ANOMALY DETECTION IN CONCURRENT SOFTWARE BY STATIC DATA FLOW ANALYSIS

Richard N. Taylor[1]

Leon J. Osterweil[2]

CU-CS-152-79 April, 1979

ANOMALY DETECTION IN CONCURRENT SOFTWARE BY STATIC DATA FLOW ANALYSIS

Richard N. Taylor[1]

Leon J. Osterweil[2]

CU-CS-152-79

April, 1979

ABSTRACT

Algorithms are presented for detecting errors and anomalies in programs which use synchronization constructs to implement concurrency. The algorithms employ data flow analysis techniques. First used in compiler object code optimization, the techniques have more recently been used in the detection of variable usage errors in single process programs. By adapting these existing algorithms the same classes of variable usage errors can be detected in concurrent process programs. Important classes of errors unique to concurrent process programs are also described, and algorithms for their detection are presented.

1.0 INTRODUCTION

Data flow analysis has been shown to be a useful tool in demonstrating the presence or absence of certain significant classes of programming errors.[1] It is an important software verification technique, as it is inexpensive and dependably detects a well defined and useful class of anomalies. Work to this point has been directed at the analysis of single process programs. Data flow analysis of concurrent programs has not been investigated. Concurrency causes difficulty in the detection of most errors which occur in single process programs; it also creates the possibility of new classes of errors.

One of the simplest errors which can occur in both categories of programs is referencing an undefined variable. (We recognize that this error can be eliminated by requiring that all variables be declared and given initial values at the point of declaration. Most programming languages do not have this requirement, however. Moreover, the presentation and discussion of this error is useful for pedagogical purposes.) Another programming anomaly which may occur in both categories is a dead variable definition. This occurs when a variable is defined twice without an intervening reference, or if a variable is defined yet never subsequently referenced. In concurrent software these types of anomalies and errors can occur in more subtle ways than in single process programs. For example, within a system of concurrent processes, one process may reference a shared variable while a parallel process may be redefining it. It is clearly desirable that such errors and anomalies be analytically detected or shown to be absent from programs.

In this paper, we show that data flow analysis can reliably demonstrate the presence or absence of these and other programming anomalies for both single process and concurrent programs. While the anomalies are of interest in

themselves, they are particularly important because experience has shown that consideration of why they arose in the program's construction often leads to the detection of significant design errors.

2.0 EXAMPLE AND BASIC DEFINITIONS

2.1 Programming Language Description

In order to clarify the types of errors we are addressing, several examples are needed. We are interested in designing analytic techniques which may be applied to a variety of languages supporting concurrent programming, such as Concurrent Pascal[2], Modula[3], and JOVIAL. The languages which are currently used for real-time, concurrent process programming display a variety of techniques to allow synchronization and communication. Some are more error-resistant than others (to say the least). Still more constructs and techniques are being proposed. For example, Ada[4], the proposed new common higher order language for embedded applications developed for the Department of Defense, displays a number of new techniques. We have attempted to avoid language and methodology dependence in the development of our analytic techniques, so that they will not readily be outdated. We have assumed only the existence of a few constructs common to nearly all contemporary concurrent languages, because tools are needed now for the languages which are already in use. But it appears likely to us that the techniques designed in creating these tools will not be obsoleted by new language designs or concurrency constructs.

The programming language which forms the basis for this presentation is derived from HAL/S, an algorithmic language designed for the production of real-time flight software[5]. HAL/S was developed for use on the Space Shuttle and is employed elsewhere within NASA for a variety of tasks [6]. We have extracted a simple yet powerful subset of this language and altered slightly the syntax and

semantics of several of its constructs. HAL/S bears many similarities to Algol 60 and PL/I. Hence the ysntax and semantics of these languages can generally be safely used in understanding the examples in this paper. Of particular interest, however are the following language constructs with which we will be primarily concerned in describing our examples and in designing our analysis.

1. Assignment statement. This statement is of the form

variable = expression ;

In executing this statement, the expression is evaluated and the result is then assigned to the variable.

2. Process declaration statements (**program**, **task**, and **close**). The declaration of each process begins with a declaration statement. The main program begins with a **program** declaration statement. Other processes begin with a **task** declaration statement. The end of a process declaration is marked with a **close** declaration statement.

3. Schedule statement. The execution of any process except for the main program is enabled through execution of a **schedule** statement. Execution of a **schedule** does not guarantee that the specified process will begin immediately, it merely indicates that the process is ready for execution. The actual time of initiation of a process is determined by the system scheduler. Any number of processes may be enabled for concurrent execution, but a process may not be scheduled to execute in parallel with itself. The schedule statement explicitly names the process or processes to be started; run-time determination of processes to be shceduled is not allowed.

4. Wait statement. This statement causes the executing process to wait for another process (or processes) to terminate before continuing with its own execution. A process has terminated when it has completed its

execution and no longer resides in the system scheduler's "ready" queue. As with the **schedule** statement, the process(es) waited for is (are) named explicitly in the declaration; run-time determination is not allowed. The statement may be formulated two ways:

wait for process_name$_1$ **and** process_name$_2$ ...

or **wait for** process_name$_1$ **or** process_name$_2$ ...

When the process names are joined through logical disjunction, the wait is interpreted as **wait-for-any**. As soon as one of the named processes has terminated, the waiting process may proceed. When the process names are joined by logical conjunction all of the named processes must terminate before the waiting process may proceed. We shall refer to this as a **wait-for-all** statement.

5. Shared variables. Program variables have associated with them Algol-like scoping rules. This scoping exists at the program level, meaning that two processes may both access the same variable. We assume that no protection mechanism exists.

6. Transput. Input to a program is accomplished through a **read** statement. Values are output via a **write** statement.

2.2 Example

Using the above constructs we now present an example program (Figure 2.2-1) which contains several anomalies.

```
1     Main: program;

2           declare integer x,y;
            /* x,y are global variables known throughout the main program and all
            tasks */
3           declare boolean flag ;

4           T1: task;
5                 write x ;
6                 wait for T3 ;
7           close T1 ;

8           T2: task;
9                 x = 5 ;
10                y = 6 ;
11          close T2 ;

12          T3: task;
13                read x ;
14          close T3 ;

            /* end of declarations */

15          schedule T1 ; /* first executable statement of Main*/
16          schedule T2 ;
17          read flag ;
18          if flag then x = 8 ;
19          write x ;
20          y = 9 ;
21          wait for T2 ;
22          if flag then y = 10 ;
23          write y ;
24          wait for T2 ;
25          schedule T1 ;

26    close Main ;
```

Figure 2.2-1: Example Program with several data flow and synchronization anomalies.

A few of the anomalies are listed below.

1. An uninitialized variable (x) may be referenced at line 5, as task T1 may execute to completion before task T2 begins.

2. The definitions of y as found in task T2 (line 10) and the main program (line 20) may be "useless", since y may be redefined at line 22, before y is ever referenced.

3. y is defined by two processes which act in parallel - thus the reference at line 23 may be to an "indeterminate" value.

4. Variable x is assigned a value by task T2 (line 9) while simultaneously being referenced by the main program at line 19.

5. There is a possibility that task T1 will be scheduled in parallel with itself at line 25 since there is no guarantee that T1 terminated after its initial scheduling.

6. The **wait** at line 24 is unnecessary as T2 was guaranteed to have terminated at line 21, and it has not subsequently been rescheduled.

7. The **wait** at line 6 will never be satisfied as T3 was never scheduled.

2.3 Event Expressions

Clearly many of these error phenomena are interrelated. Hence a more precise categorization and definition system is desirable. We shall modify some notions employed in [7] to gain this precision. In [7] errors were described in terms of anomalous or illegal sequences of events occurring along a path through a program.

For instance the events, "reference", "define", and "undefine" are the significant ones in the detection of undefined variable references and dead

variable definitions. Thus in determining the presence or absence of these errors in a given program, the execution of the program is modelled as the set of all potential execution sequences of these three events happening to each of the program variables. In a single process program any path traceable through the program's flowgraph is taken to represent a potential execution. Now denote the events "reference", "define", and "undefine" by r, d, and u, respectively. Then clearly an undefined variable reference can occur within a program if and only if there is a path subsequence of the for "ur" for some variable and some potential execution. Similarly a dead variable definition is indicated by either a "dd" or "du" path subsequence.

In a concurrent program it is more difficult to determine the potential executions and hence the potential sequences of events. Different processes may be executing simultaneously on different CPU's, or in some non-determinable interleaved order on a single CPU. If these processes operate on shared data, then the sequence of events happening to that data cannot be predicted, even though the code for each process is known. All that can be safely assumed is that every interleaving of the statements of all processes which can act concurrently must be considered a potential execution. Hence the set of execution sequences for a given concurrent program is the set of all possible sequence of events whcih could result from a potential execution of the program.

Thus, for example, in Figure 2.2-1, noting that all variables are initially undefined and that a **write** is a reference, variable x may have the sequence "urd", "udr", "ud", "ur", or "u" by the time line 17 is reached. "urd" corresponds to task T1 acting first, then T2; "udr" corresponds to T2 actually executing before T1 (there is nothing in the program prohibiting this); "u" corresponds to tasks T1 and T2 both being ready to execute, but not actually having done so.

2.4 Error Categorization and Definitions

Using the notation developed above we may now formulate definitions for the errors in which we are interested. The following are anomalies which we wish to detect in all programs. Their detection is more complicated in programs using concurency constructs.

1. Referencing an uninitialized variable. An execution during which this error occurs will have an event sequence of the form "purp"' for some program variable, where p and p' are arbitrary event sequences.

2. A dead definition of a variable. An execution during which this anomaly occurs will have an event sequence the form "pddp"' for some variable.

The following are errors and anomalies which we wish to detect in concurrent code. In the following the **schedule** event will be denoted by an "s", the **wait** by a "w". All processes will be assumed to be in state "u", unscheduled, when not scheduled.

3. Waiting for an unscheduled process. This anomaly is represented by the event expression "pusp"'.

4. Scheduling a process in parallel with itself. This anomaly is represneted by the event expression "pssp"'.

5. Waiting for a process guaranteed to have previously terminated. The expression "pwwp"' is symptomatic of this condition.

6. Referencing a variable which is being defined by a parallel process. There exists a **schedule**, s_0, such that for some variable both the event sequence "ps_0rdp"' and the event sequence "ps_0drp"' are possible.

7. Referencing a variable whose value is indeterminate. There exists a **wait**, w_0, and two separate definition points for a given variable, d_1 and d_2, such that both the event expressions "$pd_1d_2w_0r$" and "$pd_2d_1w_0r$" are possible.

For each of the above errors we will be interested in determining whether they exist in the event expression at a statement (i.e. the event expressions consisting of the preceding events concatenated with the current event) or in the event expression which represents the transformations undergone after leaving a statement. In addition we will wish to distinguish between errors which are guaranteed to occur and those which might occur.

2.5 Program Representation

At the heart of data flow analysis are algorithms which operate on an annotated graphical representation of a program. Single process programs may be represented by a flowgraph [7]. As introduced in reference [8] communicating concurrent process programs may be represented by a process augmented flowgraph, or paf. A paf is formed by connecting the flowgraphs representing the individual processes with special edges indicating all synchronization constraints. In our example language an edge must be created for each ordered pair of nodes of the type (**schedule** p_name, **task** p_name) and (**close** p_name, **wait for** p_name).

Figure 2.5-1 is a paf for the example program of Figure 2.2-1. The creation of the paf for programs in our language is quite straightforward. It is important to note however that most actual languages incorporate synchronization constructs which greatly complicate the construction of the paf. In fact, it is impossible to create a fixed static procedure capable of constructing the paf of any program written in a language which allows run-time determination of tasks to be scheduled and waited for. These issues will be discussed later in this paper.

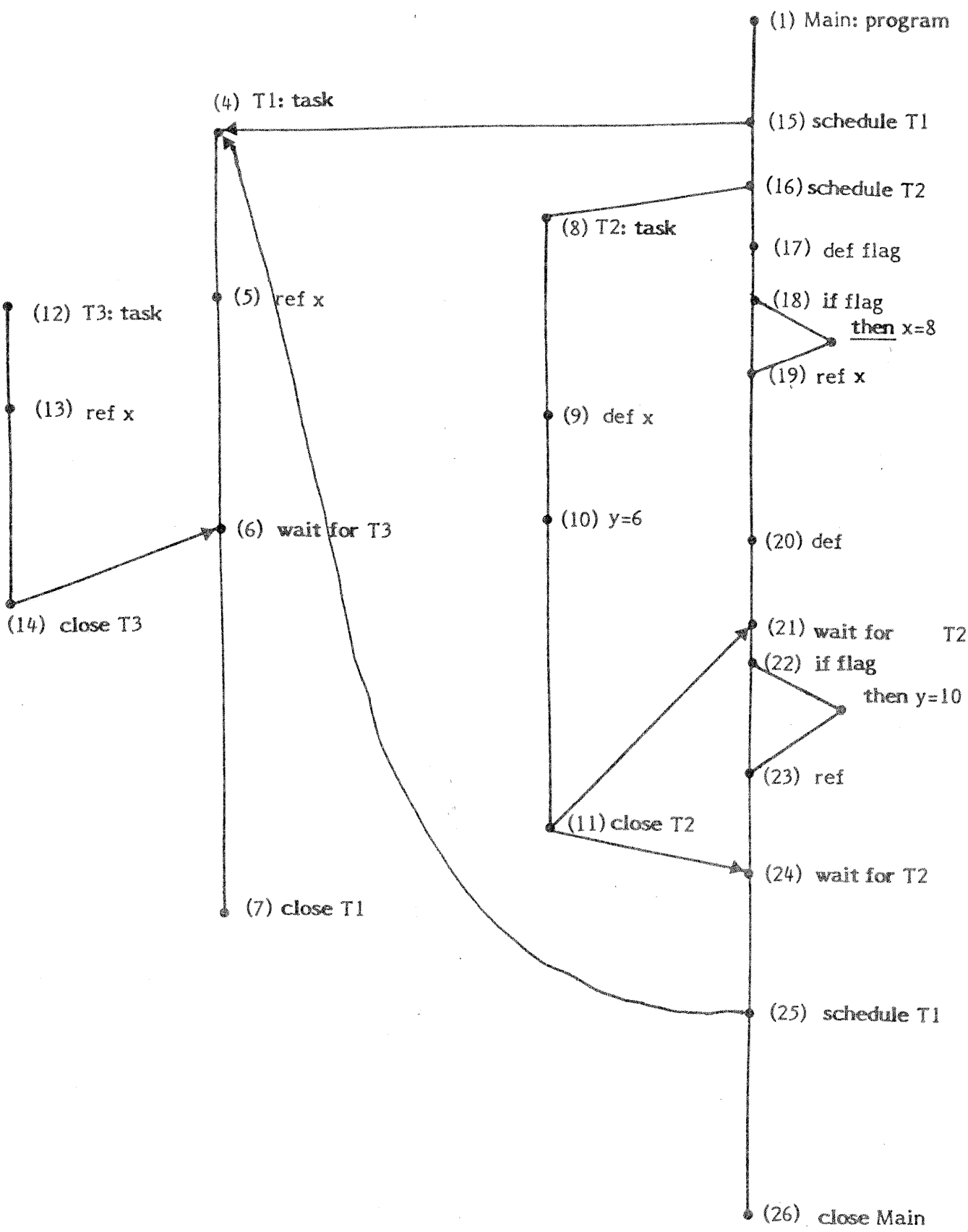

Figure 2.5-1: Process-augmented flowgraph for the program of Figure 2.2-1.

2.6 Data Flow Analysis Algorithms

Data flow analysis algorithms arose out of work in global program optimization [9] [10]. Our usage of them has a different objective, however. The algorithms are described in detail in references [7] and [11]. The purpose of these algorithms is to infer global program variable usage information from local program variable usage information, and then to infer verification and error detection results from the global usage results. The local variable usage is represented by attaching two sets of variables, gen and kill, to each program flow graph node. The global data usage is represented by attaching two sets, live and avail, to each node. The algorithms presented in the references cited assure that, when they terminate: 1) a variable v is in the live set for node n, if and only if there exists a path, p, from n to another node n' such that v is in the gen set at n', but that v is not in the kill set of any node along path p; 2) a variable v is in the avail set for node n, if and only if, for every path, p, leading up to n there exists a node n' on p such that v is in the gen set at n', but v is not in the kill set for any node between n' and n along p.

The implications and usage of these algorithms, and the modifications required to them as a result of concurrency considerations, will become apparent from considering some examples.

3.0 DETECTION OF UNINITIALIZATION ERRORS

Before we examine the large example given in section 2.2, consider the following simple example:

```
1  Main: program;

2       declare integer x ;
3       declare boolean flag ;

4       T1: task;
5             write x ;
6       close T1 ;

7       T2: task;
8             write x ;
9       close T2 ;

10      schedule T1 ;
11      read flag ;
12      wait for T1 ;
13      if flag then  read x ;
14      schedule T2 ;

15 close Main ;
```

The paf for this program is given in Figure 3.0-1. All the nodes are annotated corresponding to the program statements.

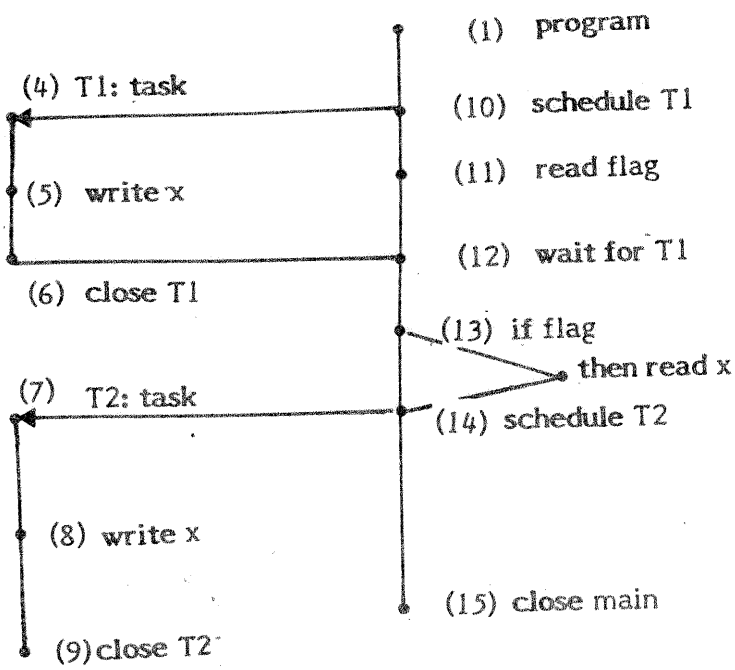

Figure 3.0-1: Paf for program with two uninitialization anomalies.

Let us now consider the uninitialization errors which are present and how they may be detected.

Two uninitialization errors are present in the program. When task T1 is executed the write statement will reference uninitialized variable x. There is no possibility for x to have been initialized, even by the main program which is operating in parallel with the task. When task T2 is executed, there exists a possibility for referencing x as uninitialized. If "flag" has the value **true**, x will have been initialized and no error will occur. If, however, flag is **false**, x will still be uninitialized. Thus we have an instance of an error which "must" occur and an instance of an error which "might" occur. In addition, we may detect each of these anomalies at two different places: the point of variable reference, or at the start node. Thus we see that there are four different subcategories of the uninitialized variable reference error.

The balance of this paper will be devoted to specifying algorithms for detecting the various subcategories of this error and a variety of other errors and phenomena of interest in the analysis of concurrent software. These algorithms will, in general, involve the use of the LIVE and AVAIL procedures described in section 2.6 of this paper. It will be shown that a diversity of diagnostic algorithms can be fashioned by using a variety of criteria for marking the nodes of the program flowgraph with gen and kill notations, and choosing suitable criteria for interpreting the output of the LIVE and AVAIL procedures.

For these reasons, it should be apparent that the algorithms presented here are much involved with placing gen and kill annotations of flowgraph nodes and interpreting live and avail annotations that subsequently appear on flowgraph nodes. This annotation information will be represented by means of bit vectors, denoted in the following way.

If an annotation criterion dictates that a particular variable, say x, is "gen'ed" at a node n, this will be indicated by seting the value of the function gen(n,x) to 1. Otherwise the value of gen(n,x) is 0. The function kill(n,x) is defined similarly. We shall assume that the program unit being analyzed has v variables, and that a one-to-one function, f, has been defined mapping the variables of the program unit onto the integers (1,...,V). Hence a bit vector is defined by the values ($gen(n,x_1)$, ... , $gen(n,x_i)$, ... , $gen(n,x_V)$) where x_i is used to denote the variable x for which f(x)=i. We use this bit vector as the definition of the function GEN(n). KILL(n) is defined similarly.

We shall also assume that there exists algorithmic procedures, LIVE and AVAIL, which operate upon a flowgraph containing N+2 nodes, and annotation functions GEN and KILL defined on the N+2 nodes. We shall always assume that node 0 represents an initialization action immediately preceding the first executable statement of a program unit. Node N+1 represents a termination action which immediatcly follows all statements which end execution of the program unit (i.e. the **close** of the main program) or end execution of any process which is not **waited** for (i.e. all process **close** nodes which are not joined to any **wait** nodes.) LIVE and AVAIL, when executed, compute annotation functions LIVE(n) and AVAIL(n), respectively, defined on the N+2 nodes. The values of LIVE(n) and AVAIL(n) are V-bit vectors for n between 0 and N+1. The bits of LIVE(n) and AVAIL(n) are defined by $live(n,x_i)$ and $avail(n,x_i)$ respectively, where x_i is the variable x for which f(x)=i. The functional dependencies of live(n,x) and avail(n,x) upon gen(n,x) and kill(n,x) are as described in section 2.6 of this paper.

If live(n,x)=1 then we shall say "variable x is live at node n." If avail(n,x)=1 then we shall say "variable x is avail at node n."

We now begin by presenting an algorithm for detecting all statements at which an uninitialized variable reference "must" occur. Referring to the example

in Figure 3.0-1, we see tht this algorithm is designed to detect that the reference to x at statement 5 is a "must" uninitialized reference error.

For this and subsequent algorithms we will need to define functions REF(n) and DEF(n) on the nodes of the flowgraph. REF(n) is a V-bit vector whose i-th component is defined by $\underline{ref}(n,x_i)$. $\underline{ref}(n,x_i)$ is 1 if and only if the statement represented by node n involves a reference to the variable x which is mapped by f onto index value i. Otherwise $\underline{ref}(n,x_i)$ is 0. DEF(n) is defined similarly. $\underline{def}(n,x_i)$ is 1 if and only if the statement represented by node n defines the variable x which is mapped by f onto the index value i. Otherwise $\underline{def}(n,x_i)$ is 0. We also define 0 to be a V-bit vector, all of whose components are 0. 1 is a V-bit vector all of whose components are 1.

ALGORITHM 3.1:

```
for n := 1 to N+1 do
      GEN(n) := 0 ;
      KILL(n) := DEF(n) ;
od ;
GEN(0) := 1 ;
KILL(0) := 0
call AVAIL ;
for n := 1 to N do
      for i := 1 to V do
            if ref(n,i) = 1 and avail(n,i) = 1
                  then print("an uninitialized reference to", f^-1(i),
                     " must occur at node ", n ) ;
            fi ;
      od ;
od ;
```

It is important to observe that algorithm 3.1 is designed to assure that the error message will only be generated when a particular variable cannot possibly be

initialized by any execution sequence leading up to the reference at the node to which the message pertains. In particular it is important for the reader to verify that this algorithm correctly analyzes the program in figure 3.0-1. Figure 3.0-2 shows the contents of each set (gen, kill, etc.) at each node upon termination of algorithm 3.1. Note that variable x is in the avail set at the **write** node in task T1. Also note that x is not in the avail set at the **write** in task T2. Thus we are assured that an error message will be produced for the reference to x at statement 5, but not for the reference at statement 8.

NODE	REF	DEF	GEN	KILL	AVAIL
0			x,flag		x,flag
1					x,flag
2	---	---	---	---	---
3	---	---	---	---	---
4					x,flag
5	x				x,flag
6					x,flag
7					
8	x				
9					
10					x,flag
11		flag		flag	x,flag
12					x
13	flag	x		x	x
14					
15					
16					

Figure 3.0-2: Contents of the data flow analysis sets for the paf of Figure 3.0-1.

We now present an algorithm for detecting "may" uninitialized variable reference errors at a node. This algorithm is designed to detect a variable reference occuring at a statement for which there exists an execution sequence which leads up to the statement and which does not initialize the variable. Referring to figure 3.0-1 again, clearly such an error occurs at statement 5, but of more interest there is also such an error at statement 8. Algorithm 3.1 does not detect the error at statement 8, but algorithm 3.2 will.

Before presenting algorithm 3.2, we must first discuss a necessary modification to the AVAIL algorithm. The AVAIL algorithm is devised to assure that at termination a variable, x, will be avail at n if and only if for every possible execution of the program leading up to n, there is a previous gen of x without an intervening kill of x. For single process programs, AVAIL(n) is computed correctly at every flowgraph node n provided that the following equality is achieved at termination of the AVAIL algorithm.

$$\text{AVAIL}(n) = \underset{\substack{\text{all } n_i, \\ \text{immediate} \\ \text{predecessors} \\ \text{of } n}}{\textbf{intersect}} \left(\text{GEN}(n_i) \textbf{ union } (\text{AVAIL}(n_i) \textbf{ intersect not } \text{KILL}(n_i))\right)$$

For concurrent process programs it is helpful to define a somewhat different equilibrium condition under certain circumstances. We proceed as follows.

Suppose n_w is a flowgraph node which represents a **wait** statement. In the paf, G, of the program containing n_w, n_w will be the tail of some edges which are usual flow of control edges, and the tail of at least one edge whose head represents the termination activity for a concurrent task. Suppose now that $(f_i)_{i=1}^{F}$ represents the set of heads of usual flow of control edges whose tails are n_w, and that $(p_i)_{i=1}^{P}$ represents the set of concurrent task termination nodes which are the heads of edges have n_w as their tails. Now create a new graph node n_w',

delete the edges ((f_1,n_w), ... , (f_F,n_w)) and replace them by the edges ((f_1,n_w') , ... , (f_F,n_w'), (n_w',n_w)). Suppose this is done for every **wait** node in G. Denote the resulting graph by G'. Now compute AVAIL(n) as usual, except use the following equilibrium condition at the **wait**-for-any nodes of G' only.

$$(*)\ AVAIL(n_w) = \underset{(p_i)_{i=1}^{P}}{\textbf{intersect}}\ AVAIL(p_i)\ \ \textbf{union}\ AVAIL(n_w')$$

A different equilibrium condition is required at **wait**-for-all nodes.

$$(*)\ AVAIL(n_w) = \underset{(p_i)_{i=1}^{P}}{\textbf{union}}\ AVAIL(p_i)\ \ \textbf{union}\ AVAIL(n_w')$$

The resulting AVAIL(n) bit vectors will be quite useful to us. Thus let us denote by AVAIL* the algorithm which employs the starred formulas as the equilibrium conditions for all of the wait nodes of G'. In all the algorithms which follow we assume that graph G' has been created and that the analysis takes place on that graph.

We can now state algorithm 3.2.

ALGORITHM 3.2:

```
for n := 1 to N+1 do
      KILL(n) := 0 ;
      GEN(n)  := DEF(n) ;
od ;
KILL(0) := 1 ;
GEN(0) := 0 ;
call AVAIL* ;
for n := 1 to N do
      for i := 1 to V do
            if ref(n,i) = 1 and avail(n,i) = 0
                  then print("an uninitialized reference to",f^-1(i),
                        "may occur at node", n);
            fi ;
      od ;
od ;
```

Using a different algorithm we may indicate to the programmer the event sequence associated with this anomaly. Unfortunately many such event sequences are unexecutable. This problem and partial remedies to it are discussed elsewhere [2]. In our example here, variable x is not in the avail set at either write statement in task T1 or T2. Thus the potential for error is reported at both nodes. In this case the associated event sequences are clearly executable.

We now present an algorithm for detecting at the start node all the "must" uninitialization errors. In the example of Figure 3.0-1 we are again interested in detecting the error which occurs at the reference to x in statement 5, except in this case the point of detection (and error message generation) will be the start node of the program.

Analogous to the presentation of Algorithm 3.2 we must discuss a necessary modification to the LIVE algorithm. The LIVE algorithm is devised to assure that at termination a variable, x, will be live at n if and only if there exists an execution sequence beginning at n such that there is a gen of x before there is a kill of x. For single process programs, LIVE(n) is computed correctly at every flowgraph node n provided that the following equality is achieved at termination of the LIVE algorithm.

$$\text{LIVE}(n) = \underset{\substack{\text{all } n_i, \\ \text{immediate} \\ \text{successors} \\ \text{of } n}}{\textbf{union}} (\text{GEN}(n_i) \textbf{ union } (\text{LIVE}(n_i) \textbf{ intersect not } \text{KILL}(n_i)))$$

For concurrent programs it is useful to define a different equilibrium condition which is applied only at schedule nodes.

$$(*) \quad \text{LIVE}(n) = \underset{\substack{\text{all } n_i, \\ \text{immediate} \\ \text{successors} \\ \text{of } n}}{\textbf{intersect}} (\text{GEN}(n_i) \textbf{ union } (\text{LIVE}(n_i) \textbf{ intersect not } \text{KILL}(n_i)))$$

We shall denote by LIVE* the algorithm which creates the live sets, employing (*) at all **schedule** nodes of G. (A graph G' is not required in this case as a schedule node only has a single control flow edge leaving it. All others lead to a **task** initialization node.)

ALGORITHM 3.3:

```
for n := 0 to N do
        GEN(n) := DEF(n) ;
        KILL(n) := REF(n) ;
od ;
GEN(N+1) := 1 ;
KILL(N+1) := 0 ;
call LIVE* ;
for i := 1 to V do
        if live(0,i) = 0
                then print( " an uninitialized reference to ", f^-1(i),
                        " will occur ");
        fi ;
od ;
```

In the example of Figure 3.0-1 variable x will be missing from the live set at the start, due to the kill present at line 5. (The live set at the **wait** node does contain x, however, as the error in task T2 is dependent on the execution sequence taken.)

The detection of possible errors is achieved through the following algorithm.

ALGORITHM 3.4:

```
for n := 0 to N+1 do
      GEN(n) := REF(n) ;
      KILL(n) := DEF(n) ;
od ;
call LIVE ;
for i := 1 to V do
      if live(0,i) = 1
            then print ( " an uninitialized reference to ", f^-1(i),
                  " may occur ");
      fi ;
od ;
```

In our now tired example variable x is in the live set at the start because of the references in both tasks. (Now note that the **wait** node has x in its live set - indicating that there is an execution sequence following which encounters a reference before any initialization. An error in that execution sequence would depend on x not being initialized before the **wait**, which of course it is not.)

To summarize briefly, two basic algorithms are involved. One computes live sets, the other avail sets. With suitably created gen and kill sets attached to the paf and special rules applied at wait nodes during the computation of avail and at **schedule** nodes during the computation of live, a comprehensive set of programming anomalies may be detected in concurrent process programs.

We are not out of the woods yet, however.

4.0 PARCELING OF ANALYSIS ACTIVITIES

Let us now return to the example of section 3, and modify the program slightly. In that example task T2 performed the same actions as task T1. There was no need to declare two tasks, except that it made our analysis simpler, as we shall see. Below we show the program written with only a single task declaration.

```
1  Main: program;

2       declare integer x ;
3       declare boolean flag ;

4       T1: task;
5             write x ;
6       close T1 ;

7       schedule T1 ;
8       read flag ;
9       wait for T1 ;
10      if flag then  read x ;
11      schedule T1 ;

12 close Main ;
```

The paf for this program is given in figure 4.0-1. As before, the nodes are numbered and annotated with the corresponding statements.

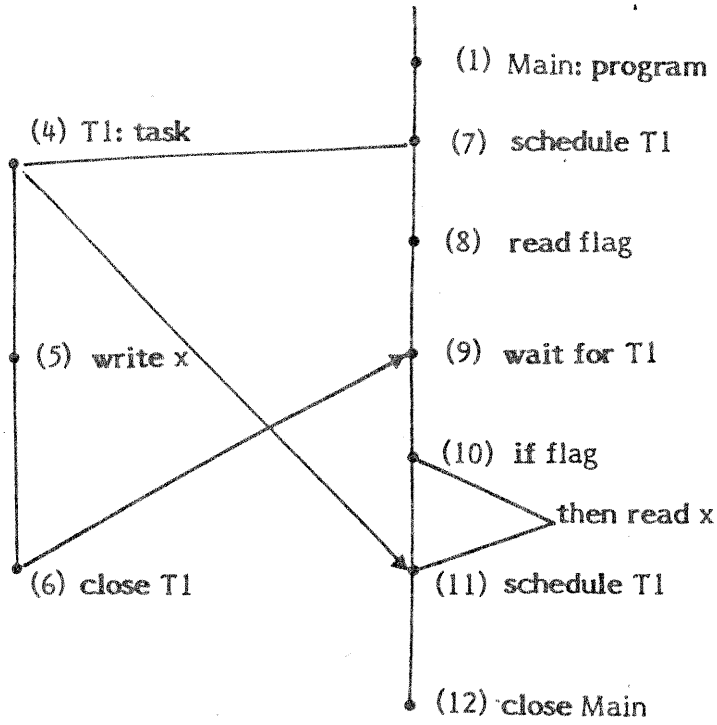

Figure 4.0-1: Paf for program with two uninitialization errors, written with a single task.

Note that the paf has been drawn with two edges entering the task's start node. Suppose now that we wish to look for "must" uninitialization errors, and detect them at the point of reference. We are therefore concerned with computing the avail sets as described in algorithm 3.1. Using this algorithm on the graph as shown will result in x not being in the avail set at the reference (at line 5). Thus we cannot say that whenever this node is executed an uninitialization error will result. Indeed this is a correct statement as the second time the task is scheduled there is only the possibility for an error at this line. This is somewhat unsatisfactory, though, as it is clear that the first time T1 is scheduled an error will occur, regardless of the execution sequence. We may improve the strength of our analysis in this regard by parceling the paf and detecting the error not at the reference, but at the point where the task is scheduled. One cost of doing this is that we will not be able to point directly to the statement in the task at which the error occurs.

Our method for doing this is based on the technique presented by Fosdick and Osterweil for handling external procedures when performing data flow analysis on single process programs [7]. Their technique abstracts the data flow in each procedure using the LIVE and AVAIL algorithms, and attaches this abstracted information to all invoking nodes for each procedure. Data flow analysis can then be perfomed on the invoking procedure. We here adapt this technique to the analysis of tasks for anomalous event sequences. The data usage patterns within each task are determined using LIVE and AVAIL. These abstractions are attached to all **schedule** and **wait** nodes referring to each task. Analysis of this remarked ("trimmed") graph then proceeds as described previously.

For the example of Figure 4.0-1 the analysis will proceed roughly as follows. The algorithms 3.1, 3.2, 3.3, and 3.4 would be run for the local variables

in task T1 first. (Since there are no local variable this step is omitted.) Next T1 is annotated as described in algorithm 3.3 for global variables (in this case x). The LIVE algorithm is run, giving the result that x is live at the task node, 4. We consequently label nodes 7 and 11 with ref(n,x)=1, indicating that execution of node 7 or node 11 always results in a subsequent reference to x. We can now run algorithms 3.1, 3.2, 3.3, and 3.4 for the variables local to Main. Algorithm 3.1 will show that x is avail at 7 indicating an uninitialized reference to x will always occur as a consequence of executing node 7. Ideally the resulting error message will indicate that the error actually occurs "somewhere" in the scheduled task. Determination of the error's precise location would be relegated to a separate (depth-first) scan of the task.

Clearly we could continue passing up data usage abstractions through an arbitrary number of levels of task scheduling. The restriction we have to impose on the program in order to adopt this technique is that the process invocation graph be acyclic. In the single process program situation there is an analogous restriction that the subroutine call graph be acyclic: recursion is prohibited. This prohibition exists for multiprocess programs as well, but the process invocation graph is also required to be acyclic. This is a stronger restriction as it is possible to have a cyclic process invocation graph which does not involve any recursion, either on the process or the subroutine level. For the moment we are satisfied that a significant class of programs is nevertheless being addressed, but further investigation is clearly called for here.

5.0 ADDITIONAL REFERENCE/DEFINITION ANOMALIES

5.1 Referencing a Variable While Defining It in a Parallel Process

Let us now reconsider the example of Section 2.2. At line 5, in task T1, we have a reference to variable x which, in absence of a "fortunate" sequencing of events, will be uninitialized when the task is first scheduled. If algorithm 3.1 is run on the paf corresponding to this program (Figure 2.5-1), an "always" uninitialization error will be detected at the reference. (We are assuming that the analysis is carried out in the parceled manner described in the preceding section, as the second time the task is scheduled the possibility exists that x will have been defined.) Would it be proper to report this as an always error? What is termed a "fortunate" sequence really makes this an anomaly. It is conceivable that known operating environment conditions guarantee that the initialization performed by task T2 transpires before the reference in task T1. A "sometimes" error message is unsatisfactory, though, as such "guarantees" are outside the domain of the program. This confusion is due to the referencing and defining of a variable by two processes which may be executing in parallel. This construction, besides impairing our other analyses in the manner described, seems inherently dangerous and should be reported as an anomaly in its own right.

This anomaly may be detected in a rather naive manner given that we can determine which sections of the program may be operating concurrently. Let us assume that the paf is parcelled into S subgraphs, G_i. Each section corresponds to a task or a portion of a task. Briefly, section boundary nodes are **program**, **task**, **close**, and **wait** nodes. (Our notion of a section is roughly equivalent to that of a task which contains no **wait** statements.)Let us further assume that we have at our disposal a boolean function, PARALLEL, which determines which sections can execute in parallel. That is, PARALLEL defines a function of two variables, i and

j, such that PARALLEL(i,j) is true if and only if G_i and G_j represent sections which might execute in parallel. It is important to note that the algorithm for determining this is not trivial. Indications of how such an algorithm can be constructed may be found in references 13 and 14. Based on these assumptions we can now state an algorithm for detecting the possibility of referencing and defining a variable from parallel tasks or sections. Suppose the nodes of graph G_i are numbered from $n_{i,0}$, the logical predecessor to the sections start node, to n_{i,l_i} the logical successor to the sections final node. For clarity, also assume as before that f maps all the variables of the program onto the integers 1-V.

Algorithm 5.1:

```
for i = 1 to S do ;
      for j = 1 to l_i do
            REF(n_{i,0}) := REF(n_{i,0}) union REF(n_{i,j}) ;
            DEF(n_{i,0}) := DEF(n_{i,0}) union DEF(n_{i,j}) ;
      od ;
od ;
for i := 1 to S do
      for j := 1 to S do
            if i ≠ j
                  then if PARALLEL(i,j)
                        then if REF(n_{i,0}) intersect DEF(n_{i,0}) = 1
                              then print (" the following may be referenced ",
                              "and defined in parallel by sections", i, "and",j);
                                    for v := 1 to V do
                                       if ref(n_{i,0},v) = 1 and
                                          def(n_{i,0},v) = 1
                                           then print( f^{-1}(v));
                                       fi ;
                                    od ;
                              fi ;
                        fi ;
            fi ;
      od ;
od;
```

This algorithm detects the possibility of references and definitions occurring in parallel. We can also construct an algorithm that determines when this error must occur, regardless of the execution paths within the processes. The only change required to algorithm 5.1 is in the creation of the REF and DEF sets at the nodes $n_{i,0}$. The REF sets at $n_{i,0}$ would be computed by an algorithm similar to 3.3, and the DEF sets by an algorithm similar to 3.1.

5.2 Unused Variable Definitions

A programming anomaly not truly erroneous but which often indicates the presence of a design error is that of unused variable definitions. The example of section 2.2 has such an anomaly in it. Variable y is defined both by task T2 and by the main program (at line 20). Y is then possibly redefined at line 22, before ever being referenced. (We will examine the anomalous situation which occurs at the reference to y (line 23) in the next section.) This anomaly may be detected by techniques very similar to those presented in section 3. Here, as with uninitialization errors, there are four cases to examine: detecting errors which always occur through examining all possible event sequences which follow a node, detecting the possibility of such errors, detecting errors which always occur through examining all event sequences preceding a node, and detecting possible errors by examining the preceding event sequences. We present here only the algorithm for determining the anomalous situation where a variable v is defined at node n, yet on all paths leading to n, v has been previously defined without any intervening reference. Algorithms for the other three related anomalous situations should be derivable by analogy. Not that the presence of reference-definition in parallel anaomalies may impair the quality of the analysis here, like as was described previously.

The procedure given here assumes that the program graph has been parcelled into task subgraphs. We assume that the process invocation graph is acyclic and that the labelling used in algorithm 5.1 is used here as well. This particular error also requires that we define a new equilibrium condition to be applied during the computation of the AVAIL sets at **wait** nodes. The new condition is as follows:

$$(**)\ AVAIL(n_w) = \underset{(p_i)_{i=1}^{p}}{\text{intersect}}\ AVAIL(p_i)\ \textbf{union}\ AVAIL(n'_w) - \underset{(p_i)_{i=1}^{p}}{\text{intersect}}\ REFED(p_i)$$

This condition applies at **wait**-for-anys. At **wait**-for-alls:

$$(**)\ AVAIL(n_w) = \underset{(p_i)}{\text{union}} \bigcap_{i=1}^{p} AVAIL(p_i)\ \text{union}\ AVAIL(n_w') - \underset{p_i}{\text{union}}\ REFED(p_i)$$

We shall denote by AVAIL** the algorithm which employs the double-starred formulas as the equilibrium conditions for all the **wait** nodes of G'. REFED(n) is a V-bit vector defined during the computation of AVAIL** which is used to save the value of some intermediate AVAIL sets.

ALGORITHM 5.2:

```
declare bit vector PROCESSED (1:S) ;
PROCESSED := 0 ;
while PROCESSED ≠ 1 do
    for i := 1 to S do
        if processed_i = 0 and processed_t = 1 for all tasks, t,
            for which G_i waits
        then
            processed_i := 1 ;
            for j = 1 to l_i do
                GEN(n_i,j) := REF(n_i,j) ;
                KILL(n_i,j) := 0 ;
            od ;
            KILL(n_i,0) := 1 ;
            call AVAIL* ;
            REFED(n_i,l_i) := AVAIL(n_i,l_i) ;
            for j := 1 to l_i do
                GEN(n_i,j) := DEF(n_i,j) ;
                KILL(n_i,j) := REF(n_i,j) ;
            od ;
            KILL(n_i,0) := 1 ;
            call AVAIL** ;
            for j := 1 to l_i do ;
                if DEF(n_i,j) intersect AVAIL(n_i,j) ≠ 0
                then print(" the definition(s) at node ",n_i,j,
                        "is always immediately preceded by another",
                        " definition. The variable(s) is (are): ");
                        for k := 1 to V do
                            if def(n_i,j,k)=1 and avail(n_i,j,k)=1 then print( f^-1(k));
                            fi;
                        od;
                fi ;
            od ;
        fi ;
    od;
od;
```

5.3 Referencing a Variable of Indeterminate Value

In the above presentation we deferred discussion of the anomalous data flow situation existing at the reference to variable y occurring at line 23 of the example program in section 2.2. Y is defined by task T2 at line 10, by the main program at line 20, and possibly again in the main program at line 22. If for the moment we ignore the definition at line 22, then it is definately indeterminate whether the definition from the task or from the main program is referenced at line 23. If we acknowledge the presence of the definition at line 22, depending on the event sequence (namely whether variable flag is **true**) the reference at line 23 may be to an indeterminate value.

The algorithm we now present is designed to detect indeterminate reference anomalies which will occur regardless of execution sequence. The anomalies will be detected at the point of indeterminate reference.

Algorithm 5.3:

```
declare bit vector PROCESSED (1:S) ;
PROCESSED := 0 ;
while PROCESSED ≠ 1 do
    for i := 1 to S do
        if processed_i = 0 and processed_t = 1 for all tasks, t,
            for which G_i waits
        then
            processed_i := 1 ;
            for j = 1 to l_i do
                GEN(n_i,j) := DEF(n_i,j) ;
                KILL(n_i,j) := 0 ;
            od ;
            KILL(n_i,0) := 1 ;
            call AVAIL* ;
            DEFED(n_i,l_i) := AVAIL(n_i,l_i) ;
            for j:= 1 to l_i do
                GEN(n_i,j) := 0 ; KILL(n_i,j) := DEF(n_i,j);
            od ;
            for all w_i,a, wait nodes in G_i do
                COUNT := 0 ;
                for all predecessor nodes, p_i,a,b, of w_i,a do
                    COUNT := COUNT vector-add AVAIL(p_i,a,G);
                od ;
                GEN(w_i,a) := 0 ;
                for v := 1 to V do
                    if COUNT_v greater than 1
                    then GEN(w_i,a,v) := 1 ;
                    fi ;
                od ;
            od ;
            call AVAIL;
            for j := 1 to l_i do
                if AVAIL(n_i,j) intersect REF(n_i,j) ≠
                then print("indeterminate reference at", n_i,j);
                fi ;
```

```
                od ;
                AVAIL (n_{i,l_i}) := DEFED (n_{i,l_i}) ;
            fi ;
        od;
    od;
```

6.0 PROCESS SYNCHRONIZATION ANOMALIES

As an outgrowth of our investigation into the detection of data flow anomalies in concurrent process software it became clear that some forms of synchronization errors could be detected in essentially the same manner. We have alluded to the nature of these errors in the introduction. They will now be considered in detail. Note that in form the synchronization anomalies are analagous to data flow anomalies. In addition, as with data flow, many of the anomalies are not strictly errors, but they are conditions which may be interpreted as erroneous in the sense of indicating deeper problems. At the very least they represent conditions which should be clearly documented.

6.1 Waiting for an Unscheduled Process

This anomaly is perhaps the most apparent, and is closest in form to the data flow anomalies already discussed. The example of section 2.2 contains such an error at line 6 in task T1. Task T3 is never scheduled, yet T1 waits for it. The analogue is to detection of uninitialized variables. As such we will present algorithm 3.1 rewritten to detect this anomaly. Thus we are interested in detecting anomalies which must occur, and the anomaly is to be detected at **wait** nodes. Our notation requires that we introduce functions SCH(n), WAIT_ALL(n), and WAIT_ANY(n). All function values are T-bit vectors. We shall assume that the program unit being analyzed has T processes, and that a one-to-one function, g, has been defined mapping the process names onto the integers (1,...,T). The i-th component of SCH(n) is defined by $\underline{sch}(n,t_i)$. $\underline{sch}(n,t_i)$ is 1 if and only if the statement represented by node n schedules the task t which is mapped by the function g onto index value i. WAIT_ALL and WAIT_ANY are similarly defined, for the two types of **wait** statements in our language.

Algorithm 6.1:

```
for n := 1 to N+1 do
        GEN(n) := 0 ;
        KILL(n) := SCH(n) ;
od ;
GEN(0) := 1 ;
KILL(0) := 0 ;
call AVAIL ;
for n := 1 to N do
        for i := 1 to T do
                if ( wait all(n,i) = 1 or wait any(n,i) = 1) and avail(n,i)=1
                        then print (" the reference to process ", g^-1(i),
                          " at node ", n, " is to a process which has not been scheduled.");
                fi ;
        od ;
od ;
```

In figure 2.5 task T3 will be in the avail set at the node corresponding to line 6, thus the error will be detected.

As may be expected, there is also an analogue to the reference-definition in parallel condition here. The following program presents such a condition.

```
1       Main: program;

2               T1: task;
3                       schedule T2 ;
4               close T1 ;

5               T2: task;
6                       /* do something */
7               close T2 ;

8               schedule T1 ;
9               /* do something */
10              wait for T2 ;

11      close Main ;
```

In this program there is the possibility that task T2 will be scheduled before the wait at line 10 is encountered. Our analysis described above will cause an

"always" message to be generated. Thus we need to perform "schedule/wait in parallel" analysis to give a complete description of the situation. This would be performed in a manner analogous to that of reference/definition in parallel analysis.

6.2 Waiting for a Process Guaranteed to Have Already Terminated

The example of section 2.2 still has additional errors to consider. At line 24 the main program waits for task T2 to complete. Yet the task was already assured to have terminated at line 21. The second **wait** is thus superfluous and possibly misleading. Since our language syntax allows us to specify **wait**-for-all and **wait**-for-any we must be careful to distinguish the errors which we will detect and the algorithms which apply in each case. To indicate the nature of our technique we will just consider a single case: looking for constructs which, regardless of event sequence, assure us that at least one of the processes named in a **wait**-for-all has in fact already terminated at a previous **wait**.

Algorithm 6.2:

```
for n := 1 to N+1 do
      KILL(n) := SCH(n) ;
      GEN(n) := WAIT_ALL(n) ;
      if the statement represented by node n is a task statement for t_i
            then gen(n,t_i) := 1 ;
      fi ;
od;
GEN(0) := 1 ;
KILL(0) := 0 ;
call AVAIL ;
for n := 1 to N do
      for i := 1 to T do
            if avail(n,i) = 1 and wait all(n,i) = 1
                  then print (" termination has already been ensured for task",
                    g^-1(i), " at node ", n);
```

```
        fi ;
    od ;
od ;
```

In figure 2.5-1 task T2 is in the avail sets of all predecessor nodes of the node corresponding to the first **wait** (line 21), but is in only one of the avail sets of the predecessor nodes of the **wait** at line 24. Thus the first **wait** is correct, while the second is anomalous.

The algorithm we have presented may be easily modified to detect the possibility of anomalies. To detect anomalies occurring at **wait**-for-anys we must develop new procedures to account for situations such as:

```
wait for T1 or T2 ;
        .
        .
        .
wait for T1 or T2 ;
```

In the absence of other synchronization statements the second **wait** is spurious; satisfaction of the first **wait** guarantees immediate satisfaction of the second.

6.3 Scheduling a Process in Parallel with Itself

The last synchronization anomaly which we shall examine is that of scheduling a process to execute in parallel with an already active incarnation of the same process. In the example of section 2.2 there is an instance of this error at line 25, where task T1 is scheduled for the second time (the first being at line 15). At no point in any process, let alone before the second **schedule**, has T1 been guaranteed to have terminated.

We will present the algorithm for detecting situations where, regardless of event sequence, termination has not been assured by the time a **schedule** is reached.

Algorithm 6.3:

```
for n := 1 to N+1 do
      GEN(n) := SCH(n) ;
      KILL(n) := WAIT_ALL(n) union WAIT_ANY(n) ;
od ;
KILL(0) := 1 ;
GEN(0) := 0 ;
call AVAIL* ;
for n := 1 to N do
      for i := 1 to T do
            if sch(n,i) = 1 and avail(n,i) = 1
                  then print (" termination of process ", g^-1(i),
                    " has never been ensured before the schedule at node ", n) ;
            fi ;
      od ;
od ;
```

If this algorithm is applied to our example an error will be detected at line 25.

As may be expected, if a **schedule** may be performed in parallel with a **wait** (on the same process) the quality of our analysis is impaired. In particular, if such a condition exists algorithm 6.3 will detect a "for sure" error, where in fact there is an event sequence where termination takes place.

7.0 CONCLUSION

7.1 Summary

In this paper we have presented several algorithms useful in the detection of data flow and synchronization anomalies in programs involving concurrent processes. Data flow is analyzed on an interprocess and interprocedural basis. The basis of the technique is analysis of a process augmented flowgraph, a graph representation of a system of communicating concurrent processes. The algorithms have excellent efficiency characteristics, and utilize basic algorithms which are present in many optimizing compilers. A procedure is outlined which allows analysis to proceed on "parcels" of the subject program. Only the most basic synchronization constructs have been considered, however.

7.2 Open Problems

Several matters discussed in this presentation clearly warrant further investigation. The most pressing need now is a consideration of additional synchronization and communication constructs. These will introduce new classes of errors and may require that changes be made to the algorithms presented here.

One issue not addressed here is the creation of correct process-augmented flowgraphs. In our sample language this was a relatively trivial task, but as additional (real) synchronization constructs are added significant problems are anticipated. It is not clear at this point if "correct" pafs can always be generated. The analysis schemes may require alteration to accommodate such a situation.

Dynamic determination of synchronization paths has not been considered at all here, but work has been done in this area [15]. Likewise recusive procedures and processes have been precluded. Work has been done in data flow analysis of recursive routines [16], but it appears inadequate for the analysis performed here.

Footnotes

1. R. N. Taylor is with the Space and Military Applications Division, Boeing Computer Services Company, P.O. Box 24346, Seattle, Washington, 98124. This work was supported by the National Aeronautics and Space Administration, under contract NAS1-15253.

2. L. J. Osterweil is with the Department of Computer Science, University of Colorado, Boulder, Colorado, 80309. This work was supported in part by the National Aeronautics and Space Administration under contract NAS1-15253 and by the National Science Foundation under Grant MCS77-02194.

References

1. L.J. Osterweil and L. D. Fosdick, "DAVE- A validation, error detection, and documentation system for FORTRAN programs," **Software-Practice and Experience**, Vol. 6, 473-486, 1976.

2. P. Brinch Hansen, "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering, Vol. SE-1 pp. 199-207, June 1975.

3. N. Wirth, "Modula: A Language for Modular Multiprogramming," Software-Practice and Experience, Vol. 7, no. 1, pp. 3-35, January, 1977.

4. DoD Requirements for High Order Computer Programming Languages, 9. STEELMAN, June, 1978.

5. F. H. Martin, "HAL/S - The Avionics Programming System for Shuttle," Proceedings AIAA Conference on Computers in Aerospace, Los Angeles, California, pp. 308-318, November, 1977.

6. T.A. Straeter, et. al. "Research Flight Software Engineering and MUST -An Integerated System of Support Tools," Proceedings COMPSAC 77, Chicago, Illinois, pp. 392-396, November 1977.

7. L.D. Fosdick and L.J. Osterweil, "Data Flow Analysis in Software Reliability," Computing Surveys, Vol. 8, no. 3, pp.305-330, September 1976.

8. R.N. Taylor and L.J. Osterweil, "A Facility for Verification, Testing, and Documentation of Concurrent Process Software," Proceedings COMPSAC 78, Chicago, Illinois, pp.36-41, November, 1978.

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

10. F. E. Allen and J. Cocke, " A program data flow analysis procedure," Communications of the ACM, Vol. 19, no. 3, pp. 137-147, March 1976.

11. M.S. Hecht and J.D. Ullman, "A simple algorithm for global data flow analysis problems," SIAM Journal of Computing, Vol. 4, pp. 519-532, December 1975.

12. L.J. Osterweil, "The Detection of Unexecutable Program Paths through Static Data Flow Analysis," Department of Computer Science Technical Report #CU-CS-110-77, University of Colorado, 1977.

13. W. Riddle, G. Bristow, C. Drey, and B. Edwards, "Anomaly Detection in Concurrent Programs," Department of Computer Science Technical Report, University of Colorado, January, 1977.

14. J. L. Peterson, "Petri Nets," Computing Surveys, Vol. 9, Number 3, pp. 223-252, September, 1977.

15. J. Reif, "Data Flow Analysis of Communication Processes," University of Rochester, 1978.

16. J.M. Barth, "A Practical Interprocedural Data-Flow Analysis Algorithm," Communications of the ACM, Vol. 21, No. 9, pp. 724-736, September 1978.

References

[Alfo 77] M. W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, SE-3 pp. 60-69 (Jan. 1977).

[Alle 76] F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," CACM, 19, pp. 137-147 (March 1976).

[Balz 69] R. M. Balzer, "EXDAMS: Extendable Debugging and Monitoring System," Proc. AFIPS 1969 Spring Joint Computer Conference 34 AFIPS Press, Montvale, N.J.

[Bell 77] T. E. Bell, D. C. Bixler and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Software Eng., SE-3 pp. 49-60, (Jan. 1977)

[Blac 77] R. K. E. Black, "Effects of Modern Programming Practice on Software Development Costs," Proceeding Fall Compcon 77 pp. 250-253 (Sept. 1977).

[Boll 79] L. A. Bollacker, "Detecting Unexecutable Paths Through Program Flow Graphs," Masters Thesis, Dept. of Comp. Sci., Univ. of Colorado at Boulder, 1979.

[Brow 78] J. R. Brown, "Programming Practices for Increased Software Quality," in Software Quality Management Petrocelli Books, New York City, 1978.

[Chea 78] T. E. Cheatham, Jr. and D. Washington," Program Loop Analysis by Solving First Order Recurrence Relations," Harvard Univ. Center for Research in Computing Technology, TR-13-78.

[Clar 76] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, SE-2 pp. 215-222 (Sept. 1976).

[Elsp 72] B. Elspas, K. N. Levitt, R. J. Waldinger and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys 4 pp. 97-147 (June 1972).

[Fair 75] R. E. Fairley, "An Experimental Program Testing Facility," Proc. First National Conf. on Software Eng., IEEE Cat. #75CH0992-8C pp. 47-55 (1975).

[Floy 67] R. W. Floyd, "Assigning Meanings to Programs," in Mathematical Aspects of Computer Science 19 J. T. Schwartz (ed.) Amer. Math. Soc. Providence, R.I. pp. 19-32 (1967).

[Fosd 76] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability," ACM Computing Surveys 8 pp. 305-330 (Sept. 1976).

[Gris 70] R. Grishman, "The Debugging System AIDS," AFIPS 1970 Spring Joint Computer Conf., 36 AFIPS Press, Montvale, N.J. pp. 59-64.

[Hant 76] S. L. Hantler and J. C. King, "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys 8 pp. 331-354 (Sept. 1976).

[Hech 75] M. L. Hecht and J. D. Ullman, "A Simple Algorithm for Global Data Flow Analysis Problems," SIAM J. Computing 4 pp. 519-532 (Dec. 1975).

[Howd 78] W. E. Howden, "DISSECT - A Symbolic Evaluation and Program Testing System," IEEE Trans. on Software Eng., SE-4 pp. 70-73 (Jan. 1978)

[King 76] J. C. King, "Symbolic Execution and Program Testing," CACM 19 pp. 385-394 (July 1976).

[Lisk 75] B. H. Liskov and S. N. Zilles, "Specification Techniques for Data Abstractions," IEEE Trans. on Software Eng., SE-1 pp. 7-19 (1975).

[Lisk 77] B. H. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," CACM 20 pp. 564-576 (Aug. 1977).

[Lond 75] R. L. London, "A View of Program Verification," 1975 International Conf. on Reliable Software, IEEE Cat. #75-CH0940-7CSR pp. 534-545 (1975).

[Mill 74] E. F. Miller, Jr., "RXVP, Fortran Automated Verification System," Program Validation Project, General Research Corp., Santa Barbara, Calif. (Oct. 1974).

[Oste 76] L. J. Osterweil and L. D. Fosdick, "DAVE - A Validation, Error Detection, and Documentation System for FORTRAN Programs," Software - Practice and Experience 6 pp. 473-486 (Sept. 1976).

[Oste 77] L. J. Osterweil, "The Detection of Unexecutable Program Paths Through Static Data Flow Analysis," Proceedings COMPSAC 77, IEEE Cat. #77CH1291-4C, pp. 406-413 (1977).

[Parn 72] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," CACM 15, pp. 1053-1058 (Dec. 1972).

[Rama 75] C. V. Ramamoorthy and S.-B. F. Ho, "Testing Large Software With Automated Software Evaluation Systems," IEEE Transactions on Software Engineering SE-1 pp. 46-58 (March 1975).