TASK INTERACTION GRAPHS FOR CONCURRENCY ANALYSIS

Douglas L. Long
Department of Computer Science
Wellesley College
Wellesley, Massachusetts 02181

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

ABSTRACT

A representation for concurrent programs, called *task interaction graphs*, is presented. Task interaction graphs divide a program into maximal sequential regions connected by edges representing task interactions. This representation is illustrated and it is shown how it can be used to create concurrency graph representations that are much smaller than those created from control flow graph representations. Both task interaction graphs and their corresponding concurrency graphs facilitate analysis of concurrent programs. Some analyses and optimizations on these representations are also described.

1 INTRODUCTION

Dealing with concurrent systems poses many interesting and challenging problems. Clearly, it is much harder for developers to reason about concurrent behavior than sequential behavior, and thus it is likely that more errors will be introduced into these systems. Because of this added complexity as well as the difficulties with reproducing results and simulating realistic scenarios, it is important that analysis techniques be developed to evaluate concurrent systems [Avru85,Bris79,Helm85,Morg87] [Shat88,Tai85,Tayl80,Tayl83a,Tayl83b]. In this paper we present a representation for concurrent systems, called a *task interaction graph*, that facilitates such analysis.

Our representation is an extension and improvement upon the work of Taylor [Tayl83a,Tayl83b]. Using a reduced flow graph representation of each task in a system, Taylor defines a *concurrency graph* that models the behavior of the total system. Since concurrency graphs capture all the possible states of a concurrent system, they provide an interesting model upon which to base a number of different analyses [Tayl83b,Youn86]. Unfortunately the number of states in a concurrency graph can be very large, thereby limiting the programs that can be analyzed and the types of analysis that can be performed.

We have been developing a model of interacting tasks that considerably reduces the number of states in concurrency graph representations. We call this representation a *Task Interaction Concurrency Graph*(TICG), since it is derived from a *Task Interaction Graph*(TIG) instead of from a control flow graph representation. Using our model, we compared the resulting representations for some of the common concurrency examples that appear in the literature. For these examples the number of states were reduced substantially, usually by well over fifty percent. Such a reduction will have a major impact on the kinds of analysis that can be applied and on the kinds of programs that can be analyzed. Moreover, this reduction comes with no loss of information. In fact, the resulting representation appears to be even more amenable to analysis. This is because a TIG divides a task, not based on control flow information, but based on task interactions, the real focus of our concern. The other benefit of this choice of representation is that the nodes in a TIG identify maximal sequential regions in the task. Thus, sequential analysis techniques could be applied to these regions and then inter-task analysis techniques developed to evaluate the impact of task interactions, in much the same way that inter-procedural analysis is carried out for program optimization.

In this paper we describe task interaction graphs and some of the ways they can be used as the basis for analysis of concurrent systems. The next section describes this graph representation and presents two examples. The third section describes how to create a TICG from a TIG and, using a simple example, illustrates this. Section 4 describes how the model can be extended to nested tasks, some of the analysis that can be done based on this model, and some refinements to the model that reduce the size of the TIG still further. The conclusion summarizes the benefits of this representation and discusses directions of future work. The TICG and TIG models have been designed to capture the rendezvous-like interprocess communication mechanism found in languages like Ada [Ada83], Distributed Processes [Brin78], and CSP [Hoar78]. For this presentation we use an Ada-like language to illustrate the approach.

2 Task Interaction Graphs

This section shows how a task interaction graph represents a task as a set of regions and a set of interactions between regions. A formal definition of a TIG is given and two examples are shown and discussed.

There are four restrictions on the kinds of tasking programs we consider. The first two are for inherent problems with any static analysis method and the last two are to simplify the discussion. First, arrays of tasks are disallowed. In general, static analysis can not distinguish between different members of a compound object, such as different elements of an array. Second, it is

This work was supported in part by National Science Foundation grant CCR-87-04478 in cooperation with the Defense Advanced Research Projects Agency (ARPA Order No.6104) and by Office of Naval Research grant N00014-88-K-0025.

©1989 ACM 0270-5257/89/0500/0044$00.75

Recommended by: William Riddle

assumed that at most a fixed number of tasks are active simultaneously. This restriction is needed since certain kinds of dynamic task creation can make static analysis intractable. Third, tasks do not share variables and, fourth, all tasks are activated at the same time and terminate at the same time. Although not shown in this paper, the third restriction can be relaxed with appropriate modifications to the model. The fourth restriction is relaxed in Section 4.1.

Formally, a task interaction graph is a tuple (N, E, S, T, L, C), where N is the set of nodes, E is the set of edges, S is the start node, T is a set of terminal nodes, L is a function that assigns a label to each edge, and C is a function that assigns pseudocode to each node. Each node of this graph represents a task region and each edge represents a task interaction. The start node represents the region where the task starts execution. The terminal nodes represent regions where the task may finish execution.

Each node in a TIG represents a different region of the task and has associated with it an explicit representation of the code for that region, referred to as *pseudocode*. In this paper, the pseudocode for regions consists of the same Ada-like language that is used to represent tasks with the addition of two kinds of *transition pseudostatements*, one that marks the entry to a region and the other to mark the exit(s). Note that regions may overlap, i.e., portions of the pseudocode describing one region may be duplicated in the pseudocode describing another region.

Each edge in a TIG represents a task interaction, indicating a transition from one region to another. The boundary between these regions is represented by the two transition pseudostatements – one in each of the two regions connected by that interaction. An EXIT(*interaction,next*) is used in the first region to indicate a place where that region may be exited, where *interaction* specifies the type of task interaction that causes the transition and *next* specifies the region that is entered after the interaction. The pseudostatement ENTER(*interaction*) is used in the second region to indicate the place where that region may be entered, where again *interaction* specifies the type of interaction. Thus in the representation of the TIG, for each edge between two nodes there is a transition pseudostatement in one node representing the head of the edge and a transition pseudostatement in the other node representing the tail of the edge.

The result of the above discussion is that each entry call and each accept statement is modeled using two interactions that divide the task into three regions. Entry calls and accepts are divided into two interactions each (e.g., starting an entry, ending an entry, starting an accept, ending an accept) because when a rendezvous is initiated, information can be passed from the calling task to the accepting task via the parameters of the call and accept statements. This changes the environment of the accepting task, dividing it into two regions at this point. When the rendezvous is ended, information can be passed in the other direction, dividing the calling task into two regions at this point. Special cases where a more compact representation can be used are considered in Section 4.

Finally, each edge in a TIG is labeled with the type of interaction that is occurring along the edge and with instance information such as task and entry names. In addition, edges may be grouped together into edge groups. These groups are used to model the Ada select statement and aid deadlock detection and are discussed in Section 4.

2.1 A Simple Example

The task shown in Figure 1 is used to illustrate what is meant by a TIG. In addition to several assignment statements, this task makes an entry call to another task (S2.P) and has one entry (accept Q).

```
task body S1 is
begin
    w := 1;
    if f(w) = 2 then
        S2.P;
        x := 2;
    else
        accept Q;
        y := 3;
    end if;
    z := 4;
end S1;
```

Figure 1: Task S1

As shown in Figure 2 the TIG for this task contains five regions. Region 1 consists of everything from the beginning of the task up to some task interaction, in this case, either the start of the entry call, S2.P, or the start of the accept, Q. Region 3

$N = \{1,2,3,4,5\}$
$E = \{(1,2),(2,3),(1,4),(4,5)\}$
$S = 1$
$T = \{3,5\}$
$L(1,2) = S2.P_S \quad L(2,3) = S2.P_E$
$L(1,4) = Q_S \quad L(4,5) = Q_E$

$C(1) =$
```
ENTER(WAIT_FOR_ACTIVATION);
task body S1 is
begin
    w := 1;
    if f(w) = 2 then
        EXIT(CALL_START(S2.P),2);
    else
        EXIT(ACCEPT_START(Q),4);
    end if;
```

$C(2) =$
```
ENTER(CALL_START(S2.P));
EXIT(CALL_END(S2.P),3);
```

$C(3) =$
```
    ENTER(CALL_END(S2.P));
    x := 2;
    z := 4;
end S1;
EXIT(TERMINATE,φ);
```

$C(4) =$
```
ENTER(ACCEPT_START(Q));
EXIT(ACCEPT_END(Q),5);
```

$C(5) =$
```
    ENTER(ACCEPT_END(Q));
    y := 3;
    z := 4;
end S1;
EXIT(TERMINATE,φ);
```

Figure 2: Task Interaction Graph for Task S1

consists of everything that occurs after the end of the entry call, S2.P, up to the next task interaction or, in this case, the end of the task. Similarly, region 5 consists of everything that occurs after the end of the accept, Q, up to the next task interaction or the end of the task. Region 2 consists of everything between the start of the entry call and the end of the entry call and region 4 consists of everything between the start of the accept and the end of the accept. Note that the last statement in the task (z := 4;) is part of both regions 3 and 5. This is because this statement would be executed under different circumstances depending on which of the two task interactions preceded it.

As can be seen, task interactions, and not control flow, cause transitions from one region to another. Thus, it is the task interaction (i.e., the start of the entry call) in the then-clause of the conditional statement that causes the transition from region 1 to region 2. If the then-clause contained only nontasking statements, then those statements would be a part of region 1 and there would be no transition out of region 1 at this point. Similarly, it is the start of the accept statement in the else-clause that causes the transition from region 1 to region 4. The end of the entry call causes the transition from region 2 to 3 and the end of the accept causes the transition from region 4 to 5.

The graphical representation of a TIG is shown in Figure 3. For the sake of brevity, the four task interactions represented in this example are represented by the labels $S2.P_S$, $S2.P_E$, Q_S, Q_E, where the subscripts S and E stand for start and end. In the following, a label containing a dot always represents an entry call; the part before the dot is a reference to a particular task and the part after the dot is a reference to a particular entry in that task. A label without a dot represents an accept. The arrow pointing to node 1 indicates that it is the start node and the double circle around nodes 3 and 5 indicates that they are terminal nodes.

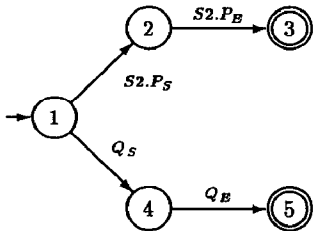

Figure 3: Graphical Representation of the TIG for task S1

2.2 A More Complex Example

Next we consider the TIG for a task that contains a more complicated control structure. Figure 4 shows a task based on an example in [Tayl83b]. The TIG for task T1 is shown in Figure 5 and the pseudocode for this graph is given in Figure 6.

The TIG for task T1 consists of five regions. The first region, represented by node 1, corresponds to everything that could occur from the time the task is activated until it makes an accept. Since the pseudocode for this region contains code that is executed only once, prior to the start of the loop, the task will not return to this region once it has left it.

The next two regions, represented by nodes 2 and 3, correspond to the bodies of the accept statements. The ENTER statement in each of these regions corresponds to an ACCEPT_START. For node 2 it represents the start of the P accept and for node 3 the start of the Q accept. Finally, each of these regions contains an EXIT statement that corresponds to the ACCEPT_END of the respective accepts.

The last two regions represent what happens after the end of the two accepts. The ENTER statement in each of these regions is found in the middle of the pseudocode because each of these regions is entered in the middle of a loop. After entering node 4, the loop is exited (note node 4 is a terminal node) or the end of the loop is reached causing a return to the beginning of the loop where the select statement is encountered. The select statement chooses between starting accept P or starting accept Q; thus, there is an edge from node 4 to node 2 and from node 4 to node 3 representing these transitions. Node 5 is similar to node 4 except it is entered after the end of the Q accept instead of the P accept.

```
task body T1 is
    DONE: boolean;
begin
    loop
        select
            accept P;
        or
            accept Q;
        end select;
        ...
        exit when DONE;
    end loop;
end T1;
```

Figure 4: Task T1

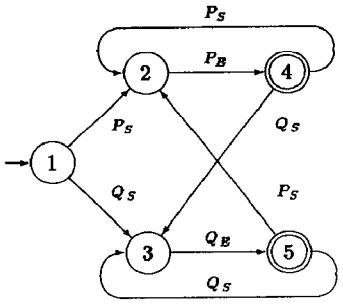

Figure 5: TIG for Task T1

3 Task Interaction Concurrency Graphs

A TICG represents the behavior of an entire program and is constructed from the TIGs of the tasks that make up the program.

Here, the TIG for $Task_i$ is denoted $G_i = (N_i, E_i, s_i, T_i, L_i, C_i)$. A vertex of a TICG, which is known as a *state* of the graph,[1] is a k-tuple, $(n_1, n_2, \ldots, n_k)$ where $n_i \in N_i$. States are connected by edges that represent the beginning and ending of rendezvous

[1]Not to be confused with the vertex of a task interaction graph, which is referred to as a node.

```
C(1) = ENTER(TASK_ACTIVATE);
       task body T1 is
           DONE: boolean;
       begin
           loop
               select
                   EXIT(ACCEPT_START(P),2);
               or
                   EXIT(ACCEPT_START(Q),3);
               end select;
           ...
           end loop;

C(2) = ENTER(ACCEPT_START(P)));
       EXIT(ACCEPT_END(P),4);
C(3) = ENTER(ACCEPT_START(Q)));
       EXIT(ACCEPT_END(Q),5);
C(4) =     loop
               select
                   EXIT(ACCEPT_START(P),2);
               or
                   EXIT(ACCEPT_START(Q),3);
               end select;
               ...
               ENTER(ACCEPT_END(P));
               ...
               exit when DONE;
           end loop;
       end T1;
       EXIT(TASK_TERMINATE,φ);
C(5) =     loop
               select
                   EXIT(ACCEPT_START(P),2);
               or
                   EXIT(ACCEPT_START(Q),3);
               end select;
               ...
               ENTER(ACCEPT_END(Q));
               ...
               exit when DONE;
           end loop;
       end T1;
       EXIT(TASK_TERMINATE,φ);
```

Figure 6: Pseudocode for Task T1

between tasks. There is an edge from state $(n_1, n_2, \ldots, n_k)$ to state $(m_1, m_2, \ldots, m_k)$ if there exists i and j such that for all $l \neq i, j$, $n_l = m_l$, and

(*i*) $(n_i, m_i) \in E_i$, and

(*ii*) $(n_j, m_j) \in E_j$, and

(*iii*) $L_i(n_i, m_i) = Task_j.E_S$ and $L_j(n_j, m_j) = E_S$, or
$L_i(n_i, m_i) = Task_j.E_E$ and $L_j(n_j, m_j) = E_E$.

The edge between these two states represents either the start or the end of a rendezvous between $Task_i$ and $Task_j$. Two states are said to be *adjacent* if they are connected by an edge that satisfies the above rules.

The definition given here for a TICG is similar to that given for control flow concurrency graphs [Tayl83b] in that there is an edge between two states only if that edge involves as few tasks as possible, e.g., there are no edges corresponding to several *independent* events occurring simultaneously. This approach does not overlook any possible states and includes all edges that correspond to the occurrence of a single event at a time. In addition, if more than one task makes an entry call on the same entry of a task there is an edge in the TICG corresponding to a rendezvous for each of these entry calls. This is because a concurrency state represents all possible orderings of the entries in the queue for each entry.

As an example, consider a program consisting of the tasks shown in Figures 4 and 7. The TIGs for these tasks are shown in Figure 5 and 8 and the TICG for this program is given in Figure 9. Each state in this graph is represented by a tuple (MAIN,T1,T2). The starting state for this example is $(1,1,1)$. There is an edge from state $(1,1,1)$ to state $(1,2,2)$, representing the start of a rendezvous between tasks T2 and T1, because there is an edge (1,2) in the TIG for task T2 and an edge (1,2) in the TIG for task T1. Similarly, there is an edge from $(1,1,1)$ to $(2,3,1)$ representing the start of a rendezvous between tasks T1 and MAIN. The other edges in the TICG likewise represent the start and end of rendezvous between pairs of tasks. Note that only states that represent valid synchronization states of the tasks are represented in the TICG. In general, the valid states are those that are reachable from the starting state $(s_1, s_2, \ldots, s_k)$.

```
procedure MAIN is          task body T2 is
begin                      begin
    T1.Q;                      T1.P;
end MAIN;                  end T2;
```

Figure 7: Procedure MAIN and Task T2

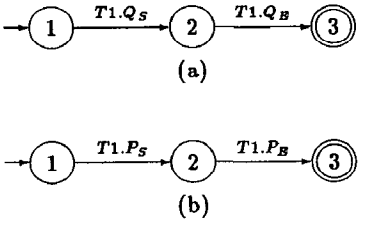

Figure 8: TIGs for (a) MAIN and (b) T2

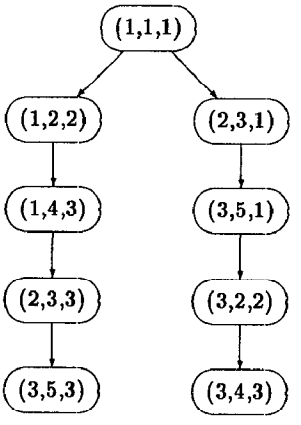

Figure 9: A Task Interaction Concurrency Graph for (MAIN, T1, T2)

A TICG for a program can be easily constructed by starting with a single state $(s_1, s_2, \ldots, s_k)$ and adding states and edges until no more states and edges can be added to the graph. The algorithm for constructing a TICG from a set of TIGs is given in Figure 10 and is very similar to the algorithm for constructing concurrency graphs from reduced flow graphs described by in [Tayl83b]. The time required to construct a TICG is comparable to the time required by the flow graph based algorithm.

```
TO_BE_CHECKED := {(s1, s2, ..., sk)};
while TO_BE_CHECKED ≠ empty do
    STATE := next state from TO_BE_CHECKED;
    for each state ASTATE adjacent to STATE do
        add edge (STATE,ASTATE) to EDGES;
        if ASTATE ∉ OLD_STATES then
            add ASTATE to TO_BE_CHECKED;
        end if;
    end loop;
    if deadlock occurs at STATE then
        output warning message;
    delete STATE from TO_BE_CHECKED;
    add STATE to OLD_STATES;
end loop;
```

Figure 10: Concurrency Graph Construction Algorithm

The basic algorithm is quite simple. Each state of the partially constructed TICG is checked to find its adjacent states. For each adjacent state, a new edge is added to the set of edges and the adjacent state is compared to a list of states that have already been found. If this state is a new state, i.e., it is not on the list of states that have already been found, then it is added to the list of states to be checked. After all the adjacent states of a state have been checked, then the state is added to the list of old states and deleted from the list of states to be check. The process is repeated using the next state in the list of states to be checked. The algorithm terminates when there are no more states to be checked. This algorithm uses three sets: TO_BE_CHECKED is the set of states to be checked, OLD_STATES is the set of states that have already been checked, and EDGES is the set of edges of the TICG. When the algorithm terminates, the TICG will consist of the states in the set OLD_STATES and edges in the set EDGES.

Using task interaction graphs to form a TICG instead of the reduced flow graphs results in a representation of a program's concurrency states that is usually substantially smaller than, and no larger than, the corresponding control flow concurrency graph, without any loss in power or applicability. Section 4.1 contains a direct comparison to a control flow concurrency graph and the corresponding TICG.

4 Analysis and Refinements

This section presents a flavor of some of the ways the task interaction model can be used to analyze concurrent systems and to represent realistic concurrent systems concisely. First, the model is extended to handle task activation and termination in Ada and then a deadlock detection analysis technique is described. Finally, an optimization that reduces the number of nodes in a concurrency graph is presented. Some more extensive analysis techniques have been explored along with several other refinements to the model, such as concise ways of capturing and evaluating shared variable usage and compact representations of procedure calls [Long88].

4.1 Task Activation and Termination

Up to this point it has been assumed that tasks all become active at the same time and terminate when the program stops execution. This ignores situations where tasks become active and terminate in the course of execution of a program. For example, the Ada model of task activation and termination allows tasks that are declared within a task, subprogram, or block to become active just prior to the start of execution of the first statement in that task, subprogram, or block. Using Ada rules, the task, subprogram, or block is known as the master of the activated task. Tasks can terminate (1) when they have finished execution and all their subordinate tasks have finished execution or (2) when the task is waiting at a terminate alternative, the task's master has terminated, and all other tasks activated by the task's master have either terminated or are waiting at a terminate alternative.

To include this hierarchical view of task activation and termination in the TIG model, interactions for representing these activities are needed. It is our view that the interactions involved in task activation and termination are of a different nature than those described earlier in the paper for rendezvous and, thus, are referred to as *implicit* interactions to distinguish them from the *explicit* interactions associated with the rendezvous. Just as explicit interactions separate tasks into regions, implicit interactions separate regions into *subregions*. It is important to note that adding extra nodes and edges to a task representation could have a substantial impact on the size of the concurrency graphs built from that representation. Implicit edges are used to construct concurrency graphs in a manner that will minimize their effect on the size of the resulting graph.

The implicit interactions used to model task activation and termination are illustrated in Figure 11. The four interactions illustrated here are WaitForActivation (WA), ActivateDependents (AD), WaitForDepsToTerminate (WT), and Terminate (T). Each master task will have an AD edge that corresponds to a point at which it activates dependent tasks and a WT edge that corresponds to the point at which all these dependents have terminated. Each dependent task will have a WA edge that corresponds to the point at which it becomes active and a T edge that corresponds to the point at which it terminates. Another implicit interaction, WaitSelectTerminate (ST), which is not illustrated here, is used to model the terminate alternative of the select statement.

Figure 11: Implicit Interactions in a Task

The use of implicit edges is illustrated with the following example. Suppose that the tasks of Figure 4 and Figure 7 are reorganized as shown in Figure 12. In this example, the tasks T1 and T2 are placed inside a procedure SUBR, which is called by the procedure MAIN. The TIGs for these tasks are shown in Figure 13. In these TIGs, regions are denoted by numbers and subregions are denoted by letters (i.e., the nodes 1a and 1b represent two subregions of region 1). Note that tasks without masters do not have WA and T edges and that tasks without dependents do not have AD and WT edges.

```
procedure MAIN is
    procedure SUBR is
        task body T1 is ...end T1;
        task body T2 is ...end T2;
    begin
        T1.Q;
    end SUBR;
begin
    SUBR;
end MAIN;
```

Figure 12: A Nested Concurrent Program

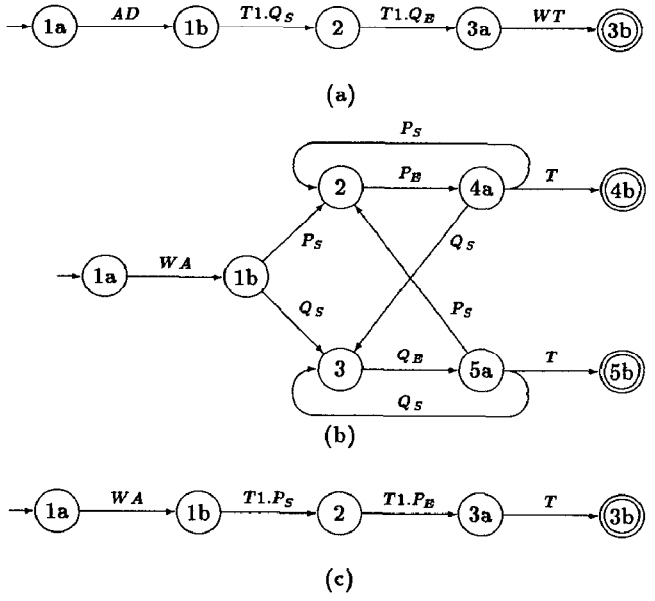

Figure 13: TIGs for (a) MAIN (b) T1 and (c) T2

The TICG for the TIGs in Figure 13 is given in Figure 14. This example illustrates several ways that implicit edges can be optimized during the construction of a TICG. For example, consider the edge between states $(1a, 1a, 1a)$ and $(1b, 1b, 1b)$ that represents the activation of the two dependent tasks by the main procedure. If a concurrency state were to be constructed for each intermediate step in this process, it would require five states to represent this activation as shown in Figure 15. Of course, the situation will be much worse when there are more than two dependent tasks. However, the unoptimized version provides no more information about the tasks than was already known, i.e., that there is more than one order in which the tasks can become activated. By advancing the master task along its AD edge at the same time as each of its dependents advance along their WA edges, these intermediate states can be eliminated without loss of any information. Similar optimizations are possible for other implicit edges. For example, in some situations tasks can advance along their T edges at the same time as their siblings and their masters. This occurs in Figure 14, in the transition from (3a, 5a, 3a) to state (3b, 5b, 3b). However, care must be exercised for

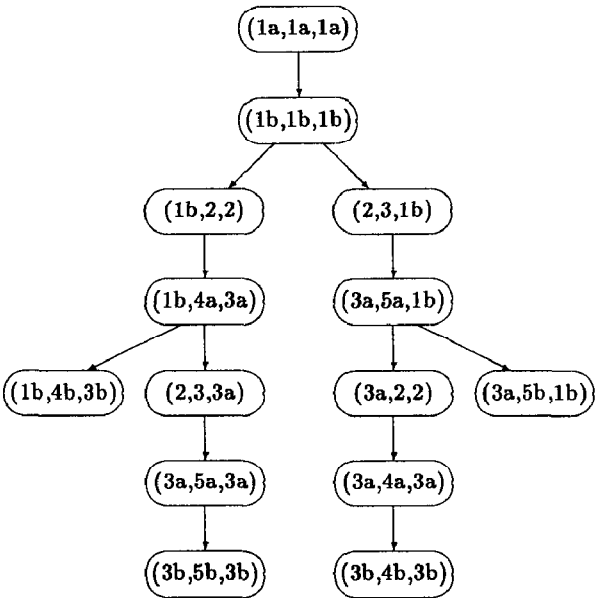

Figure 14: A TICG With Implicit Interactions

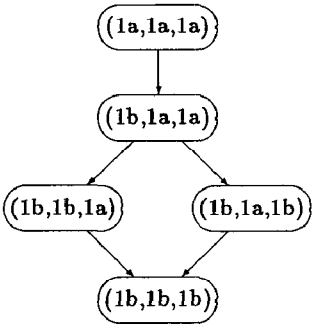

Figure 15: Unoptimized Task Activation

those tasks, such as T1, that have T edges and some other edge leaving the same node. When this occurs, each edge must be considered independently. For this example, this results in two successor states to state (1b, 4a, 3a) and state (3a, 5a, 1b).

The approach outlined here compares favorably with that found in [Tayl83b]. For example, in that paper the concurrency graph for a program similar to the one in Figure 12 contains 36 states and 54 edges as compared to the TICG representation, which has only 14 states and 13 edges. Moreover, this does not appear to be an unrepresentative example. Other programs we have examined have resulted in considerable reductions. For example, the TICGs for the 2,3, and 4 philosopher versions of dining

philosophers problem contain 40, 268, and 1792 states, respectively. The concurrency graphs contained in [Wamp85] for these examples contained 51, 470, and 4176 states, respectively.

The interactions described here are sufficient to model nested tasks, procedures, and blocks in Ada if there are no task interactions in the declarative regions of tasks. Such task interactions might occur, for example, if a declaration called a function that made an entry call. This restriction could be easily eliminated by the addition of an implicit edge type for TIGs that would force the elaboration of the declarations of all the dependents to complete before the execution of the body of the master task was allowed to begin. The cost of such a TIG edge, however, would be quite high in that it would require a large number of states in the associated TICG in order to represent all the possible ways the dependent tasks could complete the elaboration of their declarations and begin execution. As one might suspect from the resulting size of a concurrency graphs, allowing declarative task interactions is an unwise programming practice. Thus we decided not allowing task interaction in declarative regions is a reasonable restriction. On the other hand, using the TIG approach we have been able to remove most of the restrictions usually imposed by concurrency analysis techniques (e.g., [Dill88,Kemm88]). Such restrictions tend to make concurrency analysis inapplicable to realistic programs.

4.2 Deadlock Detection

Deadlock occurs when a task waits for a rendezvous that can never occur. To detect deadlock, a check can be applied during the construction of each state in the TICG. This check depends, in part, on the concept of edge groups. Groups are used to model Ada select statements where a task can select from among one of several different alternatives. A task will remain blocked until one of the alternatives can be chosen. A select statement is modeled in a TIG by placing all the edges representing its alternatives in the same edge group. Deadlock occurs at a state $(n_1, n_2, \ldots, n_k)$ if there exists an i and an edge $(n_i, m_i) \in E_i$ such that one of the following four conditions hold and no other tasks are able to rendezvous.

(i) $L_i(n_i, m_i) = Task_j.P_S$ and

(1) for no edge $(n_j, m_j) \in E_j$ does $L_j(n_j, m_j) = P_S$, or

(2) for some edge $(n_j, m_j) \in E_j$, $L_j(n_j, m_j) = Q$, where $Q \neq P_S$ and (n_j, m_j) is not in the same edge group as any edge labeled P_S.

(ii) $L_i(n_i, m_i) = Task_j.P_E$ and

(1) for no edge $(n_j, m_j) \in E_j$ does $L_j(n_j, m_j) = P_E$, or

(2) for some edge $(n_j, m_j) \in E_j$, $L_j(n_j, m_j) = Q$, where $Q \neq P_E$ and (n_j, m_j) is not in the same edge group as any edge labeled P_E.

(iii) $L_i(n_i, m_i) = P_S$ and for all $j \neq i$

(1) there is no edge $(n_j, m_j) \in E_j$ such that $L_j(n_j, m_j) = Task_i.P_S$, or

(2) there is an edge $(n_j, m_j) \in E_j$ such that $L_j(n_j, m_j) = Q$, where $Q \neq Task_i.P_S$ and (n_j, m_j) is not in the same edge group as any edge labeled $Task_i.P_S$.

(iv) $L_i(n_i, m_i) = P_E$ and for all $j \neq i$

(1) there is no edge $(n_j, m_j) \in E_j$ such that $L_j(n_j, m_j) = Task_i.P_E$, or

(2) there is an edge $(n_j, m_j) \in E_j$ such that $L_j(n_j, m_j) = Q$, where $Q \neq Task_i.P_E$ and (n_j, m_j) is not in the same edge group as any edge labeled $Task_i.P_E$.

The first two conditions have to do with entry calls. $Task_i$ might be waiting to start (or end) a call to entry P of $Task_j$ and $Task_j$ may not be at a node from which it could start (or end) the corresponding accept. This can occur in two ways. For the first case, it might be that $Task_j$ is in a region that does not contain the start (or end) of an accept P. In this case, there is no edge leaving node n_j of $Task_j$ that corresponds to this interaction. Even if $Task_j$ is in a region that contains the start (or end) of an accept P, there is no guarantee that the task will get to it, leading to case two. For case two, it might be that $Task_j$ executes a path through its current region that leads to a different task interaction, in which case it would end up waiting for some other task interaction to occur. In either of these cases, if no other tasks are able to rendezvous then this situation will never change and the program is deadlocked.

The last two conditions have to do with accepts. $Task_i$ might be waiting to start (or end) an accept P and there may be no task that is able to start (or end) a corresponding entry call. This can occur if for each other task either of the following two conditions hold: it is not in a region that contains a start (or end) of an entry call to accept P of $Task_i$, or it is waiting for some other interaction to occur.

In summary, deadlock occurs if $Task_i$ is waiting for a rendezvous and no other tasks are able to rendezvous at this point. Note that deadlock occurs even if $Task_i$ is able to rendezvous along some other edge $(n_i, m_i') \in E_i$ if this edge is not in the same edge group as (n_i, m_i). Thus, for task interaction concurrency graphs, deadlock can occur at states that have edges leading out of them as well as at states that have no edges leading out of them. For the example of Section 4.1, these rules would detect the two places where there is the potential for deadlock. In Figure 14 it can be seen that deadlock can occur at states (1b,4b,3b) and (3a,5b,1b) because T1 can terminate leaving either MAIN or T2 with no task with which to rendezvous.

4.3 A Reduction in the Number of Nodes

Concurrency graphs are very sensitive to small changes in the task interaction graphs used to create them. Any time the number of nodes in a TIG can be reduced, the corresponding TICG will be smaller. One place that one might try to make task interaction graphs smaller is by reducing the number of nodes that are needed to model entry calls and accepts. Unfortunately, at least three nodes are needed to model the general case of an entry call or an accept. However, in the special case where the accept statement has no body, two nodes are satisfactory.

To see why this can be done, consider an entry call statement that is currently modeled using three nodes. The pseudocode for the center node contains an ENTER pseudostatement and an EXIT pseudostatement and nothing else. This node represents the suspension of execution of the calling task while the accepting task is executing the body of the accept statement. When the accept statement has no body, it is also modeled with three nodes. The pseudocode for the center node contains an ENTER pseudostatement and an EXIT pseudostatement and nothing else.

When these tasks start the rendezvous, they each advance to their middle nodes. From here they can immediately end the rendezvous. Since this rendezvous is being used purely for synchronization, there is no loss in replacing this two step process by a single step.

The example in Section 3 is used to illustrate this reduction. Since neither of the accepts have bodies, the edges representing the start and end of the entry calls or accepts can be replaced with a single edge representing the entire entry call or accept, respectively. (The subscript *SE* is used in the label on such an edge.) Thus, for task MAIN and T1, node 2 can be eliminated and, for task T2, nodes 2 and 3 can be removed. The pseudocode for the remaining nodes is almost the same as in the original example except that the interactions CALL_START_END and ACCEPT_START_END are used to indicate the entire call or accept.

The new concurrency graph for this example is given in Figure 16 and is almost half the size of the original concurrency graph.

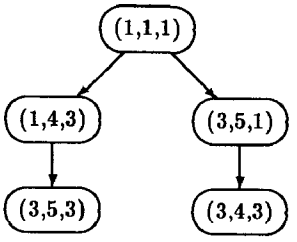

Figure 16: A Simplified Concurrency Graph

5 Conclusion

The TIG model of concurrent systems provides an interesting representation of concurrency that facilitates analysis. The approach recognizes maximal sections of noninteracting code sections and represents each such region as a node in the task interaction graph. The edges of this graph capture the task interactions. We have shown that using the TIG model can result in concurrency graph representations that are substantially smaller than models based on control flow. Moreover, because it is easy to identify the code associated with a state in the concurrency graph, it appears that certain kinds of analysis, such as symbolic execution [Youn86], can be more easily supported using this model.

To date we have developed rules for translating most of the constructs supported by Ada into the appropriate TIG representation. In this translation, we attempt to find rules that will reduce, or at least not increase, the number of nodes in the TIG, since any reduction in the number of nodes in the TIG results in a significant reduction in the associated TICG. Such a reduction will also have a corresponding reduction in the cost of any associated analysis. Thus, it is worthwhile to carefully consider further optimizations that can be performed on the TIG representation. For example, we have been investigating optimizations for procedure calls that activate tasks, shared variable references, and some others situations. It is important to note that we have been able to define rules for concisely representing most reasonable concurrency constructs that occur in Ada. Unlike many concurrency analysis approaches that must severely restrict the types of programs they can consider, the TIG approach can be applied to realistic concurrent systems.

For the programs we have examined so far, the number of states in the TICG have been quite reasonable. Furthermore, each resulting TICG has been substantially smaller than the corresponding control flow concurrency graph. Our hypothesis is that the complexity of the TICG for the typical system will be quite reasonable, although worse case analysis clearly shows it is an intractable problem [Tayl83a,DeMi79]. We feel it is imperative to conduct some experimental studies so we can evaluate typical performance for realistic systems. We are currently building a prototype system that can automatically create the TIG and TICG for actual, production programs. We intend to use the prototype to do experimental studies on the size and complexity of the generated graphs.

We have been investigating several kinds of analysis techniques that can be applied to the TIG and TICG models. Some of these techniques are relatively simple to apply and can be carried out during the creation of the graphs; others require post processing, which might even be directed by information gathered during the creation of the graphs. An example of one kind of analysis that can be done during the creation of the TICG is deadlock detection, as is shown in Section 4. An example of the kind of analysis that requires post processing is "dangerous" parallelism. This occurs when a shared variable can be assigned and referenced in various orders. The possibility of such situations can be detected during TICG construction and then analyzed afterwards.

Deadlock detection and dangerous parallelism are just two examples of the kinds of analysis that can be performed using a TICG representation. We are currently investigating more powerful analysis techniques. Because sequential processing is carefully separated from task interactions, it appears that some sequential analysis techniques could be applied to task regions and the results incorporated into inter-region analysis, similar to the techniques currently used for inter-procedural analysis.

ACKNOWLEDGMENT

The authors wish to acknowledge Joe Fialli, who has provided many helpful suggestions on the development of the TIG model and this paper.

References

[Ada83] **Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A),** United States Department of Defense, Washington, D.C., January 1983.

[Avru85] George S. Avrunin, Laura K. Dillon, Jack C. Wileden, and William E. Riddle. Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems. Dept. of Comp. and Info. Science, University of Massachusetts, Amherst, Technical Report 85-13, May 1985.

[Brin78] Per Brinch Hansen. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934-941, November 1978.

[Bris79] G. Bristow, C. Drey, B. Edwards and W. Riddle. Anomaly Detection in Concurrent Programs. *Proceedings of the 4th International Conference on Software Engineering*, 265–273, 1979.

[DeMi79] Richard DeMillo and Raymond Miller. Implicit Computation of Synchronization Primitives. *Information Processing Letters*, 9(1):35-38, July 1979.

[Dill88] Laura K. Dillon. Symbolic Execution-Based Verification of Ada Tasking Programs. *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, 3-13, May 1988.

[Helm85] David Helmbold and David Luckham. Debugging Ada Tasking Programs. *IEEE Software*, 2(2):47–57, March 1985.

[Hoar78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Kemm88] L.J. Harrison and R.A. Kemmerer. An Interleaving Symbolic Execution Approach for the Formal Verification of Ada Programs with Tasking. *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, 3-13, May 1988.

[Long88] Douglas L. Long and Lori A. Clarke. Task Interaction Graphs: A Representation For Concurrency Analysis. Department of Computer & Information Science, University of Massachusetts, Amherst, March 1988.

[Morg87] E. Timothy Morgan and Rami R. Razouk. Interactive State–Space Analysis of Concurrent Systems. *IEEE Transactions on Software Engineering*, SE-13(10):1080–1091, October 1987.

[Shat88] 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.

[Tai85] K.C. Tai. On Testing Concurrent Programs. *Proceedings of COMPSAC 85*, 310–317, October 1985.

[Tayl80] Richard N. Taylor and Leon J. Osterweil. Anomaly Detection In Concurrent Software By Static Data Flow Analysis. *IEEE Transactions on Software Engineering*, SE-6(3):265–278, May 1980.

[Tayl83a] Richard N. Taylor. Complexity of Analyzing the Synchronization Structure of Concurrent Programs. *Acta Informatica*, 19:57–84, 1983.

[Tayl83b] Richard N. Taylor. A General-Purpose Algorithm For Analyzing Concurrent Programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[Wamp85] Gordon K. Wampler. A Static Concurrency Analysis Tool for Ada (SCA). Master's Dissertation, University of California, Irvine, 1985.

[Youn86] Michal Young and Richard N. Taylor. Combining Static Concurrency Analysis With Symbolic Execution. In *Proceedings of the Workshop on Software Testing*:170–178, IEEE Computer Society Press, July 1986.