Data Flow Analysis of Concurrent Systems that use the Rendezvous Model of Synchronization

Douglas Long*
Lori A. Clarke†

COINS Technical Report 91–31
July 1991

*Department of Computer Science
Lafayette College
Easton, Pennsylvania 18042-1781

†*Software Development Laboratory*
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This report to appear in
Proceedings of the Fourth Conference on Testing, Analysis, and Verification, October 1991

This work was supported in part by the Ofice of Naval Research, grant N00014-90-J-1791, and by the National Science Foundation, grant CCR-87-04478, in cooperation with the Defense Advanced Research Projects Agency (ARPA Order No.6104).

Data Flow Analysis of Concurrent Systems that use the Rendezvous Model of Synchronization

Douglas Long
Department of Computer Science
Lafayette College
Easton, Pennsylvania 18042-1781

Lori A. Clarke
Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

1 Introduction

Because of the complex communication patterns supported in concurrent systems, it is extremely difficult for developers to understand and reason about these systems. Thus, it is important that automated analysis techniques be developed to help detect problems and assist in software understanding for these systems. There has been considerable research on various analysis techniques for concurrent systems, including static analysis techniques [ADW89, MR90, McD89, SC88, TO80, Tay83b], dynamic analysis techniques [CT91, HL85, RL89, Tai86], and hybrid techniques [Dil88, HK88, YT88, YTFB89].

Data flow analysis is a well-recognized, static analysis technique that has been successfully used on sequential systems to support program optimization, static type checking, and anomaly detection. In addition, there has been considerable research on efficient algorithms for implementing intraprocedural and interprocedural data flow analysis techniques. In this paper we describe a technique for applying data flow analysis to concurrent programs that use the rendezvous model of inter-task communication. Such languages include Ada [Ref83], Distributed Processes [BH78] and CSP [Hoa78]. We also show how the resulting information can be employed to detect anomalies in concurrent programs.

One of the major benefits of applying data flow analysis for anomaly detection is that it can discover interesting classes of problems relatively efficiently. The classes of problems that it can address involve recognizing whether specified patterns of events can occur. Predefined patterns typically include such things as references to undefined variables or two consecutive definitions of some variable without an intervening reference [FO76]. Data flow analysis can also be used to search for general user-defined sequences of events [OO90].

Our approach to data flow analysis of concurrent programs is similar in many ways to the interprocedural data flow analysis described in [All74]. In interprocedural data flow analysis it is desirable to analyze each procedure once and produce summary information that can be used to analyze calls to that procedure. Procedures are analyzed in reverse invocation order so that each procedure is analyzed before the procedures that invoke it. Of course, this method breaks down in the presence of recursion but other methods have been devised for this situation [Bar78, Ros79, HS89]. Our approach to concurrent programs also analyzes each portion of a concurrent program once. This is done by dividing tasks into *fragments*. We introduce a new representation, called a *rendezvous graph*, to determine the order in which task fragments can be analyzed so that each is analyzed before any portion of the program that uses it. Recursion between fragments is not a problem in interfragment analysis since fragments cannot make recursive calls on themselves.

This work builds upon the work of Taylor and Osterweil [TO80]. In the Taylor and Osterweil paper, data flow analysis techniques are applied to concurrent programs written in HAL/S, which supports a more primitive or lower-level inter-task communication model than the rendezvous. Their paper describes the importance of "parceling" a program into components that only have to be analyzed once but does not present a general approach for doing this. We have been able to take advantage of the higher-level communication model supported by the rendezvous to define a general decom-

This work was supported in part by the Office of Naval Research, grant N00014-90-J-1791, and by the National Science Foundation, grant CCR-87-04478, in cooperation with the Defense Advanced Research Projects Agency (ARPA Order No.6104).

position model. Our anomaly detection techniques are refinements to the algorithms given in [TO80].

In our examples we use the syntax of Ada. In Ada, two tasks synchronize their activities and exchange data primarily via rendezvous. A rendezvous occurs when a client task makes an *entry call* to a server task and the server task *accepts* the entry call. The execution of a task making an entry call is suspended until another task accepts that call. Likewise, the execution of a task accepting an entry call is suspended until another task makes a corresponding entry call. During the rendezvous, the execution of the calling task is suspended while the accept statement of the accepting task is executed. At the end of a rendezvous both the client and server task can proceed with their execution. Information may be exchanged via parameters at the start and end of the rendezvous. Thus, the rendezvousing tasks synchronize and exchange information at the start and end of the rendezvous.

For this paper, we make a number of assumptions that are designed to simplify our presentation. We assume that the program under analysis does not have nested tasks or blocks, there are no shared variables between tasks, there are no abort statements, the model for all entry call parameters is in–out, there are no recursive procedures, and that task priorities are not used. For the most part, we can relax these assumptions without difficulty. One possible exception is abort statements, which are problematical in any kind of static analysis. In addition, we further assume that there is exactly one accept statement for each entry of a task. For an entry that has more than one accept statement, there is more than one set of associated summary information. There are two ways to solve this problem. The first is to determine exactly which entry calls are serviced by which accept statements and to analyze each case separately. Unfortunately, this determination may require a substantial amount of effort [Tay83b, Tay83a]. In some situations this extra work may be worthwhile, in which case the techniques in [Tay83a, YT86, LC89] may be applied. The second way to solve this problem is to use a worst case analysis of the summary information for the various accept statements. This will require less effort, but may result in less accurate results.

The next section of this paper describes the task fragments and the rendezvous graph. Sections 3 and 4 present the intrafragment and interfragment data flow analysis techniques, respectively. Section 5 illustrates these techniques with an example. Section 6 describes how the resulting information can be used for anomaly detection. Finally, the conclusion describes some limitations of this approach and directions for future research.

2 The Fragment Model

In this section, we introduce task fragments and how they are dependent on one another. In our analysis, task fragments are analogous in many ways to procedures, serving as the basic units of analysis. Fragments invoke other fragments in much the same way that procedures invoke other procedures. We introduce a *rendezvous graph* to capture these dependencies; it is analogous to the call graph used in interprocedural analysis [All74].

Of central importance is our view of accept statements. To a task that is making an entry call on another task, the entry call can be considered a remote procedure call. The accept statement in the accepting task has the role of the remote procedure body. Information is passed from one task to the other via parameters. The analysis of such a remote procedure call is similar to that for interprocedural analysis. The remote procedure is first analyzed to produce summary information about the effects of the procedure on the parameters. This summary information is then used in the analysis of the calling task.

In an accepting task, an accept statement can be considered as an implicit procedure call. I.e., when the accept statement is encountered, it can be viewed as a procedure body that is being implicitly called. Information is passed from the implicitly calling task to the implicitly called procedure by references to variables visible in the scope of the accept statement. One can think of an implicit parameter for each such variable. The analysis of such an implicit procedure call is similar to that for interprocedural analysis. The implicitly called procedure is first analyzed to produce summary information about the effects of the implicit procedure on the implicit parameters. This summary information is then used in the analysis of the calling procedure.

This view of an accept statement as both a remote procedure for external tasks and an implicit procedure for the task in which the accept statement is embedded allows the accept statement to be analyzed independently and separately from the analysis of the tasks that make entry calls or contain the accept. A single analysis of the accept can be used to produce summary information that can be used for the analysis of all the calling tasks and the accepting task. The analysis can produce summary information for the explicit formal parameters and the implicit parameters at the same time. When summary information is needed for an entry call, the summary information for the explicit parameters is used and the implicit parameters are considered as local variables. When summary information is needed for an implicit call, the summary information for the implicit parameters is used and the explicit parameters are considered as local variables.

The first step in our analysis is to recognize these embedded procedures and determine their order of evaluation. Given a set of tasks T_1 ,..., T_k we do this by dividing each task into *task fragments* representing the calling and called parts of the tasks. The main body of each task is itself a fragment. Each accept statement in a task is a task fragment and is considered separately from the task or accept statement in which it is embedded. Nested accepts result in nested fragments.

Traditionally, data flow analysis is applied to control flow graphs annotated with definition and reference information. For each fragment of a task we therefore construct a Control Flow Graph (CFG). In each CFG, each entry call and each implicit call on an embedded accept statement is represented by a call node. Other nodes in the CFG represent simple statements. Edges represent flow of control between nodes of the graph. The CFG for a fragment F is denoted by CFG(F). We assume, without loss of generality, that each control flow graph contains a single entry and a single exit point. The entry node is referred to as the *start* node and represents the initialization that occurs immediately before the first executable statement of the fragment. The exit node is referred to as the *terminal* node and represents termination activity immediately after all statements that end execution of the fragment. For example, consider the two tasks shown in Figure 1. There are three task fragments in this example, the fragment for $T1$, the fragment for $T2$ and the fragment for the accept statement in $T2$. The CFG's of each of these fragments are shown in Figure 2. The call nodes are marked with a $*$. Node $*8$ represents an entry call and node $*17$ represents an implicit call to the accept statement represented by nodes 17 through 20.

In traditional data flow analysis the order of evaluation of the procedures is determined by examining the call graph for a program. We introduce a *rendezvous graph*, which extends the call graph representation to also capture the order in which fragments interact. A rendezvous graph contains a node for each task fragment and procedure. A directed edge (u, v) in this graph represents the invocation of fragment or procedure v by fragment or procedure u. The rendezvous graph for Example 1 is shown in Figure 3. Since interprocedural analysis is well understood we restrict the remainder of this paper to interfragment analysis.

The analysis presented here is greatly simplified because we assume that the rendezvous graph for a set of tasks will be acyclic. In the absence of recursive procedures, it is our expectation that this will be the normal situation because the presence of a cycle in the rendezvous graph will normally be an indication of a deadlock situation. Deadlock would occur, for example, if a task A makes an entry call on a task B, which then makes an entry call on a task C, which then makes an entry call on any of the tasks A, B, or C. Figure 4 shows the cycle caused by C making an entry call on task A. The last entry call introduces a cycle into the

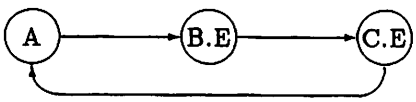

Figure 4: A Rendezvous Graph with a Cycle

rendezvous graph. Since A, B, and C are suspended during their entry calls, they would be unable to accept the last entry call made by C and deadlock occurs. The fact that entry calls cannot be recursive and our restriction that procedures calls are not recursive substantially simplifies the analysis presented in this paper. We have yet to consider the effect of recursive procedures on this analysis.

It is possible to have a rendezvous graph with cycles that does not deadlock. For example, task A and task B may both make entry calls on an entry of task C and that entry of C may be able to make an entry call on A when it is called by B but not when it is called by A. The rendezvous graph contains a cycle, but deadlock does not occur. A similar situation occurs in interprocedural data flow analysis. Techniques, such as node-splitting, exist that can accommodate this type of cycle in a call graph for interprocedural data flow analysis [Hec77] and we expect that similar techniques can be used to accommodate this type of cycle in rendezvous graphs. However, in this paper, we restrict ourselves to acyclic rendezvous graphs.

Because we wish to analyze each fragment of a task only once, we cannot analyze a task fragment until all task fragments that are called by that task fragment have been analyzed. A rendezvous graph provides information about the order in which task fragments can be analyzed. The partial order defined by the rendezvous graph is used to construct a total order on the set of task fragments. In the next section we consider data flow analysis within a fragment and in section 4 we consider data flow analysis between fragments.

3 Intrafragment Data Flow Analysis

In this section we describe data flow analysis for a single task fragment. Consider the CFG for the fragment to be analyzed. Each node n of the CFG is assigned two sets, $G(n)$ and $K(n)$, that summarize local information

```
1   Task body T1 is
2   begin
3       a := 1;
4       while a < 100 loop
5           if a < 0 then
6               a := a + 1;
7           else
8               T2.E(a);
9           end if;
10          b := a
11      end loop;
12  end T1
```

```
13  Task body T2 is
14  begin
15      x := 2;
16      while x < 200 loop
17          accept E(a: in out integer);
18              a := a + x;
19              x := a
20          end E;
21          x := x+1;
22      end loop;
23  end T2
```

Figure 1: Example 1

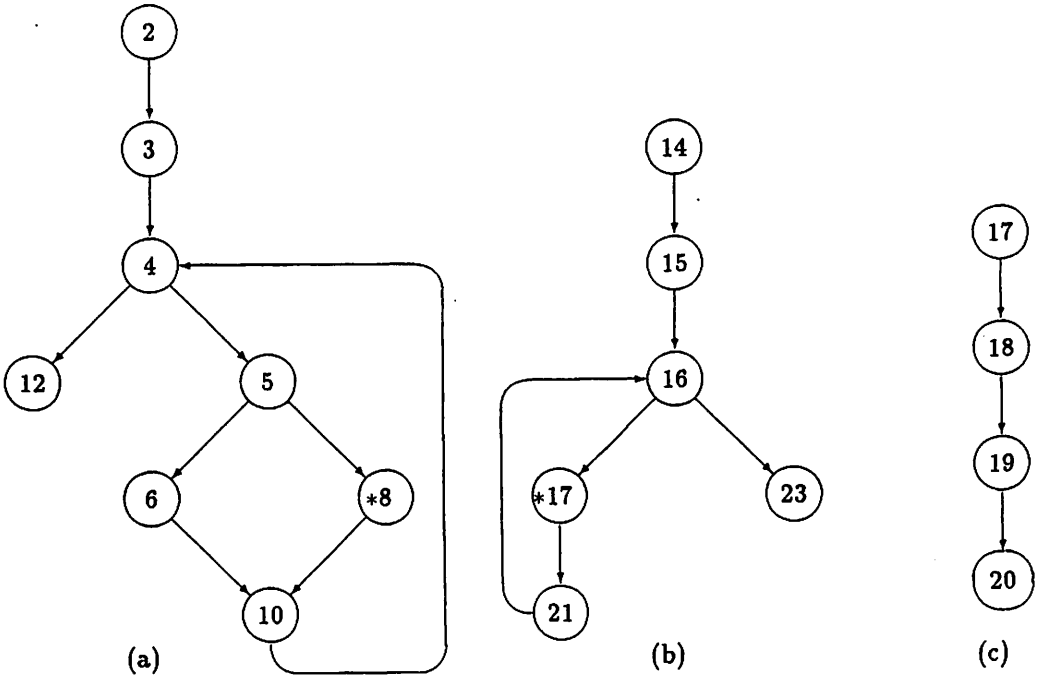

Figure 2: (a) CFG(T1) (b) CFG(T2) (c) CFG(T2.E)

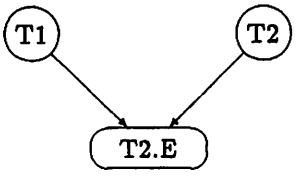

Figure 3: Rendezvous graph for Example 1

at that node. In common terminology these sets are referred to as the *gen* set and the *kill* sets respectively. The definitions of G and K vary depending on the data flow problem to be solved.

Global data flow information is also summarized by attaching sets to each node of the CFG. The two general classes of global data flow information that are considered in this paper are AVAIL and LIVE.

A variable v is in the set $\text{AVAIL}_{G,K}(n)$ if and only if for every path from the start node to n, v is in $G(n')$ for some node n' on the path and is not in $K(n'')$ at any node n'' on the path between n' and n. $\text{AVAIL}_{G,K}(n)$ can be calculated by solving the data flow equations in Figure 5(a) for each node n, other than the start node s. AVAIL for the start node s, which has no predecessors, is defined to be the empty set. $PRED(n)$ is the set of predecessor nodes of n.

A variable v is in the set $\text{LIVE}_{G,K}(n)$ if and only if there exists a path from n to another node n' such that v is in $G(n')$ but is not in $K(n'')$ for any node n'' on the the path. $\text{LIVE}_{G,K}(n)$ can be calculated by solving the following data flow equations shown in Figure 5(b) for each node n, other than the terminal node t. LIVE for the terminal node t, which has no successors, is defined to be the empty set. $SUCC(n)$ is the set of successor nodes of n.

Both AVAIL and LIVE can be calculated using a number of different algorithms that are of complexity no worse than $O(N^2)$, where N is the number of nodes in the CFG [Hec77].

For the start node s and terminal node t of CFG(F), G and K are supposed to represent information before and after a call to F, respectively. When calculating AVAIL, one wants a variable to be in $G(s)$ if and only if the corresponding variable in the calling fragment is in the AVAIL set of the call node. Note that $G(t)$ and $K(t)$ are not used for the calculation of AVAIL and can thus be left undefined. When calculating LIVE, one wants a variable to be in $G(t)$ if and only if the corresponding variable in the calling fragment is in the LIVE set of the call node. Note that $G(s)$ and $K(s)$ are not used for the calculation of LIVE and can thus be left undefined. Since a fragment is analyzed before the fragments that call it, it is necessary to make worst case assumptions about G and K for start and terminal nodes. These worst case assumptions are dependent, of course, on the data flow problem being solved. Note that, for the problems we are considering, K is irrelevant for the start and terminal nodes since the AVAIL(s) and LIVE(t) are defined to be empty, and is therefore defined to be the empty set at these nodes. Section 5 presents a specific data flow problem and describes and justifies initial values of G for that problem.

4 Interfragment Data Flow Analysis

Tasks fragments are analyzed according to a total order determined from the rendezvous graph for the fragments. Suppose that order is given by $F_1, F_2, \ldots, F_n$, where for $j \leq k$, F_j does not call any fragment F_k. Thus, in the analysis of F_j we can make use of the results of the analysis of any fragment F_i that is called by F_j ,where $1 \leq i < j \leq n$.

The analysis of individual fragments was described in Section 3. The purpose of interfragment analysis is to produce summary information about previously analyzed fragments for use in the analysis of fragments that invoke those fragments. Two sets of summary information are kept for each fragment. The first, $G(F_j)$, contains all variables, including all explicit and implicit parameters, for which there is a G of the variable in the fragment that reaches outside the fragment. The second, $K(F_j)$ contains all variables, including all explicit and implicit parameters, that are killed by the call to the fragment. Definitions of $G(F_j)$ and $K(F_j)$ differ for AVAIL and LIVE and will be distinguished by subscripts AVAIL and LIVE. G_{AVAIL} and K_{AVAIL} are described in section 4.1. G_{LIVE} and K_{LIVE} are described in section 4.2. Once this summary information is produced it can be used to calculate $G(c)$ and $K(c)$ for each call node c of other fragments that call F_j. This is described in section 4.3.

4.1 AVAIL

The interfragment AVAIL analysis calculates the summary information $G_{\text{AVAIL}}(F_j)$ and $K_{\text{AVAIL}}(F_j)$ for each fragment F_j.

$G_{\text{AVAIL}}(F_j)$ contains all variables, including all explicit and implicit parameters, such that for every path through F_j there is a G of the variable without a subsequent K of the variable. To calculate $G_{\text{AVAIL}}(F_j)$ we define a pessimistic version of G called G_{pes}. G_{pes} is identical to G except that for the start node s, $G_{pes}(s) = \emptyset$. For other nodes n, $G_{pes}(n) = G(n)$. For the terminal node $G_{pes}(t)$ is undefined. With these definitions, $G_{\text{AVAIL}}(F_j)$ is $\text{AVAIL}_{G_{pes},K}(t)$.

$K_{\text{AVAIL}}(F_j)$ contains all variables that are killed by the fragment, i.e., the variables such that if they are in the AVAIL set at the start of the fragment then they are not in the AVAIL set at the end of the fragment. To calculate $K_{\text{AVAIL}}(F_j)$ we define an optimistic version of G called G_{opt}. G_{opt} is identical to G except that for the start node s, $G_{opt}(s) = \{\text{all variables in } F_j\}$. For other nodes n, $G_{opt}(n) = G(n)$. For the terminal node $G_{opt}(t)$ is undefined. With these definitions, $K_{\text{AVAIL}}(F_j)$ is $\overline{\text{AVAIL}_{G_{opt},K}(t)}$.

$$\text{AVAIL}_{G,K}(n) = \bigcap_{n_i \in PRED(n)} \left(G(n_i) \cup (\text{AVAIL}_{G,K}(n_i) \cap \overline{K(n_i)}) \right)$$

(a)

$$\text{LIVE}_{G,K}(n) = \bigcup_{n_i \in SUCC(n)} \left(G(n_i) \cup (\text{LIVE}_{G,K}(n_i) \cap \overline{K(n_i)}) \right)$$

(b)

Figure 5: Data Flow Equations for (a) AVAIL and (b) LIVE

The complete interfragment data flow analysis for AVAIL is as follows.

1. Calculate $\text{AVAIL}_{G_{opt},K}(i)$ for each node i of the $\text{CFG}(F_j)$ using the intrafragment algorithm.
2. Calculate $\text{AVAIL}_{G_{pes},K}(i)$ for each node i of the $\text{CFG}(F_j)$ using the intrafragment algorithm.
3. Let
$$G_{\text{AVAIL}}(F_j) = \text{AVAIL}_{G_{pes},K}(t)$$
where t is the terminal node of $\text{CFG}(F_j)$.
4. Let
$$K_{\text{AVAIL}}(F_j) = \overline{\text{AVAIL}_{G_{opt},K}(t)}$$
where t is the terminal node.

4.2 LIVE

The interfragment LIVE analysis calculates the summary information $G_{\text{LIVE}}(F_j)$ and $K_{\text{LIVE}}(F_j)$ for each fragment F_j.

$G_{\text{LIVE}}(F_j)$ contains all variables, again including all explicit and implicit parameters, such that for some path through F_j there is a G of the variable before there is a K of the variable. To calculate $G_{\text{LIVE}}(F_j)$ we define a pessimistic version of G called G_{pes}. G_{pes} is the same as G except that for the terminal node t, $G_{pes}(t) = \emptyset$. For other nodes n, $G_{pes}(n) = G(n)$. For the start node $G_{pes}(s)$ is undefined. With these definitions, $G_{\text{LIVE}}(F_j)$ is $\text{LIVE}_{G_{pes},K}(s)$.

$K_{\text{LIVE}}(F_j)$ contains all variables that are killed by the fragment, i.e., the variables such that if they are in the LIVE set at the end of the fragment then they not in the LIVE set at the start of the fragment. To calculate $K_{\text{LIVE}}(F_j)$ we define an optimistic version of G called G_{opt}. G_{opt} is the same as G except that for the terminal node t, $G_{opt}(t) = \{\text{all variables in } F_j\}$. For other nodes n, $G_{opt}(n) = G(n)$. For the start node $G_{opt}(s)$ is undefined. With these definitions, $K_{\text{LIVE}}(F_j)$ is $\overline{\text{LIVE}_{G_{opt},K}(s)}$.

The complete interfragment data flow analysis for LIVE is as follows.

1. Calculate $\text{LIVE}_{G_{opt},K}(i)$ for each node i of the $\text{CFG}(F_j)$ using the intrafragment algorithm.
2. Calculate $\text{LIVE}_{G_{pes},K}(i)$ for each node i of the $\text{CFG}(F_j)$ using the intrafragment algorithm.
3. Let
$$G_{\text{LIVE}}(F_j) = \text{LIVE}_{G_{pes},K}(s)$$
where s is the start node of $\text{CFG}(F_j)$.
4. Calculate
$$K_{\text{LIVE}}(F_j) = \overline{\text{LIVE}_{G_{opt},K}(s)}$$
where s is the start node.

4.3 Using Summary Information

Once the interfragment data flow analysis has produced the necessary summary information for a fragment F_j, the summary information can be used in the inter- and intrafragment analysis of succeeding fragments. If c is a call node in a fragment that calls fragment F_j then $G(F_j)$ and $K(F_j)$ are used to calculate $G(c)$ and $K(c)$ as follows. If c is a call node for an entry call then $G(c)$ is the set of actual parameters that correspond to formal parameters in $G(F_j)$ and $K(c)$ is the set of actual parameters that correspond to formal parameters in $K(F_j)$. If c is a call node for an implicit procedure call then $G(c)$ is the set of implicit parameters in $G(F_j)$ and $K(c)$ is the set of implicit parameters in $K(F_j)$.

The interfragment analysis and the intrafragment analysis must be applied in the specified order. Note that for most data flow problems either $G_{opt} = G$ or $G_{pes} = G$, in which case the intrafragment analysis is redundant. Even if this is not the case, $\text{AVAIL}_{G,K}$ is

easily calculated from $\text{AVAIL}_{G_{opt},K}$ and $\text{AVAIL}_{G_{pes},K}$ and $\text{LIVE}_{G,K}$ is easily calculated from $\text{LIVE}_{G_{opt},K}$ and $\text{LIVE}_{G_{pes},K}$, although we do not show it here.

5 A Simple Example

This section demonstrates the application of the formulas of the previous section. The tasks for this example are shown in Figure 6. The rendezvous graph for this example is shown in Figure 7. There are six task fragments in this example. The fragment S1.ADDONE, representing the accept statement for the entry ADDONE of task S1, is called by fragments T1 and S2.ADDTWO and implicitly called by fragment S1. The fragment S2.ADDTWO, representing the accept statement for the entry ADDTWO of task S2, is called by fragment T2 and implicitly called by fragment S2. The control flow graphs of each of these fragments are given in Figure 8.

Each node of each CFG is numbered with the statement number that it represents. In addition, each node is labeled with two sets, *DEF* and *REF*. The DEF set is the set of variables defined at that node and the REF set is the set of variables referenced at that node. The DEF and REF for call nodes sets are initially unknown but are calculated later after analysis of the called fragment is completed. The DEF and REF sets are defined to be the emptyset at the start node and the terminal node of each CFG.

5.1 Calculating AVAIL

In the following we use the AVAIL data flow equations to find the set of variables that are always uninitialized at each point of the program. This can be done by defining G and K appropriately. Since is it is not known at the time a fragment is analyzed which variables are always uninitialized at the start of a fragment, the conservative approach is to assume that all variables in the fragment are uninitialized. The analysis then tells us at which points in the fragment a variable is uninitialized *if* it is uninitialized at the start of the fragment. After the analysis of all fragments is complete this information can be used to determine which variables are actually uninitialized.

We define G and K as follows. For the start node s, of each CFG, we define $G(s)$ to be the set of all variables in the fragment and $K(s)$ to be the emptyset. For the terminal node t, $G(t)$ and $K(t)$ are undefined since they are not used in the calculation of AVAIL. For non-call nodes n, we define $G(n) = \emptyset$ and $K(n) = \text{DEF}(n)$. For a call node c, $G(c)$ and $K(c)$ are initially unknown because they can only be calculated after the interfragment analysis of the called fragment.

From the rendezvous graph in Figure 7 it can be seen that the fragments can be analyzed in the order S1.ADDONE, T1, S1, S2.ADDTWO, T2, S2. The results of the intrafragment analysis for each fragment are shown in Tables 1–6. For the interfragment analysis, note that for this data flow problem it happens that $G_{opt} = G$, so $\text{AVAIL}_{G_{opt},K} = \text{AVAIL}_{G,K}$. Also, $\text{AVAIL}_{G_{pes},K}$ for each node of each CFG is the empty set.

The first fragment to be analyzed is S1.ADDONE. G, K, and AVAIL are shown in Table 1. This CFG contains no call nodes.

The interfragment analysis of S1.ADDONE gives the following summary information for this fragment.

$$\begin{aligned} G_{\text{AVAIL}}(\text{S1.ADDONE}) &= \text{AVAIL}_{G_{pes},K}(31) \\ &= \emptyset \end{aligned}$$

$$\begin{aligned} K_{\text{AVAIL}}(\text{S1.ADDONE}) &= \overline{\text{AVAIL}_{G_{opt},K}(31)} \\ &= \{a, count1\} \end{aligned}$$

Before the intrafragment analysis of the next fragment T1, can be carried out we must first determine G and K for the call node 7. These sets consist of the actual parameters of the entry call for which the corresponding formal parameters are in G_{AVAIL}(S1.ADDONE) and K_{AVAIL}(S1.ADDONE) respectively. Since $G_{\text{AVAIL}}(\text{S1.ADDONE}) = \emptyset$ it contains no formal parameters so $G(7) = \emptyset$. On the other hand, K_{AVAIL}(S1.ADDONE) contains the formal parameter a, so $K(7)$ contains the corresponding actual parameter i. Since, in this example, the first parameter is defined in the accept statement, this is consistent with our definition of kill when determining uninitialized variables.

G, K, and AVAIL for T1 are shown in Table 2.

The analysis of S1 is similar to that for T1, except that node 26 is an implicit call on S1.ADDONE. Thus, $G(26)$ and $K(26)$ consist of the implicit parameters of G_{AVAIL}(S1.ADDONE) and K_{AVAIL}(S1.ADDONE) respectively. Since $G_{\text{AVAIL}}(\text{S1.ADDONE}) = \emptyset$, it contains no implicit parameters, so $G(26) = \emptyset$. K_{AVAIL}(S1.ADDONE) contains the implicit parameter $count1$, so $K(26) = \{count1\}$.

G, K, and AVAIL for S1 are shown in Table 3.

In S2.ADDTWO nodes 41 and 42 are entry calls on S1.ADDONE. Since $G_{\text{AVAIL}}(\text{S1.ADDONE}) = \emptyset$ it contains no formal parameters so $G(41) = G(42) = \emptyset$. K_{AVAIL}(S1.ADDONE) contains the formal parameter a, so $K(41)$ and $K(42)$ contain the corresponding actual parameter b. G, K, and AVAIL for S2.ADDTWO are shown in Table 4.

The interfragment analysis of S2.ADDTWO gives the following summary information for the fragment.

```
1   task body T1 is
2       i, n, c : integer;
3   begin
4       i := 0;
5       input(n);
6       while i < n do
7           S1.ADDONE(i,c);
8       end while;
9       output(i);
10  end T1;

11  task body T2 is
12      j, m, d : integer;
13  begin
14      j := 0;
15      input(m);
16      while j < m do
17          S2.ADDTWO(j,d);
18      end while;
19      output(j);
20  end T2;
```

```
21  task S1 body is
22      count1,max1 : integer;
23  begin
24      count1 := 0;
25      while count1 < max1 do
26          accept ADDONE(a,e : in out integer) do
27              a := a + 1;
28              if e / = 0 then
29                  count1 := count1 + e
30              end if;
31          end ADDONE;
32      end while;
33  end S1;

34  task S2 body is
35      count2,max2,g : integer;
36  begin
37      count2 := 0;
38      while count2 < max2 do
39          accept ADDTWO(b,f : in out integer) do
40              g := f / 2;
41              S1.ADDONE(b,g);
42              S1.ADDONE(b,g);
43              if f / = 0 then
44                  count2 := count2 + f
45              end if;
46          end ADDTWO;
47      end while;
48  end S2;
```

Figure 6: Example 2

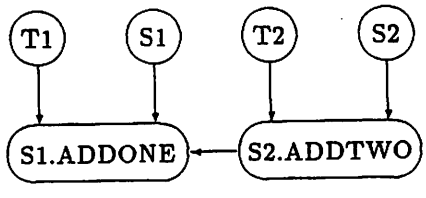

Figure 7: Rendezvous graph for Example 2

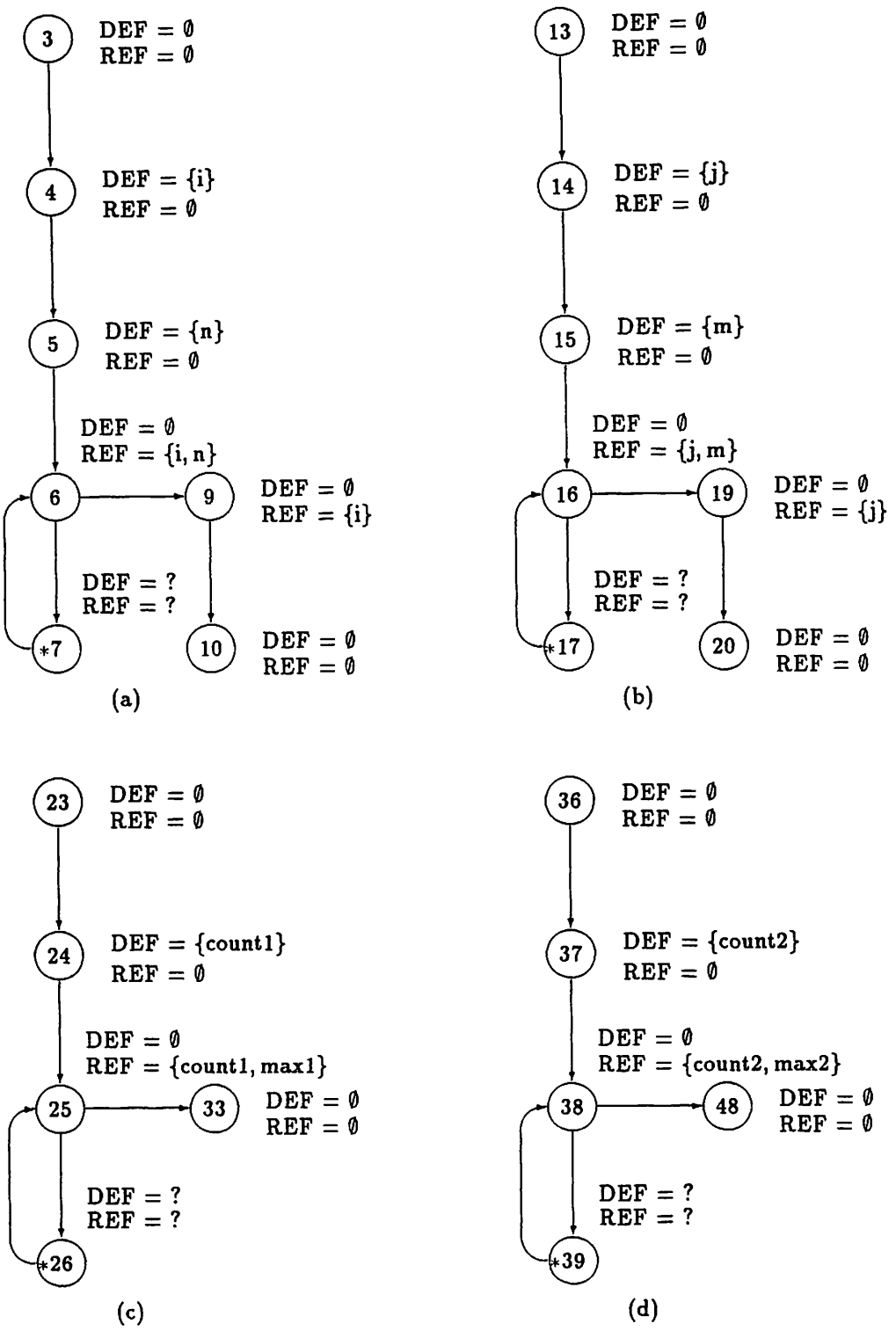

Figure 8: (a) CFG(T1) (b) CFG(T2) (c) CFG(S1) (d) CFG(S2)

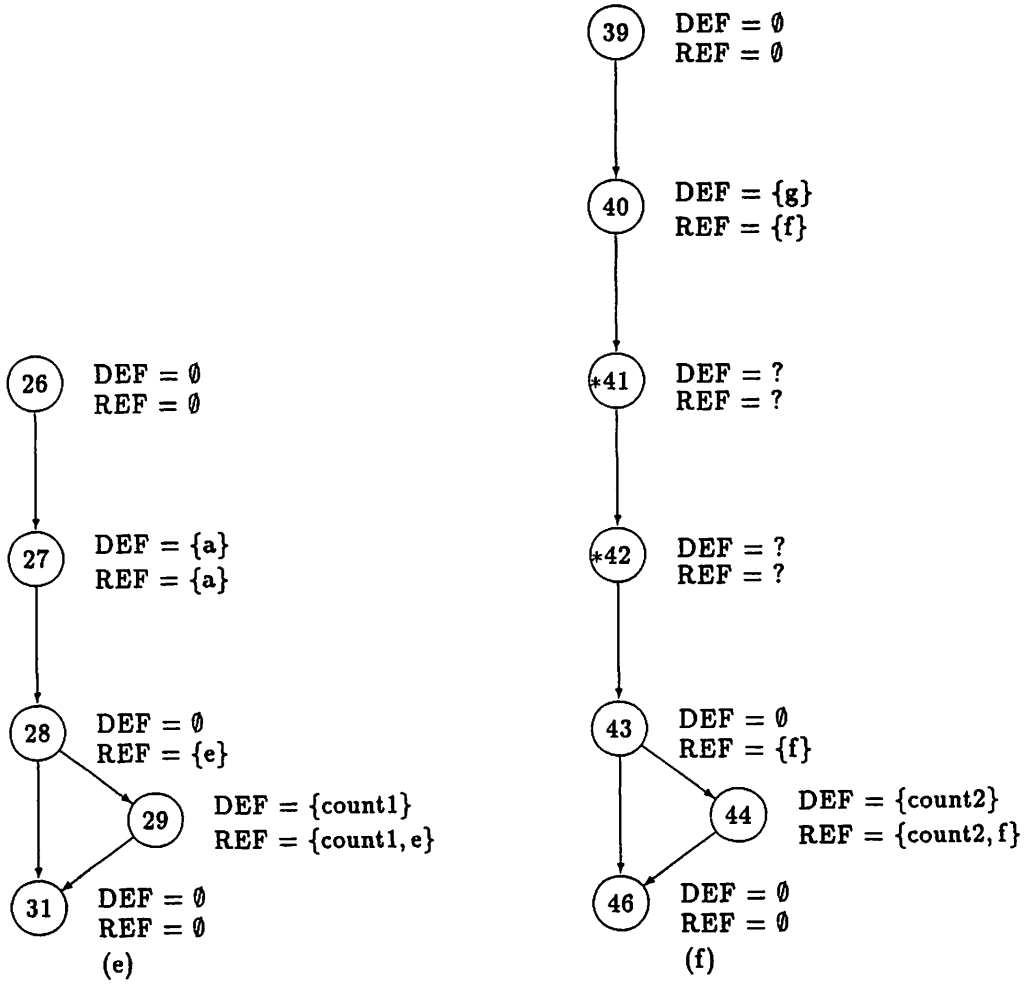

Figure 8: (e) CFG(S1.ADDONE) (f) CFG(S2.ADDTWO)

n	$G(n)$	$K(n)$	$\text{AVAIL}_{G,K}(n)$	$REF(n)$
26	a,e,count1,max1	∅	∅	∅
27	∅	a	a,e,count1,max1	a
28	∅	∅	e,count1,max1	e
29	∅	count1	e,count1,max1	count1,e
31	?	?	e,max1	∅

Table 1: AVAIL for S1.ADDONE

n	$G(n)$	$K(n)$	$\text{AVAIL}_{G,K}(n)$	$REF(n)$
3	i,n,c	$\emptyset$	$\emptyset$	$\emptyset$
4	$\emptyset$	i	i,n,c	$\emptyset$
5	$\emptyset$	n	n,c	$\emptyset$
6	$\emptyset$	$\emptyset$	c	i,n
*7	$\emptyset$	i	c	i,c
9	$\emptyset$	$\emptyset$	c	i
10	?	?	c	$\emptyset$

Table 2: AVAIL for T1

n	$G(n)$	$K(n)$	$\text{AVAIL}_{G,K}(n)$	$REF(n)$
23	count1,max1	$\emptyset$	$\emptyset$	$\emptyset$
24	$\emptyset$	count1	count1,max1	$\emptyset$
25	$\emptyset$	$\emptyset$	max1	count1,max1
*26	$\emptyset$	count1	max1	count1
33	?	?	max1	$\emptyset$

Table 3: AVAIL for S1

n	$G(n)$	$K(n)$	$\text{AVAIL}_{G,K}(n)$	$REF(n)$
39	count2,g,max2,b,f	$\emptyset$	$\emptyset$	$\emptyset$
40	$\emptyset$	g	count2,g,max2,b,f	f
*41	$\emptyset$	b	count2,max2,b,f	b,g
*42	$\emptyset$	b	count2,max2,f	b,g
43	$\emptyset$	$\emptyset$	count2,max2,f	f
44	$\emptyset$	count2	count2,max2,f	count2,f
46	?	?	max2,f	$\emptyset$

Table 4: AVAIL for S2.ADDTWO

$$G_{\text{AVAIL}}(\text{S2.ADDTWO}) = \text{AVAIL}_{G_{pes},K}(46) = \emptyset$$

$$K_{\text{AVAIL}}(\text{S2.ADDTWO}) = \overline{\text{AVAIL}_{G_{opt},K}(46)} = \{count2, g, b\}$$

To analyze T2 we first determine G and K for the call node 17. Since $G_{\text{AVAIL}}(\text{S2.ADDTWO}) = \emptyset$ it contains no formal parameters so $G(17) = \emptyset$. $K_{\text{AVAIL}}(\text{S2.ADDTWO})$ contains the formal parameter b, so $K(17)$ contains the corresponding actual parameter j.

G, K, and AVAIL for T2 are shown in Table 5.

In S2 node 39 is an implicit call on S2.ADDTWO. Since $G_{\text{AVAIL}}(\text{S2.ADDTWO}) = \emptyset$, it contains no implicit parameters, so $G(39) = \emptyset$. $K_{\text{AVAIL}}(\text{S2.ADDTWO})$ contains the implicit parameters $count2$ and g, so $K(39) = \{count2, g\}$.

G, K, and AVAIL for S2 are shown in Table 6.

6 Anomaly Detection

Data flow analysis has a wide variety of applications in compiler optimization, anomaly detection, and other areas. Our interest is in anomaly detection, although the techniques described in this paper are applicable in other areas as well. A number of interesting classes of anomalies can be detected in a program by making a suitable choice for G and K, applying AVAIL or LIVE as appropriate, and then analyzing the results using a suitable anomaly condition. Examples are: uninitialized variables, where there is a reference to a variable before a definition of that variable; dead definitions, where a definition is followed by another definition or by the end of the scope of the definition without an intervening reference; and definitions that do not influence the output, where there is no def-use chain from a definition to an output. In each case we can distinguish between *always* occurrences of the anomaly, where the anomaly occurs on every path and therefore always occurs, and *sometimes* occurrences, where the anomaly occurs on some but not all paths and therefore sometimes occurs during some executions but not during others. Furthermore, one must be aware that, because this is a static analysis technique, further analysis may be necessary to determine if the reported anomalies actually occur in the program or are spurious. Spurious results are discussed further in the next section.

Using the previous example from Section 5, we first demonstrate anomaly detection for an always uninitialized variable condition. For this case, an always uninitialized variable anomaly occurs for a variable v at any node n of a task where $v \in \text{AVAIL}_{G,K}(n)$ and $v \in \text{REF}(n)$.

Anomalies are reported only after the analysis of top level fragments (i.e., those fragments not called by any other fragment). Anomalies detected during the analysis of lower level fragments (i.e., called by other fragments) are summarized and this summary information is used during the analysis of higher level fragments.

For this case, anomaly analysis proceeds in the same order as the calculation of AVAIL. Starting with fragment S1.ADDONE we see that $a \in \text{REF}(27)$ and $a \in \text{AVAIL}_{G,K}(27)$. Likewise there is a potential anomaly for e at node 28 and 29 and $count1$ at 29. If any of these variables is undefined at the start of this fragment then there is an undefined reference anomaly in this fragment. But it is not known at this point if these variables are undefined at the start of the fragment, so we produce a summary set $\text{REF}(\text{S1.ADDONE}) = \{a, e, count1\}$ that will be used in the analysis of fragments that call S1.ADDONE.

For fragment T1, the REF set is given in Figure 8 for all nodes except the call node 7. REF(7) is determined by examining REF(S1.ADDONE). The formal parameters a and e are both in this set so the corresponding actual parameters i and c are in REF(7). With this information we see that the REF to c at node 7 is in fact a reference to an uninitialized variable. Since T1 is a top level fragment we report this anomaly.

The analysis of the remaining fragments is similar. For fragment S1, $\text{REF}(26) = \{count1\}$ and comparison of REF and AVAIL show that there is an uninitialized reference to $max1$ at node 25. For fragment S2.ADDTWO, $\text{REF}(41) = \text{REF}(42) = \{b, g\}$ so $\text{REF}(\text{S2.ADDTWO}) = \{b, f, count2\}$. For fragment T2, $\text{REF}(17) = \{j, d\}$ so the REF to d at node 17 is a reference to an uninitialized variable. For fragment S2, $\text{REF}(39) = \{count2\}$ so the REF to $max2$ at node 38 is an uninitialized reference.

Now, suppose we wished to find occurrences of uninitialized variables that *sometimes* occur. We can use AVAIL to solve this problem as well by defining $G(s) = \emptyset$, $K(s) = \emptyset$, $G(n) = \text{DEF}(n)$ and $K(n) = \emptyset$, where s is a start node of a fragment and n is a non-start node. We report a sometimes uninitialized variable anomaly for a variable v at any node n of a task where $v \notin \text{AVAIL}_{G,K}(n)$ and $v \in \text{REF}(n)$. The REF set of a call node is calculated in the same way as in the previous example.

We could also solve the always and sometimes uninitialized variable problem using LIVE. This is left as an exercise for the reader.

n	$G(n)$	$K(n)$	$\text{AVAIL}_{G,K}(n)$	$REF(n)$
13	j,m,d	∅	∅	∅
14	∅	j	j,m,d	∅
15	∅	m	m,d	∅
16	∅	∅	d	j,m
*17	∅	j	d	j,d
19	∅	∅	d	j
20	?	?	d	∅

Table 5: AVAIL for T2

n	$G(n)$	$K(n)$	$\text{AVAIL}_{G,K}(n)$	$REF(n)$
36	count2,g,max2	∅	∅	∅
37	∅	count2	count2,g,max2	∅
38	∅	∅	max2	count2,max2
*39	∅	count2,g	max2	count2
48	?	?	max2	∅

Table 6: AVAIL for S2

7 Conclusion

In this paper we have demonstrated how data flow analysis techniques can be applied to concurrent programs that use a rendezvous model of inter-task communication. Our approach is based on dividing the system into fragments, where each fragment can be analyzed separately. The summary information about each fragment is then used whenever that fragment is invoked. Two kinds of invocations are distinguished and each must be treated differently. A rendezvous graph is also defined for determining the order in which fragments can be analyzed. Since the overall approach is very similar to interprocedural data flow analysis, efficient algorithms exist for computing all the required information.

The major drawback of data flow analysis is that it computes conservative estimates about the behavior of a system and thus sometimes reports spurious anomalies. There are two major reasons for these spurious results. The first has to do with complex data structures, where the precise identity of a component of an object is computed at run time; for example, via an index or pointer. In such cases, static analysis can not always identify the precise component and therefore assumes that all components have been involved.

The second reason is referred to as the *infeasible path* problem. Data flow analysis usually involves analyzing a control flow graph representation of a program. A control flow graph represents all paths through a program, where a path is a syntactically legal execution sequence. Because of the semantics of the program, however, not all of such paths may be valid. For example, a FOR LOOP that has a loop iteration variable that goes from one to ten with no additional exits from the loop can not be executed other than ten times. A static analysis of this graph, however, would consider the fall through case, where the loop is never executed, a viable path.

Dealing with complex data structures and infeasible paths are inherent limitations of data flow analysis. One approach to dealing with these problems is to carefully distinguish when a detected anomaly is definitely an indication of a erroneous pattern of behavior from when it might be a spurious result. The latter is usually reported as a warning, and the former as an error. When a warning is reported it is expected that additional analysis will be required to determine if a problem actually exists.

Thus, data flow analysis is often seen as a two phase process. During the first phase, data flow analysis algorithms are applied and errors and warnings are reported. During the second phase, additional analysis is attempted to determine if the execution pattern associated with a warning report is feasible. Of course, in general, the path feasibility problem is not decidable. In practice, it can frequently be determined relatively easily by a developer examining the program.

For concurrent systems we propose a three phase approach for data flow analysis. The two phases associated with sequential data flow analysis remain, but a new phase is needed between them to determine if the task interaction specified by a data flow anomaly is a feasible interaction. For concurrent systems, infeasi-

ble interactions can be due to deadlock, livelock, or because interactions cannot execute concurrently. This is an additional complication to the path feasibility problem. Usually such analysis requires construction of a reachability graph, which can be exponential in terms of the number of tasks. While in the worse case a search through the reachability graph might be exponential, we believe techniques can be developed to support, at least experimentally, efficient directed searches that should not require the construction of the entire reachability graph [YT86]. A benefit of our approach is that reachability is not even an issue unless the data flow analysis phase has indicated a reason for concern about a particular task interaction.

At this point in time we have not implemented the data flow analysis techniques so we do not have any experience with the number and kinds of anomalies that are actually discovered. We believe that conducting such experiments will provide valuable insight and intend to pursue this in the future.

References

[ADW89] G.S. Avrunin, L.K. Dillon, and J.C. Wileden. Experiments With Automated Constrained Expression Analysis of Concurrent Software Systems. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification (R.A. Kemmerer, ed.)*, pages 124–130, December 1989. Appeared as *Software Engineering Notes*, 14(8).

[All74] F.E. Allen. *Information Processing 74*, chapter Intraprocedural Data Flow Analysis, pages 398–402. North Holland Pub. Co., Amsterdam, 1974.

[Bar78] J.M. Barth. A Practical Interprocedural Data Flow Analysis Algorithm. *Communications of the ACM*, 21(9):724–736, Sept. 1978.

[BH78] P. Brinch Hanson. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[CT91] R.H. Carver and K.-C. Tai. Replay and Testing for Concurrent Programs. *IEEE Software*, 8(2):66–74, March 1991.

[Dil88] L. K. Dillon. Symbolic Execution-Based Verification of Ada Tasking Programs. In *Third Int. Conference on Ada Applications and Environments*, pages 3–13, May 1988.

[FO76] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *Computing Surveys*, 8(3):305–330, September 1976.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, 1977.

[HK88] L. J. Harrison and R. A. Kemmerer. An Interleaving Symbolic Execution Approach for the Formal Verification of Ada Programs with Tasking. In *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, pages 3–13, May 1988.

[HL85] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HS89] M.J. Harrold and M.L. Soffa. Interprocedural Data Flow Testing. In *Proceedings of the SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification, Key West, Florida*, pages 158–167, Dec. 1989.

[LC89] D.L. Long and L.A. Clarke. Task Interaction Graphs For Concurrency Analysis. In *11th International Conference on Software Engineering*, pages 44–52, May 1989. Pittsburgh, Pennsylvania.

[McD89] C.E. McDowell. A Practical Algorithm for Static Analysis of Parallel Programs. *Journal of Parallel and Distributed Computing*, 6:515–536, 1989.

[MR90] S.P. Masticola and B.G. Ryder. Static Infinite Wait Anomaly Detection in Polynomial Time. Tech Report LCSR-TR-141, Laboratory for Computer Science Research, Hill Center for the Mathematical Sciences, Busch Campus, Rutgers University, New Brunswick, NJ 08903, January 1990.

[OO90] K.M. Olender and L.J. Osterweil. Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.

[Ref83] Reference Manual for the Ada Programming Language, Washington DC. *United States Department of Defense*, January 1983.

[RL89] D. S. Rosenblum and D. C. Luckham. Testing the correctness of tasking supervisors with TSL specifications. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification (R. A. Kemmerer, ed.)*, pages 187–196, 1989. Published as *Software Engineering Notes*, 14(8).

[Ros79] B.K. Rosen. Data Flow Analysis for Procedure Languages. *Journal of the ACM*, 26(2):322–344, April 1979.

[SC88] S. M. Shatz and W. K. Cheng. A Petri Net Framework for Automated Static Analysis of Ada Tasking Behavior. *Journal of Systems and Software*, 8(5):343–359, 1988.

[Tai86] K.-C. Tai. A graphical notation for describing executions of concurrent Ada programs. *Ada Letters*, 6:94–103, January–February 1986.

[Tay83a] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] R.N. Taylor. A General-Purpose Algorithm For Analyzing Concurrent Programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[TO80] R.N. Taylor and L.J. Osterweil. Anomaly Detection in Concurrent Software by Static Data Flow Analysis. *IEEE Transactions on Software Engineering*, SE-6(3):265–277, May 1980.

[YT86] M. Young and R.N. Taylor. Combining Static Concurrency Analysis With Symbolic Execution. In *Proceedings of the Workshop on Software Testing*, pages 170–178, July 1986. Published by IEEE Computer Society Press.

[YT88] M. Young and R.N. Taylor. Combining Static Concurrency Analysis and Symbolic Execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.

[YTFB89] M. Young, R. N. Taylor, K. Forester, and D. Brodbeck. Integrated concurrency analysis in a software development environment. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification, (R. A. Kemmerer, ed.)*, pages 200–209, 1989. Published as *Software Engineering Notes*, 14(8).