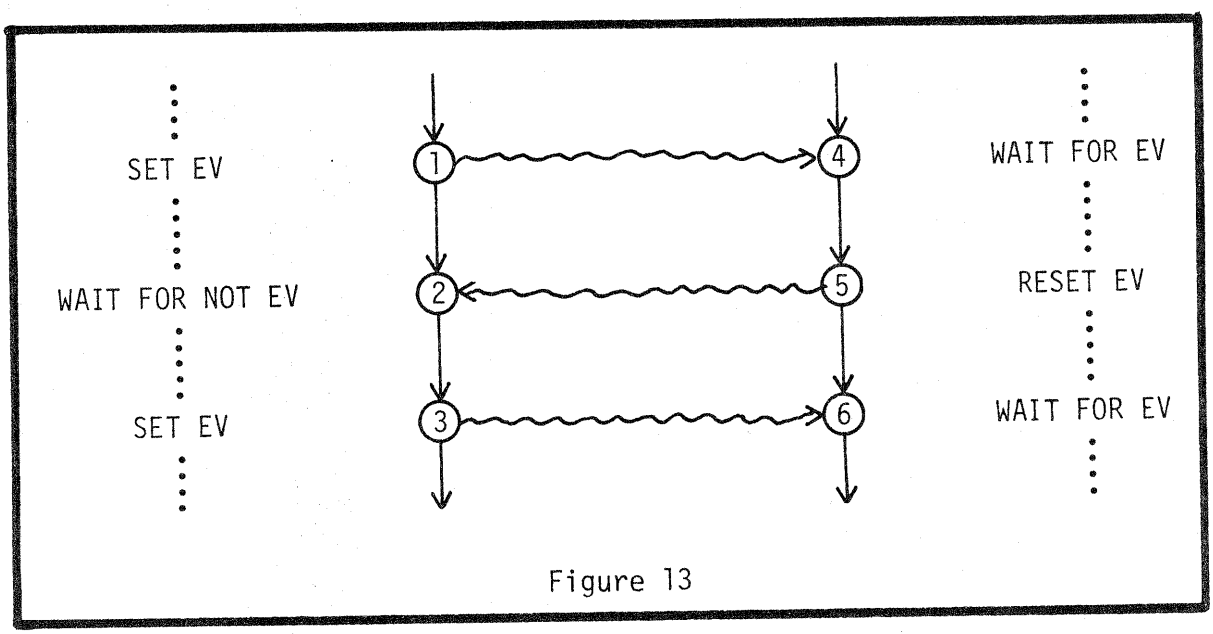


Figure 13

Spurious IPPEs can be removed by using two algorithms, BEFORE and AFTER. These recursive algorithms determine the sets of nodes whose execution will occur before or after each node n in the graph. The BEFORE(n) set is calculated so that it contains all those nodes which, if they are executed at all, are executed before node n. To do this, the BEFORE set of the strongly connected component containing n is first determined and added to BEFORE(n). Then, nodes in the intersection of the BEFORE sets of those nodes with edges into n and lying on elementary paths from an entry node to n's component are added. Finally, members of the intersection of the BEFORE sets of nodes which are tails of IPPEs into n are included also. The AFTER(n) set likewise contains all nodes which, if executed at all, are executed after node n; it is computed in a manner similar to BEFORE.

If, for any IPPE, the predecessor node is in the AFTER set of the successor node, or the successor node is in the BEFORE set of the predecessor node, condition 2) above is violated, and the IPPE is removed. If, for any IPPE, a node negating the effect of the predecessor node occurs in the AFTER set of the predecessor node and the BEFORE set of the successor node, the IPPE violates condition 3) above and can be removed. The removal of an IPPE may alter the generated BEFORE and AFTER 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 and AFTER sets will be subsets of the actual BEFORE and 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 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 will terminate at this point.

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 necessary 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 others, 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 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 side effect data flow of global variables as a result of the subprogram's invocation.

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 graph itself to find these sets of concurrent nodes. We first determine the sets of nodes whose execution will occur before or after each node n using the BEFORE and AFTER algorithms described in section V.3. Then, for node n in process p, CONCURRENT(n) contains those nodes m, not in p, such that m $\notin$ BEFORE(n) and m $\notin$ AFTER(n).

By using the knowledge of the dominators of a node, these sets can be further subdivided into BEFORE, ALWAYS_BEFORE, AFTER, ALWAYS_AFTER, CONCURRENT, and ALWAYS_CONCURRENT, heretofore collectively referred to as the execution sequence sets. ALWAYS_BEFORE(n) is the subset of BEFORE(n) which contains those nodes m' such that all paths from the start of the process to n include node m'; whereas m ε BEFORE(n) indicates that m lies on at least one path from the start of the process to n. Similarly for AFTER(n) and ALWAYS_AFTER(n). The execution of a member of ALWAYS_CONCURRENT(n) may occur either before, after, or in parallel with n. This possibility exists for all execution sequences which include n. However, for a member of CONCURRENT(n), the potential for concurrency of execution exists for at least one sequence which includes n, but not necessarily for all sequences.

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 *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 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.

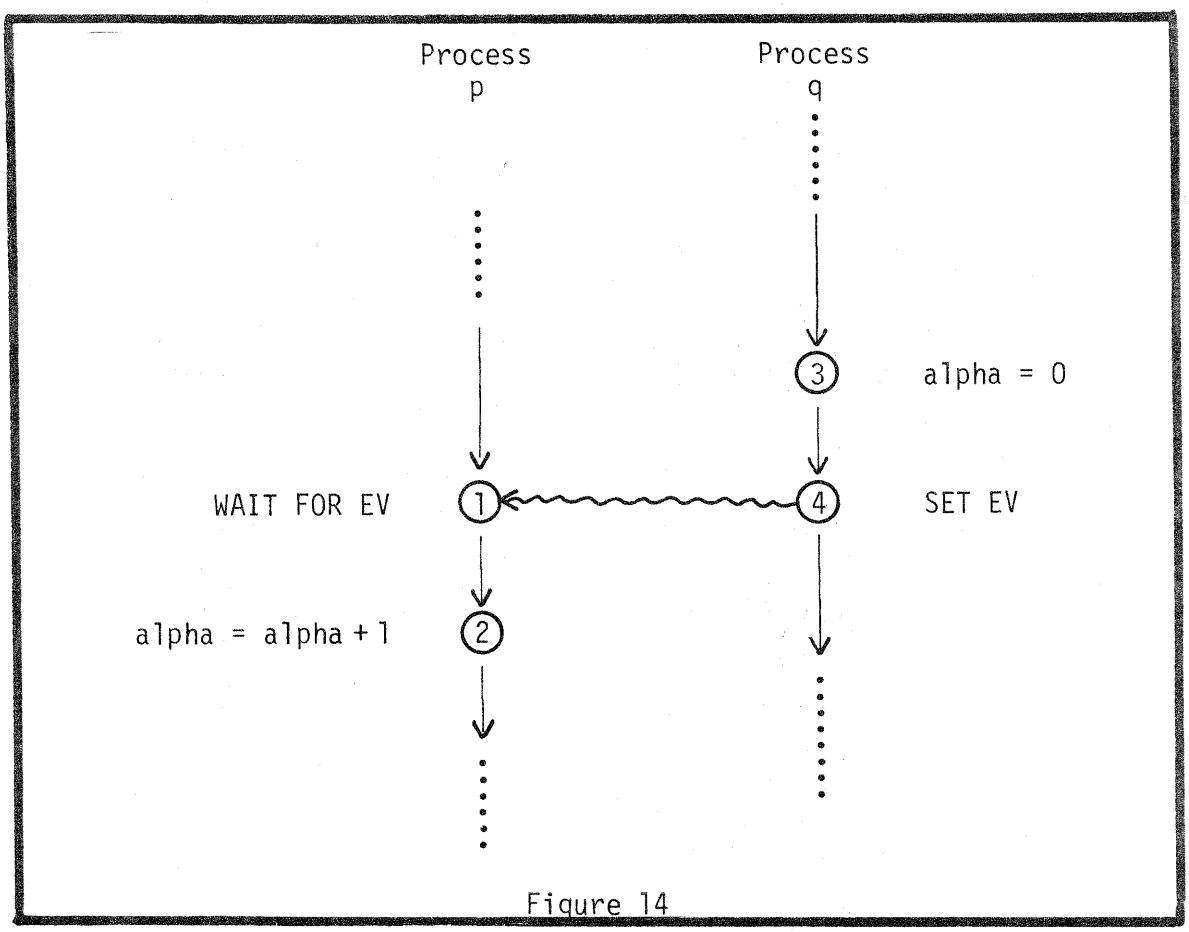


Figure 14

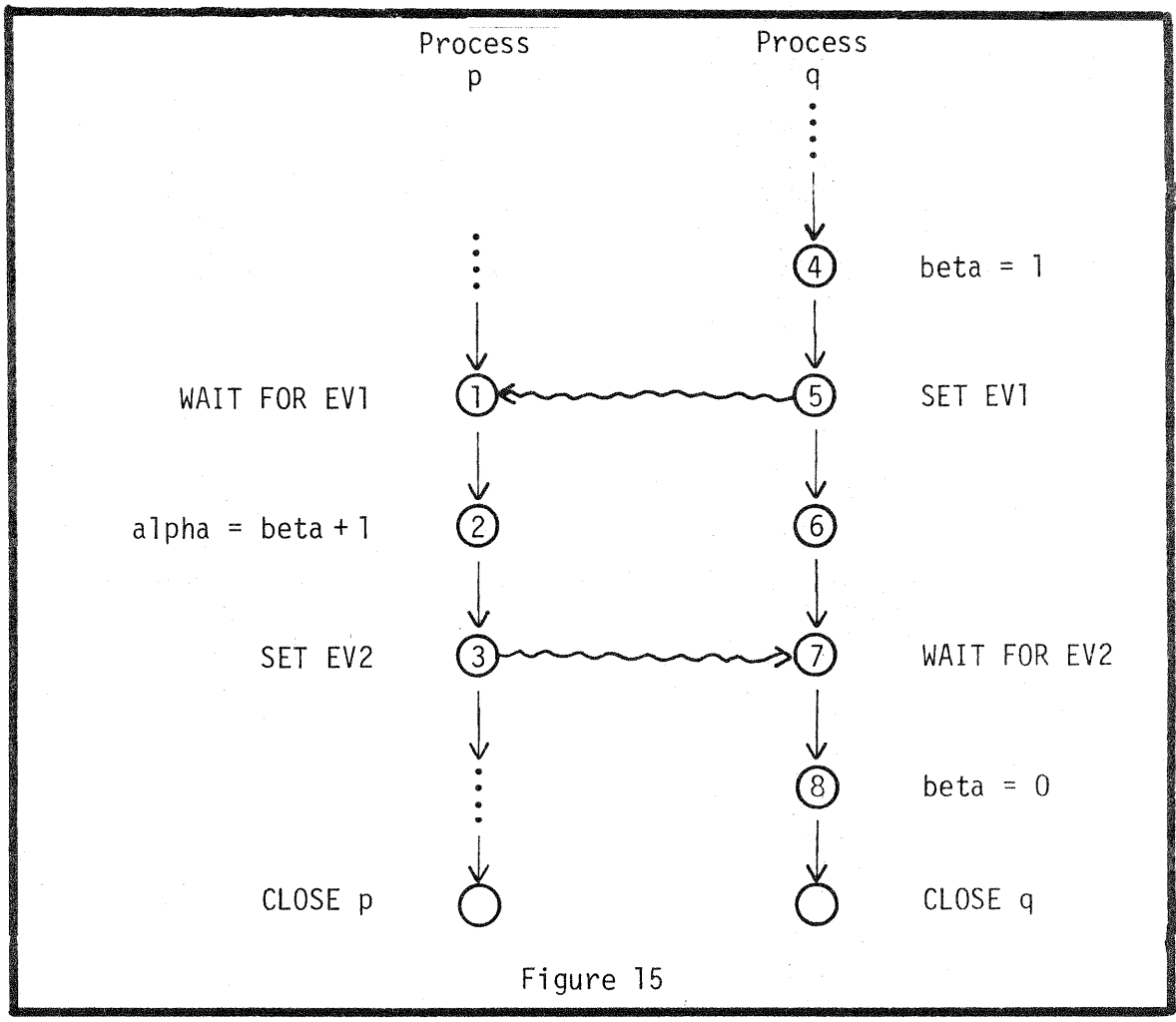


Figure 15

The three types of anomalies detected in single-process programs 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. Here, *alpha* may or may not be defined when node 4 is executed since node 7 ε CONCURRENT(4). This is a possible case of anomaly type (1), a reference to an uninitialized variable. An example of anomaly type (2), 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 9: 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.

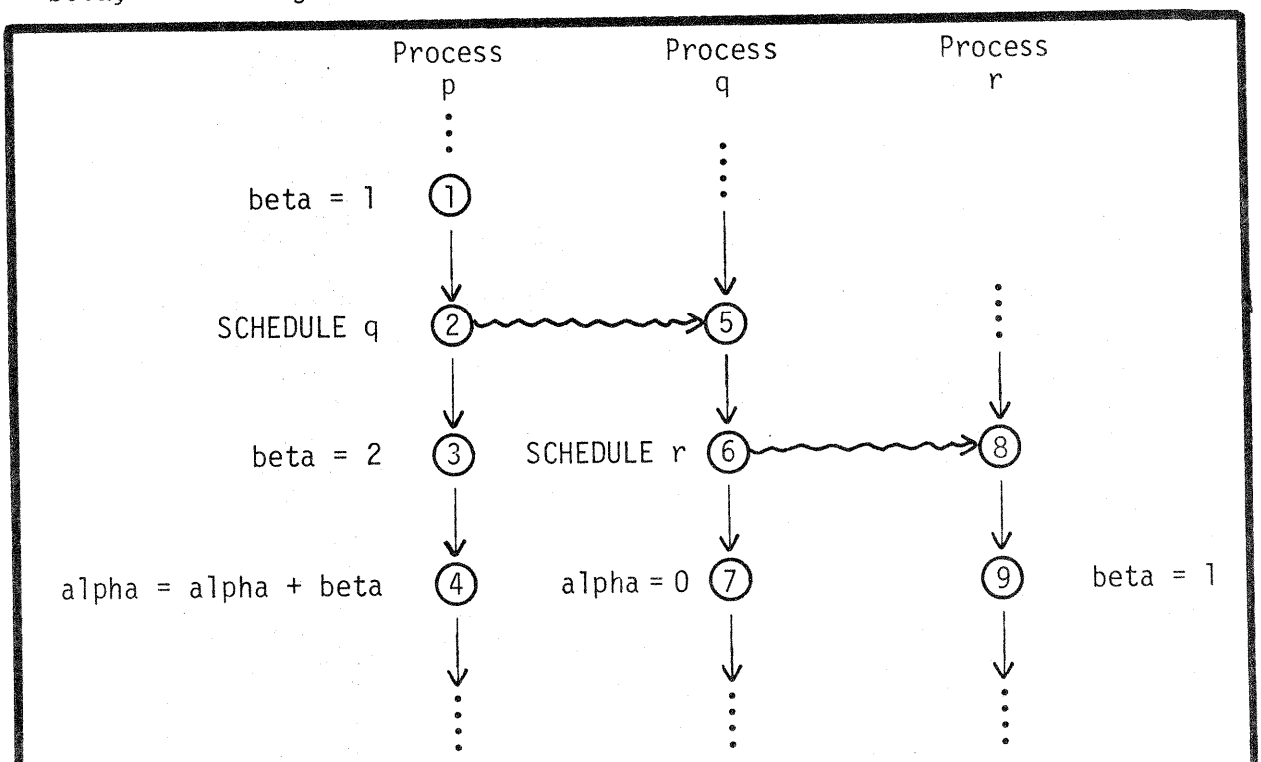


Figure 16

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; i.e., those nodes setting the term to true from the ALWAYS_BEFORE, ALWAYS_CONCURRENT and ALWAYS_AFTER sets for the WAIT node. Further, it is assumed that all nodes from the concurrent sets will, in fact, execute before the WAIT is reached. Given these assumptions, and considering only those nodes which now will execute before the WAIT is reached, the term will be indefinitely false if each node setting it to true can be followed by another setting it back to false. 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 note n_T setting t to true,

$$n_T \in \{\text{ALWAYS\_BEFORE}(w) \cup \text{ALWAYS\_CONCURRENT}(w)\}$$

there exists a node n_F setting t to false,

$$n_F \in \{\text{CONCURRENT}(n_T) \cup \text{AFTER}(n_T)\}$$
$$\cap \{\text{BEFORE}(w) \cup \text{CONCURRENT}(w)\}$$

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

Two other types of anomalies proved readily detectable from the execution sequence sets. The first of these is the possibility that a process can be rescheduled while it is still running (this is a violation of the rules of HAL/S). Checking for this requires examining the execution sequence sets at each schedule or close node. If at either of these nodes, a different schedule or close on the same process appears in the CONCURRENT set, then there is a potential anomaly. Further, if a different schedule on the same process appears in the BEFORE set at a schedule node, and the corresponding close is not also in the BEFORE set, this also signifies a potential anomaly.

The second type of anomaly is the possibility of the premature termination of a process. Although this is generally not a language definition violation, 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.

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 are 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.

We are currently investigating the needs of system developers to determine additional assertions that would be useful.

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.