TASK INTERACTION GRAPHS: AN INTERMEDIATE REPRESENTATION FOR CONCURRENCY ANALYSIS

Douglas L. Long†
Lori A. Clarke‡

COINS Technical Report 88–47
December 1988

†Department of Computer Science
Wellesley College
Wellesley, Massachusetts 02181

‡*Software Development Laboratory*
Department of Computer & Information Science
University of Massachusetts
Amherst, Massachusetts 01003

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

Task Interaction Graphs: An Intermediate Representation for Concurrency Analysis

Douglas L. Long
Lori A. Clarke

December 1988

Abstract

A representation for concurrent programs, called *task interaction graphs*, is presented. Task interaction graphs divide a program into maximal sequential regions, which are connected by edges representing task interactions. The construction of task interaction graphs for a number of common programming language features, including tasks and procedures, is described and techniques for handling shared variables are considered. Formal descriptions of the graphs for each of these features is presented along with several examples. 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

The design and analysis of concurrent programs is widely considered to be more difficult than the design and analysis of sequential programs since the introduction of parallelism into a program introduces problems, such as deadlock, that are not found in sequential programs. Even though they are more difficult to write, concurrent programs offer many advantages over sequential programs, and it is therefore desirable to develop tools and techniques to aid in their development. The techniques that have been developed for the analysis of concurrent programs can be divided into two classes. Run-time techniques monitor a program during execution and collect information about the behavior of the program. Static analysis techniques try to determine the behavior of the program before execution. This report describes a new representation for tasks that can be used in the analysis of concurrent programs and as a basis for extending some sequential analysis techniques to concurrent programs.

Early attempts to deal with the complexity of writing concurrent programs led to the development of many different models of concurrent programming. One of the more recent of these is the rendezvous model found in Ada. This model is used in this report and the examples are illustrated with an Ada-like language. Although much of the following centers around a specific language, the results of this report are applicable to any language that uses rendezvous-like synchronization such as Ada [Ada83], Distributed Processes [Brin78], and CSP [Hoar78]. In the rendezvous model, a concurrent program consists of a number of *tasks* that may be executed concurrently. Tasks interact with each other through *rendezvous*. These occur when a task makes an *entry call* to an entry of another task and the other task *accepts* the entry call. The execution of a task making an entry call or an accept is suspended until another task makes a corresponding accept or call. Calls and accepts may also have parameters that are used to exchange data at the start and end of a rendezvous.

This report builds on the technique of static concurrency analysis developed by Taylor [Tayl83b], which is also applicable to languages that use the rendezvous model of concurrency. This approach uses a reduced flow graph for each task to construct a concurrency graph that models the behavior

of the program.[1] Each vertex (or state) of the concurrency graph represents a possible concurrency state of the program. A concurrency state is a tuple consisting of one node from the reduced flow graph of each task in the program. The concurrency graph can be analyzed to answer questions about the possible behavior of the program, such as: which rendezvous occur, what actions may occur in parallel, and when does deadlock occur? Concurrency graphs created from flow graphs are referred to in this report as *control flow concurrency graphs.*

This report defines a smaller, more compact representation for tasks called a *task interaction graph.* The central idea underlying this representation is that *task interactions* divide a task into *task regions.* A task interaction occurs at a point in a task where the behavior of the task can be influenced by another task. Task synchronization and the exchange of data between tasks are examples of task interactions. Task regions are the natural embodiment of the behavior of interacting tasks. An individual task region encompasses all possible behaviors of a task starting at one task interaction and ending at the next interaction. Regions contain only one entry point, corresponding to the starting interaction, but may contain many exit points, each reachable by a different path through the region. This single entrance, multiple exit characteristic is a consequence of the desire for task regions to be maximal, in the sense that they contain everything except direct task interactions. Task interaction graphs can be used to construct concurrency graphs, called *task interaction concurrency graphs,* that have a number of advantageous properties.

This work is motivated in two ways. First of all, in order for concurrency graphs to be useful it is necessary to control their size. The number of states in a concurrency graph can be exponential in the number of tasks in the program due to the combinatorial nature of the way the states are constructed. More significant than the potential for there being a large number of states, however, is the relationship between the number of nodes in the representation of a single task and the number of states in a concurrency graph of a program containing that task. A small increase in the number of nodes in the task representation can result in a large increase in the number of states in the concurrency graph. In the worst case, the addition of a single node to one task could result in twice as many states in the concurrency graph.

[1] A flow graph is reduced by removing those nodes that are not in some way related to tasking.

One way to reduce the number of states in a concurrency graph is to reduce the number of nodes needed to represent each task in a program. A task interaction graph tries to use as few nodes as possible to represent a task. Since even a small reduction in the size of the representation of a just a single task can result in a much smaller concurrency graph, task interaction concurrency graphs will, in general, be much smaller than concurrency graphs constructed from less compact representations. The use of the more compact representation does not result in any loss of applicability or generality. The savings in the size of the concurrency graphs results from representing more information about a task at each node.

The second motivation for this work is to provide a better representation for the way one task interacts with other tasks. A task interaction graph emphasizes task interaction over other activities of the task. If a task did not interact with another task, then an analysis of the task could be carried out as if it were a sequential program, independent of external influence. Task interactions introduce external factors into the behavior of the task that cause it to be dependent on the states of all the other tasks in the program. However, these external factors only change at points of task interaction. Between task interactions these external factors do not change, presenting the task with a constant environment. Thus, task interactions divide a task into regions each of which can essentially be treated as a block of sequential code. Task interaction graphs represent tasks in a way that reflects the importance of the task interactions and the task regions that are determined by the task interactions. In this representation, control structures are treated secondarily to task interactions.

Task interaction graphs provide a basis for a variety of different analysis techniques beyond those described in this report. A task region, which has a constant environment with respect to the rest of the program, is a maximal piece of code that can be analyzed independently from other regions. This independence means that each region can be treated as a unit and the overall analysis of the program only requires analyzing the interactions between these units. This may lead to considerable simplification when applying other types of analysis to concurrent programs. This will be the subject of future work.

The next section discusses task interaction graphs in more detail. The first part of Section 2 defines task interaction graphs formally, the next part shows how to construct them and the third

part gives an example. Section 3 discusses task interaction concurrency graphs and how to construct them from task interaction graphs. Section 4 extends the definitions in Section 2 and compares task interaction concurrency graphs with control flow concurrency graphs and Section 5 suggests directions for future work.

```
task body T1 is
begin
    w := 1;
    if f(w) = 2 then
        T2.P;
        x := 2;
    else
        accept Q;
        y := 3;
    end if;
    z := 4;
end T1;
```

Figure 1: Task T1

2. Task Interaction Graphs

The previous section introduced the idea that a task should be considered to be a set of interacting task regions, each of which has a constant environment with respect to the rest of the program. This section shows how task interaction graphs represent tasks as sets of regions and interactions between regions.

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 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. The third restriction is relaxed in Section 4.1.

The task shown in Figure 1 is used to illustrate what is meant by a task region. In addition to several assignment statements, this task makes an entry call to another task (T2.P) and has one entry (Q). An informal picture of the task interaction graph for this task is shown in Figure 2. This task contains five regions. Region 1 consists of everything from the beginning of the task up to

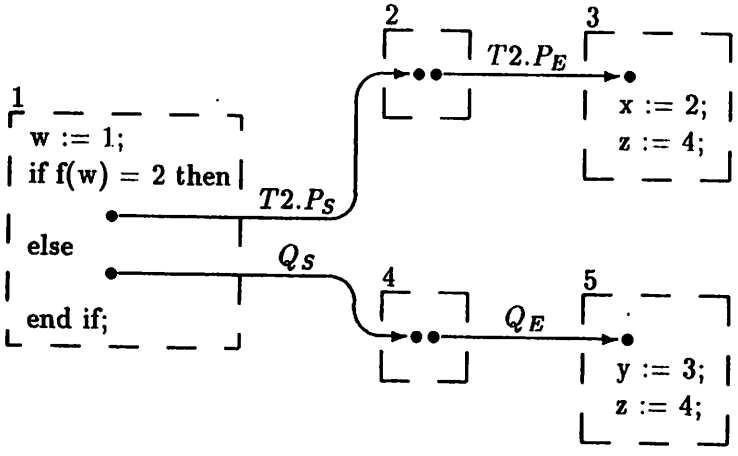

Figure 2: Informal task interaction graph for task T1

some task interaction, in this case, either the start of the entry call, T2.P or the start of the accept, Q. Region 3 consists of everything that occurs after the end of the entry call, T2.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 5 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.

Calls and accepts actually cause four distinct types of task interactions: starting an entry call, ending an entry call, starting an accept, and ending an accept. In Figure 2, the task interactions are represented by the arrows from one region to another. Task interactions, and only task interactions, cause transitions from one region to another.[2] 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 non-tasking statements, then those statements would be

[2]The reader should keep in mind that these edges do not represent control flow and that this is not a control flow graph.

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 call causes the transition from region 2 to 3 and the end of the accept causes the transition from region 4 to 5. As can be seen from this example, each call and each accept is modeled using two interactions and three regions.

Calls and accepts are divided into two interactions each 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. The start of a rendezvous does not actually change the environment of the calling task nor does the end of a rendezvous change the accepting task. Nevertheless the calling task is also divided into two regions at the start of a call and the accepting task is divided into two regions at the end of a rendezvous. This makes it easier to keep track of the synchronization between the calling and accepting tasks. In general, this will not affect the size of concurrency graphs because the calling and accepting tasks are linked together by the rendezvous. Special cases where a more compact representation can be used are considered in Section 4.3.

Each node in a task interaction graph (TIG) represents a different region of the task and has associated with it an explicit representation of the code for that region. The representation for regions is not fixed by the task interaction graph model and may vary based on the analysis to be done. Any convenient representation for task regions can be used as long as it can be modified to include a small amount of additional information that indicates where regions begin and end. A representation for regions that includes a way to mark the beginning and end of regions is referred to as *pseudocode*. In this report, the pseudocode for regions consists of the same Ada-like language that is used to represent tasks with addition of four non-executing *pseudostatements* that mark the beginning and the end of regions.

A task interaction causes a transition from one region to another. The boundary between these regions is represented by two pseudostatements - one in each of the two regions connected by that interaction. The pseudostatement in the first region indicates a place where that region may be

exited and the type of task interaction that causes the transition from the first region to the second region. The pseudostatement in the second region indicates the place where that region may be entered. In the informal TIG in Figure 2 these pseudostatements are represented by the solid dots. An interaction is represented by an edge of the TIG. In the formal representation of the TIG, for each edge between two nodes there is a pseudostatement in one node representing the head of the edge and a pseudostatement in the other node representing the tail of the edge.

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. The labels on the edges provide information about the interactions and the pseudocode at each node provides information about each region.

We can return to the task shown in Figure 1 to illustrate some of the features of a TIG. The TIG for this task is shown in Figure 3. This task has five task regions separated by tasking activities. Nodes 1, 3 and 5 were discussed earlier. Nodes 2 and 4 are necessary for the correct modeling of the entry call and accept and represent regions between the start and end of the call and accept. The pseudocode for each node is represented using Ada-like code except that calls and accepts are replaced by the pseudostatements ENTER(*interaction*) and EXIT(*interaction*,*next*). *Interaction* refers to the type of task interaction that causes the entrance to and exit from the region and *next* is the region that is entered after the interaction that causes the exit from the region. In this example, the four interactions are: CALL_START(T2.P), CALL_END(T2.P), ACCEPT_START(Q), and ACCEPT_END(Q). The *interaction* information in the ENTER pseudostatement and the *next* information in the EXIT pseudostatement are redundant since this information can be determined from the set of edges and the labels on the edges. They have been included in the pseudocode in order to make it more informative to the reader.

Each edge in a TIG is labeled with the type of interaction that is occurring along the edge. For the sake of brevity, the four types of task interaction represented in this example are represented by the labels $T2.P_S$, $T2.P_E$, Q_S, Q_E, where the subscripts S and E stand for start and end. In the

$N = \{1,2,3,4,5\}$
$E = \{(1,2),(2,3),(1,4),(4,5)\}$
$S = 1$
$T = \{3,5\}$
$L(1,2) = T2.P_S \quad L(2,3) = T2.P_E \quad L(1,4) = Q_S \quad L(4,5) = Q_E$

$C(1) =$ ENTER(TASK_ACTIVATE);
task body T1 is
begin
w := 1;
if f(w) = 2 then
EXIT(CALL_START(T2.P),2);
else
EXIT(ACCEPT_START(Q),4);
end if;

$C(2) =$ ENTER(CALL_START(T2.P));
EXIT(CALL_END(T2.P),3);

$C(3) =$ ENTER(CALL_END(T2.P));
x := 2;
z := 4;
end T1;
EXIT(TASK_TERMINATE,ϕ);

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

$C(5) =$ ENTER(ACCEPT_END(Q));
y := 3;
z := 4;
end T1;
EXIT(TASK_TERMINATE,ϕ);

Figure 3: Formal task interaction graph for task T1

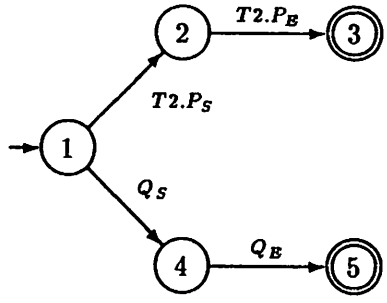

Figure 4: A task interaction graph for task T1

following, a label containing a dot will always represent a call; the part before the dot is a reference to a particular task and the part after the dot a reference to a particular entry in that task. A label without a dot will represent an accept. In addition, edges may be grouped together into edge groups. These groups are used to model the Ada select statement and aid deadlock detection. Unless explicitly mentioned otherwise each edge in a TIG belongs to its own unique edge group.

A drawing of this TIG is shown in Figure 4. 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.

2.1 Constructing Task Interaction Graphs

The first step in our analysis of a concurrent program is to construct a task interaction graph for each task. This section shows how to translate a high level programming language representation of a task to a TIG for that task. Of course, TIG's can be constructed from other semantically equivalent task representations as well.

In many languages, program statements are constructed from one or more simpler statements. TIG's can be constructed in much the same way. I.e., the TIG of a compound statement can be constructed from the TIG's of the embedded statements. This section starts by describing the TIG's for call statements and accept statements and other simple statements. Most of the remainder

of Section 2.1 shows how to construct the TIG's for a variety of common compound statements. Section 2.1 concludes with a discussion of TIG's for tasks. An example of the translation process is shown in Section 2.2.

In the following, general programming language structures are enclosed in ⟨⟩'s. For example, ⟨*statement*⟩ represents any valid statement and ⟨*statement-list*⟩ represents a sequence of one or more ⟨*statement*⟩'s. $C(i)||C(j)$ denotes the concatenation of the pseudocode $C(i)$ and $C(j)$.

2.1.1 Simple Statements

This section shows how to construct task interaction graphs for simple statements, i.e., statements that do not have other statements embedded in them.

The first statements considered are simple non-tasking statements, such as assignment statements. These statements do not cause any interaction with another task so must necessarily be contained entirely within a region. Thus, the TIG of such a statement consists of a single node u and no edges. The node u is both the start node and the only terminal node. The pseudocode for this node is the statement itself.

Simple tasking statements, i.e., calls and accepts, are considered next.

A ⟨*call-statement*⟩ is an entry call to another task. For example, T.P(⟨*parameter-list*⟩) is a call to entry P of task T. Recall that an entry call causes two task interactions, the start of the call and the end of the call. Thus, an entry call separates a task into three separate task regions: activity before the rendezvous (including waiting for it to occur), activity during the rendezvous (when the task is suspended), and activity after the rendezvous. The start of the call causes the transition from the first region to the second region and the end of the call causes the transition from the second region to the third region. During the start of the call, data may be transferred from the calling task to the called task via parameters and during the end of the call, data may be transferred in the opposite direction.

An entry call is modeled using a TIG that consists of three nodes and two edges. The nodes represent the three separate regions and the edges represent the start and the end of the entry call.

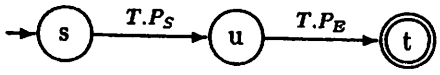

Figure 5: TIG for an entry call.

Here is the TIG of a ⟨*call-statement*⟩.

$$
\begin{array}{ll}
N & = \{s, u, t\} \\
E & = \{(s, u), (u, t)\} \\
S & = s \\
T & = \{t\} \\
L(s, u) & = T.P_S \quad L(u, t) = T.P_E \\
C(s) & = \text{EXIT(CALL\_START(T.P(⟨}\textit{parameter-list}\text{⟩)),}u\text{);} \\
C(u) & = \text{ENTER(CALL\_START(T.P(⟨}\textit{parameter-list}\text{⟩)));} \\
 & \quad \text{EXIT(CALL\_END(T.P(⟨}\textit{parameter-list}\text{⟩)),}t\text{);} \\
C(t) & = \text{ENTER(CALL\_END(T.P(⟨}\textit{parameter-list}\text{⟩)));}
\end{array}
$$

A drawing of this TIG is shown in Figure 5.

An ⟨*accept-statement*⟩ is treated similarly to a ⟨*call-statement*⟩. The general form of a ⟨*accept-statement*⟩ without a body is

accept Q(⟨*parameter-list*⟩);

An accept also causes two task interactions, the start of the accept and the end of the accept. Thus, an accept without a body also separates a task into three parts. There is a region corresponding to activity before the rendezvous, activity during the rendezvous, and activity after the rendezvous. The TIG for this statement follows.

$$
\begin{array}{ll}
N & = \{s, u, t\} \\
E & = \{(s, u), (u, t)\} \\
S & = s \\
T & = \{t\} \\
L(s, u) & = Q_S \quad L(u, t) = Q_E \\
C(s) & = \text{EXIT(ACCEPT\_START(Q(⟨}\textit{parameter-list}\text{⟩)),}u\text{);}
\end{array}
$$

$C(u)$ = ENTER(ACCEPT_START(Q(⟨*parameter-list*⟩)));
EXIT(ACCEPT_END(Q(⟨*parameter-list*⟩)),t);
$C(t)$ = ENTER(ACCEPT_END(Q(⟨*parameter-list*⟩)));

An ⟨*accept-statement*⟩ may also have a body; this form is considered in the next section. The next section also shows how the TIG's of ⟨*accept-statement*⟩'s are incorporated into the TIG's of a select statement.

2.1.2 Statement Lists

This section and the next shows how to construct TIG's for more complicated statements. The discussion here will be kept informal. Formal descriptions of the TIG's for the statements discussed here can be found in the Appendix. The statements considered here do not themselves cause task interaction, but may contain embedded statements that do.

A ⟨*statement-list*⟩ is a sequence of one or more statements, each of which may or may not contain embedded ⟨*call-statement*⟩'s or ⟨*accept-statement*⟩'s. The TIG of a ⟨*statement-list*⟩ is constructed from the TIG's of the individual statements in the list. Since all statements that can be executed between two task interactions are by definition part of a single region, when two statements are executed in sequential order the flow of control from the first statement to the second statement does not imply a transition between regions. Thus, the TIG for two sequential statements is constructed by merging together a terminal node in the TIG for the first statement with the start node of the TIG for the next statement. This is done for each terminal node of the TIG of the first statement such that control can pass from the part of the statement represented by that node to the second statement.[3] Since a TIG may have more than one such terminal node, each of these terminal nodes is merged with a copy of the start node of the next statement. Note that this causes the replication of the pseudocode for the start node of the second statement in different nodes of the new TIG.

The TIG of a ⟨*statement-list*⟩ of more than two statements is constructed by repeated application of the above rules.

[3]Some terminal nodes might represent parts of the first statement that would cause the task to terminate. In this case, there would be no transfer of control from the first statement to the second statement and thus no need to merge the start node of the second statement onto these terminal nodes.

Several example's will be used to illustrate how the TIG's of statements in a statement list are combined to construct the TIG of the statement list. First, consider the case of two simple non-tasking statements. The TIG's of each statement are single nodes and these TIG's are combined by merging these two nodes together into a single node. This node represents the region that contains the two simple statements. Similarly, the TIG of a sequence of simple non-tasking statements would consist of a single node representing the region that contains these statements.

Next, consider a simple non-tasking statement and a simple tasking statement such as a ⟨*call-statement*⟩. The TIG's for these statements are combined by merging the single node of the TIG of the simple non-tasking statement and the start node of the TIG for the ⟨*call-statement*⟩. This has the effect of adding the simple non-tasking statement to the region that comes before the start of the call. In a similar manner, simple non-tasking statements that follow simple tasking statements are added to the region that follows the end of the call or accept.

The TIG's of two simple tasking statements are combined by merging the terminal node of the first TIG with the start node of the second TIG. This node represents the region between the end of the first call or accept and the start of the second call or accept.

2.1.3 Compound Statements

This section shows how to construct TIG's for compound statements that contain embedded calls or accepts.

The TIG for an ⟨*accept-statement*⟩ with a body can now be constructed. The general form of these statements is

accept Q(⟨*parameter-list*⟩) do ⟨*statement-list*⟩ end Q;

An accept with a body also separates the task into three parts. There is a region corresponding to the activity before the rendezvous and a region corresponding to activity after the rendezvous. However, during the actual rendezvous the task is not suspended and thus may not be represented by a single region. Instead, it is executing the enclosed ⟨*statement-list*⟩ and thus is represented by the TIG of the ⟨*statement-list*⟩. The start of an accept corresponds to a transition from the before

region to the start node of the TIG of the ⟨*statement-list*⟩. The end of an accept corresponds to a transition from a terminal node of the TIG of the ⟨*statement-list*⟩ to the after region.

An ⟨*accept-statement*⟩ with a body is modeled by constructing a new TIG from $G_1 = (N_1, E_1, S_1, T_1, L_1, C_1)$, the TIG of the ⟨*statement-list*⟩. The new TIG is constructed by adding a new start node s and a new terminal node t. An edge is added from s to s_1 and from each terminal node in G_1 to t. The complete graph is defined as follows:

$$
\begin{aligned}
N &= N_1 \cup \{s, t\} \\
E &= E_1 \cup \{(s, s_1)\} \cup \{(n, t) \mid n \in T_1\} \\
S &= s \\
T &= \{t\}
\end{aligned}
$$

$$
L(n,m) = \begin{cases} L_1(n,m) & \text{if } (n,m) \in E_1 \\ Q_S & \text{if } (n,m) = (s, s_1) \\ Q_E & \text{if } n \in T_1 \text{ and } m = t \end{cases}
$$

$$
C(n) = \begin{cases} C_1(n) & \text{if } n \in N_1 - \{T_1 \cup s_1\} \\ \text{EXIT(ACCEPT\_START(Q(⟨}\textit{parameter-list}\text{⟩))),}s_1\text{)} & \text{if } n = s \\ \text{ENTER(ACCEPT\_START(Q(⟨}\textit{parameter-list}\text{⟩)))}\|C_1(n) & \text{if } n = s_1 \\ C_1(n)\|\text{EXIT(ACCEPT\_END(Q(⟨}\textit{parameter-list}\text{⟩))),}t\text{)} & \text{if } n \in T_1 \\ \text{ENTER(ACCEPT\_END(Q(⟨}\textit{parameter-list}\text{⟩)))} & \text{if } n = t \end{cases}
$$

A drawing of this TIG is shown in Figure 6. The dashed box represents G_1, where s_1 is the start node of G_1 and $t_{1,1}, \ldots, t_{1,k}$ are the terminal nodes of G_1.

The next statement to consider is a simple conditional statement.

if ⟨*expression*⟩ then ⟨*statement-list*⟩ end if;

If the ⟨*expression*⟩ is true then the ⟨*statement-list*⟩ is evaluated. This means that the test and the pseudocode for the start node of the ⟨*statement-list*⟩ are part of the same region. If the ⟨*expression*⟩ is false then the ⟨*statement-list*⟩ is not evaluated, which means that there would be no transition out of the region containing the test.[4] This means that the node containing the test would be a

[4] In principle, it is possible that the ⟨*expression*⟩ might contain a call to a function that contains task interactions. In this case, the TIG would have to be modified so that the evaluation of the ⟨*expression*⟩ is split over more than one region. However, since the order in which operands of an expression is evaluated is not always well-defined this case will not be considered here.

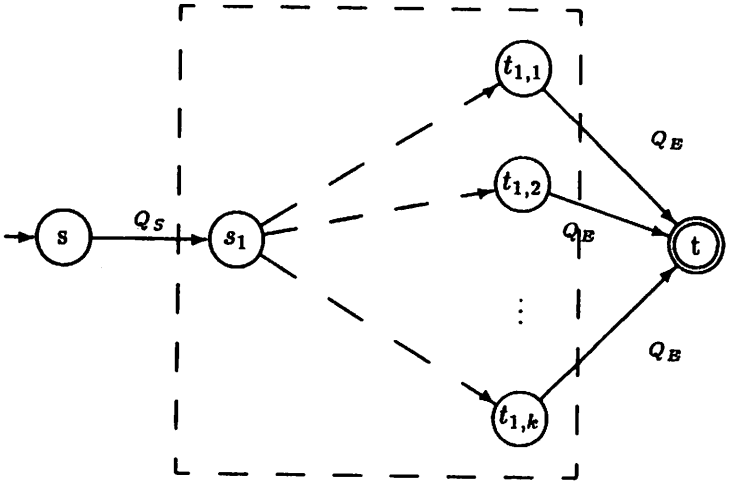

Figure 6: TIG for an accept.

terminal node of the TIG of the conditional statement. The TIG of the if statement is constructed from the TIG of the ⟨*statement-list*⟩. The only changes to the TIG of the ⟨*statement-list*⟩ are to add the evaluation of the conditional expression to the start node and to add the start node to the set of terminal nodes if it wasn't already there.

A more general form of the if statement contains an else clause.

if ⟨*expression*⟩ then $\langle \textit{statement-list} \rangle_1$ else $\langle \textit{statement-list} \rangle_2$ end if;

If the ⟨*expression*⟩ is true then $\langle \textit{statement-list} \rangle_1$ is evaluated, otherwise $\langle \textit{statement-list} \rangle_2$ is evaluated. This means that the test, the pseudocode for the start node of $\langle \textit{statement-list} \rangle_1$, and the pseudocode for the start node of $\langle \textit{statement-list} \rangle_2$ are part of the same region. The TIG of this statement is constructed from the TIG's of the ⟨*statement-list*⟩'s by combining the start nodes of the two TIG's into a single node. The pseudocode for this new start node contains the test and the pseudocode for the original start nodes.

See Figure 4 for an example of a TIG for an if-then-else.

Next we consider the TIG of a general loop statement

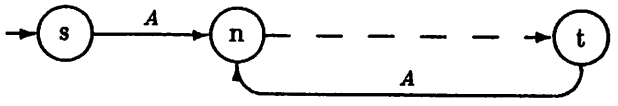

Figure 7: Adding edges for a loop.

loop ⟨*statement-list*⟩ end loop;

where the ⟨*statement-list*⟩ may contain one or more ⟨*exit-statement*⟩'s. An ⟨*exit-statement*⟩ is a statement of the form

exit when ⟨*expression*⟩;

that causes the loop to terminate if the ⟨*expression*⟩ is true.[5] The first step in the construction is to add looping edges from the terminal nodes of G_1, the TIG of ⟨*expression*⟩, back to the beginning of the loop. If this were a control flow graph, edges would be added from the terminal nodes to the start node. But in a TIG, edges must represent task interaction of some sort. Instead, a copy of the start node is incorporated into each terminal node and an edge is added from each terminal node to each node that can be reached from the start node. The label on each of these edges is the same as the label on the edge from the start node to the node reached by the edge. This is illustrated in Figure 7. For each edge (s, n), where s is a the start node, an edge (t, n) is added from each terminal node t to node n.

The next step is to construct the set of terminal nodes. These are the nodes where the task will be when the loop terminates. There are two ways that a loop can terminate at a node. The first is when the pseudocode for the node contains a statement that will cause the task to terminate. The second is when the pseudocode for the node contains a statement that will cause an exit from the loop. Finally, the pseudocode for the start node, terminal nodes and nodes containing an ⟨*exit-statement*⟩ is modified to include the loop statement.

[5] In the context of a ⟨*statement-list*⟩ an ⟨*exit-statement*⟩ is treated like any other simple statement.

Next we consider the TIG's of select statement, which allow a task to initiate a rendezvous with one of several different tasks. The selecting task chooses (nondeterministically) from among those tasks that are available to rendezvous with it. If there is no task waiting to establish a rendezvous corresponding to one of the choices, then that choice is ignored when deciding which task to rendezvous with. If there is no task waiting to establish a rendezvous corresponding to any of the choices, then the selecting task is delayed until there is.

This section considers the three general forms of the select statement. The first form to be considered is:

select ⟨*accept-statement-list*⟩ {or ⟨*accept-statement-list*⟩} end select;

An ⟨*accept-statement-list*⟩ is a list of statements, the first of which is an ⟨*accept-statement*⟩. A select statement contains one or more ⟨*accept-statement-list*⟩'s separated by the keyword *or*. The TIG for this select statement is constructed by combining the start nodes of the ⟨*accept-statement-list*⟩'s into a single new start node *s*. All the edges leaving *s* are placed into a single edge group.

For the second form of a select statement consider the case where one of the select alternatives is a terminate statement. Without loss of generality we can assume that the terminate statement is the last select alternative. Thus, the second form to be considered is:

select ⟨*accept-statement-list*⟩ {or ⟨*accept-statement-list*⟩ } or terminate; end select;

The only differences between this example and the previous example are that the start node is added to the set of terminal nodes and the pseudocode for the start node has the terminate alternative added.

And finally we consider the third form of a select statement.

select ⟨*accept-statement-list*⟩ {or ⟨*accept-statement-list*⟩} else ⟨*statement-list*⟩ end select;

The TIG for this statement is constructed in the same way as before except that the else clause is incorporated into the start node.

2.1.4 Tasks

This section describes a set of rules that translate program statements into the corresponding task interaction graphs. These translation rules can be used in conjunction with a bottom-up parsing technique to build a translator that translates any statement into a task interaction graph. As with any bottom-up technique, the TIG of any compound statement is constructed from the TIG's of the statements embedded in the statement.

The task interaction graph for a task is constructed from the task interaction graph of the statements in the body of the task by adding the task header and declarations and a default ENTER pseudostatement to the pseudocode for the start node and *"end task_name;"* and a default EXIT pseudostatement to the pseudocode terminal nodes. The pseudocode for the start node has the pseudostatement ENTER(TASK_ACTIVATE) added at the beginning followed by the task header. This indicates that this region is entered when the task is activated. The pseudocode for each terminal node where the task can finish execution has the pseudostatement EXIT(TASK_TERMINATE,ϕ) added after the *"end task_name;"*. This indicates that the region is exited when the task terminates. Terminal nodes that terminate solely because of a terminate alternative in a select statement do not need this EXIT pseudostatement.

2.2 An Example

This section looks at a task interaction graph for a more complicated example than has been seen so far. Figure 8 shows the skeleton of a task based on an example in [Tayl83b]. (The other tasks in this example are discussed in Section 3.1.)

The TIG's for the accept statements are constructed as shown in Section 2.1. The first step considered here is to construct the TIG for statement 2, the select statement. The completed TIG for this statement is shown in Figure 9 and the pseudocode for this TIG is shown in Figure 10. This TIG is constructed from the TIG's of the ⟨*accept-statement-list*⟩, consisting of statement 3 and the ⟨*accept-statement-list*⟩ consisting of statement 4. Each of these TIG's has three nodes. To construct the new TIG, the start nodes of these two TIG's are combined into a new start node 1. The edges leaving this new node are put into a single edge group.

```
        task body T1 is
            DONE: boolean;
        begin
1           loop
2               select
3                   accept P;
                or
4                   accept Q;
                end select;
5               exit when DONE;
            end loop;
        end T1;
```

Figure 8: The Task T1

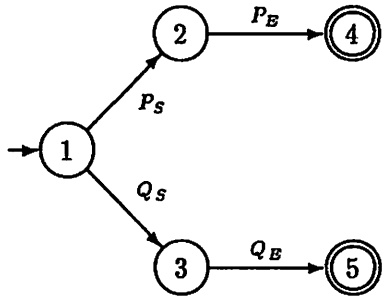

Figure 9: TIG for the select statement

```
C(1) = select
           EXIT(ACCEPT_START(P),2);
       or
           EXIT(ACCEPT_START(Q),3);
       end select;

C(2) = ENTER(ACCEPT_START(P));
       EXIT(ACCEPT_END(P),4);

C(3) = ENTER(ACCEPT_START(Q));
       EXIT(ACCEPT_END(Q),5);

C(4) = ENTER(ACCEPT_END(P));

C(5) = ENTER(ACCEPT_END(Q)));
```

Figure 10: Pseudocode for the select statement

The next step is to construct the TIG for statement 1, the loop statement, from the TIG for statements 2 and 5. This involves adding new edges from the terminal nodes 4 and 5 to node 2 and to node 3. The labels on edges (4,2) and (5,2) are the same as the label on edge (1,2) and the labels on edges (4,3) and (5,3) are the same as the label on edge (1,3). Figure 12 shows how the pseudocode for the start node is combined with the pseudocode for nodes 4 and 5 to create the new pseudocode for these nodes.

The final step is to add the task header and default ENTER and EXIT pseudostatements to the TIG for statement 1 to create the TIG for the entire task. The completed task interaction graph for task T1 is shown in Figure 11. The pseudocode for the nodes of this graph is shown in Figure 12. The edge groups for this example are $\{(4,2),(4,3)\}$, $\{(1,2),(1,3)\}$, $\{(5,2),(5,3)\}$, $\{(2,4)\}$, and $\{(3,5)\}$.

This task 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, 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

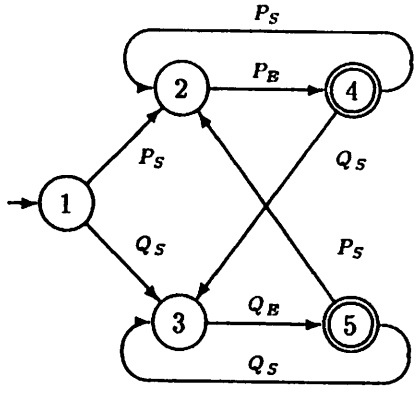

Figure 11: TIG for task T1

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 are entered in the middle of a loop. After entering node 4, the loop is exited or the end of the loop is reached causing a return to the beginning of the loop, and finally the select statement is encountered causing the task to exit the region. Node 5 is similar except it is entered after the end of the Q accept instead of the P accept.

```
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;
       end T1;

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 12: Pseudocode for task T1

3. Task Interaction Concurrency Graphs

A task interaction graph is a representation of the behavior of a single task in a concurrent program. The behavior of the entire program is represented by a task interaction concurrency graph. Static concurrency analysis is suited to situations where there are a fixed number of tasks, k, which are denoted $Task_1, Task_2, \ldots, Task_k$. The TIG for $Task_i$ is denoted $G_i = (N_i, E_i, s_i, T_i, L_i, C_i)$.

The concurrency graph of a program is built from the TIG's of the tasks that make up the program. A vertex of a concurrency graph, which is known as a *state* of the concurrency graph,[6] 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 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 $l \neq i, j$, $n_l = m_l$, 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 is similar to that given in [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 will be an edge in the concurrency graph 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

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

```
task body PHIL1 is
begin
   loop
      THINKING;
      FORK1.UP;
      FORK2.UP;
      EATING;
      FORK1.DOWN;
      FORK2.DOWN;
   end loop;
end PHIL1;
```

```
task body PHIL2 is
begin
   loop
      THINKING;
      FORK2.UP;
      FORK1.UP;
      EATING;
      FORK2.DOWN;
      FORK1.DOWN;
   end loop;
end PHIL2;
```

```
task body FORK1 is
begin
   loop
      accept UP;
      accept DOWN;
   end loop;
end FORK1;
```

```
task body FORK2 is
begin
   loop
      accept UP;
      accept DOWN;
   end loop;
end FORK2;
```

Figure 13: Philosopher and Fork Tasks

entry and it is assumed that for each entry call at least one of these orderings will result in the run time scheduler choosing that entry call for the next rendezvous.

Section 3.1 contains an example of a concurrency graph and Section 3.2 shows how to construct concurrency graphs from task interaction graphs.

3.1 Examples

This section presents the concurrency graph for the following variation on the Dining Philosophers Problem. Two Philosophers sit a table alternately eating and thinking. Unfortunately, they have only two forks between them and each philosopher needs to pick up both forks in order to eat. When they are finished eating both forks are put down. This situation is modeled by the tasks PHIL1, PHIL2, FORK1, and FORK2, one for each philosopher and one for each fork. These tasks are shown in Figure 13. The task interaction graphs for these tasks are shown in Figure 14. The tasks FORK1 and FORK2 are abbreviated $F1$ and $F2$ and the accept statements UP and DOWN are abbreviated U and D. The pseudocode for the task PHIL1 is shown in Figure 15. The pseudocode for the other tasks is similar.

The concurrency graph for the dining philosophers is shown in Figure 16. Each state in this

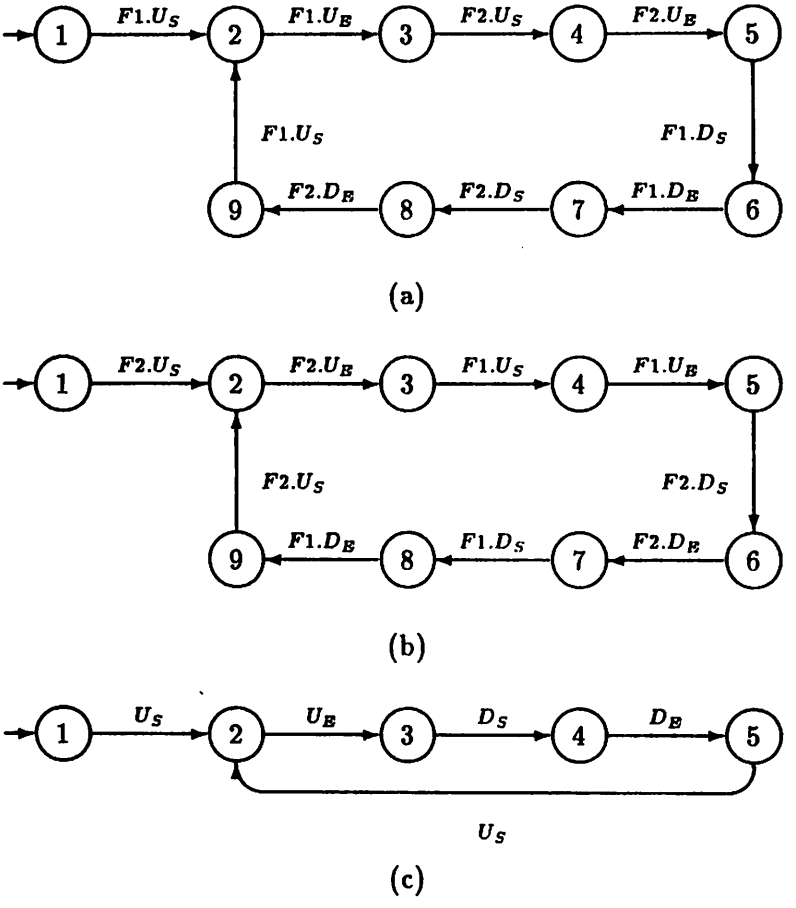

Figure 14: TIG's for (a) PHIL1 (b) PHIL2 (c) FORK1 and FORK2

```
C(1) = ENTER(TASK_ACTIVATE);
       task body PHIL1 is
       begin
          loop
             THINKING;
             EXIT(CALL_START(FORK1.UP),2);
             ...
          end loop;

C(2) = ENTER(CALL_START(FORK1.UP));
       EXIT(CALL_END(FORK1.UP),3);

C(3) = ENTER(CALL_END(FORK1.UP));
       EXIT(CALL_START(FORK2.UP),4);

C(4) = ENTER(CALL_START(FORK2.UP));
       EXIT(CALL_END(FORK2.UP),5);

C(5) = ENTER(CALL_END(FORK2.UP));
       EATING;
       EXIT(CALL_START(FORK1.DOWN),6);

C(6) = ENTER(CALL_START(FORK1.DOWN));
       EXIT(CALL_END(FORK1.DOWN),7);

C(7) = ENTER(CALL_END(FORK1.DOWN));
       EXIT(CALL_START(FORK2.DOWN),8);

C(8) = ENTER(CALL_START(FORK2.DOWN));
       EXIT(CALL_END(FORK2.DOWN),9);

C(9) = loop
          THINKING;
          EXIT(CALL_START(FORK1.UP),2);
          ...
          ENTER(CALL_END(FORK2.DOWN));
       end loop;
```

Figure 15: Pseudocode for PHIL1

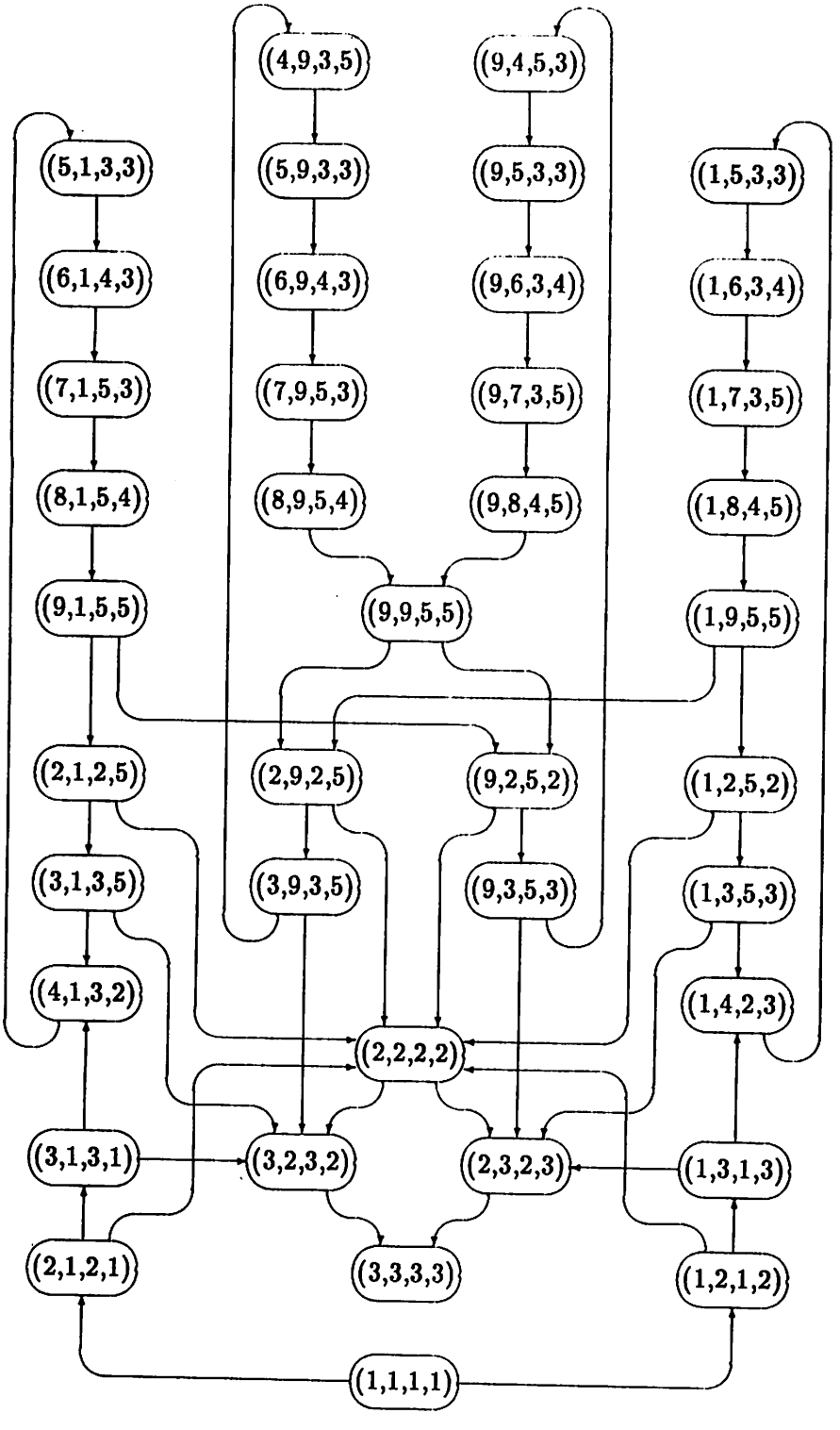

Figure 16: Concurrency Graph for the Dining Philosophers

graph is represented by a tuple (PHIL1,PHIL2,FORK1,FORK2). The starting state for this example is $(1,1,1,1)$. There is an edge from state $(1,1,1,1)$ to state $(2,1,2,1)$ because there is an edge (1,2) in the TIG for PHIL1 with label $F1.U_S$ and edge (1,2) in the TIG for FORK1 with label U_S. This edge in the concurrency graph represents the start of a rendezvous between tasks PHIL1 and FORK1. Similarly there is an edge from $(1,1,1,1)$ to $(1,2,1,2)$ representing the start of a rendezvous between PHIL2 and FORK2. The other edges in the concurrency graph likewise represents the starts and ends of rendezvous between pairs of tasks. Note that only states that represent valid synchronization states of the tasks are represented in the concurrency graph. In general, the valid states are those that are reachable from the starting state $(s_1, s_2, \ldots, s_k)$. In this example, deadlock is easy to detect and occurs at state (3,3,3,3) because this state has no edges leading out of it. Section 3.3 contains a general discussion of deadlock detection.

As another example, consider the concurrent program whose outline is shown in Figure 17. The task interaction graphs for the main task, task T1, and task T2 are shown in Figure 18. The pseudocode for the tasks MAIN and T2 are shown in Figure 19. The pseudocode for task T1 is shown in Figure 12. The task interaction concurrency graph for this example is shown in Figure 20. This concurrency graph contains just 9 states and 8 edges; Section 4.3 shows how to reduce this to just 5 states and 4 edges.

3.2 Constructing Task Interaction Concurrency Graphs

Concurrency graphs are 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 concurrency graphs from flow graphs was first described by Taylor in [Tayl83b]. This section describes a version of the algorithm that can be used to construct concurrency graphs from task interaction graphs.

The basic algorithm is quite simple. Each state of the partially constructed concurrency graph is checked to find its adjacent states. For each adjacent state the 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

```
procedure MAIN is
begin
    SUBR;
end MAIN;
```

```
procedure SUBR is
    task body T1 is
        DONE: boolean;
    begin
        loop
            select
                accept P;
            or
                accept Q;
            end select;
            exit when DONE;
        end loop;
    end T1;
    task body T2 is
    begin
        T1.P;
    end T2;
begin
    T1.Q;
end SUBR;
```

Figure 17: A Concurrent Program

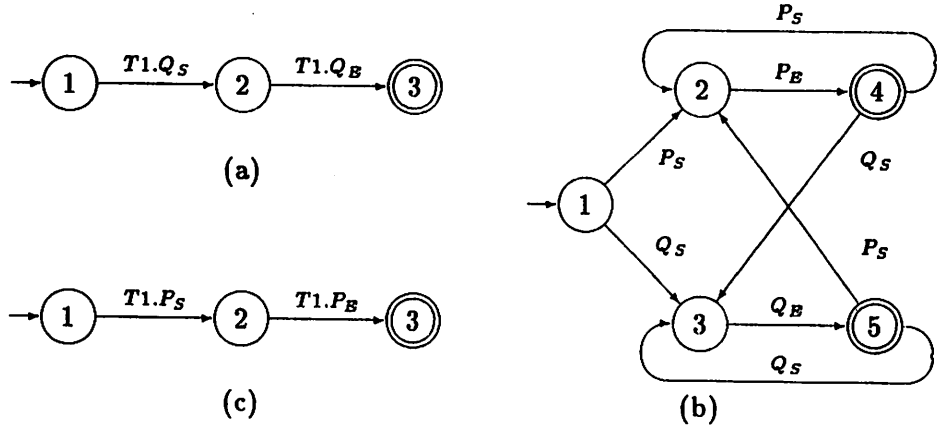

Figure 18: TIG's for (a) MAIN (b) T1 and (c) T2

```
C(1) = procedure MAIN is
       begin
           SUBR;
           procedure SUBR is
               task body T1 is ...end T1;
               task body T2 is ...end T2;
           begin
               EXIT(CALL_START(T1.Q),2);
C(2) =         ENTER(CALL_START(T1.Q));
               EXIT(CALL_END(T1.Q),3);
C(3) =         ENTER(CALL_END(T1.Q));
           end SUBR;
       end MAIN;
```

(a)

```
C(1) = ENTER(TASK_ACTIVATE);
       task body T2 is
       begin
           EXIT(CALL_START(T1.P),2);
C(2) =     ENTER(CALL_START(T1.P));
           EXIT(CALL_END(T1.P),3);
C(3) =     ENTER(CALL_END(T1.P));
       end T2;
       EXIT(TASK_TERMINATE,φ);
```

(b)

Figure 19: Pseudocode for (a) MAIN and (b) T2

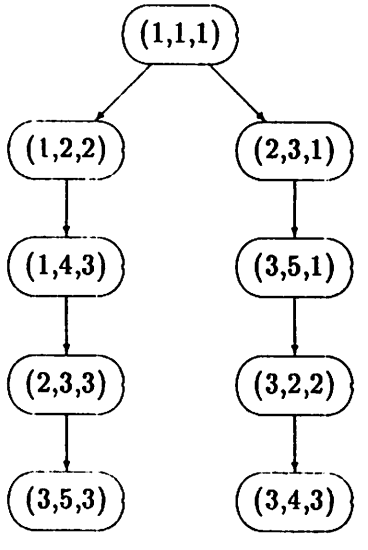

Figure 20: A Task Interaction Concurrency Graph

```
TO_BE_CHECKED := {(s_1, s_2, ..., s_k)};
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 21: Concurrency Graph Construction Algorithm

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 checked. 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.

It is also possible to check each state for deadlock as it is being checked to find its neighbors by applying a few simple tests. These tests are described in the next section.

The complete algorithm is shown in Figure 21. 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 concurrency graph. When the algorithm terminates the concurrency graph will consist of the states in the set OLD_STATES and edges in the set EDGES.

3.3 Deadlock Detection

Deadlock occurs when a task waits for a rendezvous that can never occur. Deadlock can be detected by applying an additional test as each state is checked. 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.

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. It might be the case 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. It might be the case that $Task_j$ executes a path through its current region that leads to different task interaction (or causes the task to terminate). The task 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. Rules (*i*) and (*ii*) are used to detect these situations.

(*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.

The next two conditions have to do with accepts. $Task_i$ might be waiting to start (or end) a accept P and there may be no task that is able to start (or end) a corresponding call. This can occur for the same two reasons. Either no task is in a region that contains a start (or end) of a call to accept P of $Task_i$, or all other tasks can end up waiting for some other interaction to occur. Rules (*iii*) and (*iv*) are used to detect these situations.

(*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 $Task_i.P_E$.

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 $(n_i, m_i') \in E_i$. Thus, for task interaction concurrency graphs, deadlock can occur at states that have edges leading out of them.

3.4 Task Interaction Concurrency Graphs vs Control Flow Concurrency Graphs

The task interaction graph approach to concurrency analysis offers a number of advantages over the reduced flow graph approach of [Tayl83b]. The most important advantage is that TIG's are often smaller (and never larger) than the corresponding reduced flow graph. I.e., a TIG will have fewer nodes than the corresponding reduced flow graph for the same task. The primary advantage of this more compact representation is that task interaction concurrency graphs can be much smaller than the corresponding control flow concurrency graphs. Another advantage is that TIG's are a better representation of the ways a task interacts with other tasks since task interactions are emphasized over control flow. In comparison, control flow graphs emphasize control flow over task interaction. Finally, TIG's include an explicit representation of the non-tasking parts of a task. For example, for the state $(n_1, n_2, \ldots, n_k)$ this representation is $(C_1(n_1), C_2(n_2), \ldots, C_k(n_k))$. In contrast, much of the information about the non-tasking parts of of a task is lost when the flow graph is reduced.

```
                                        task body TASK2 is
     task body TASK1 is                     U : integer;
     begin                              begin
1        V := 1;                    2        V := 2;
     end TASK1;                         3        U := V;
                                        end TASK2;
```

Figure 22: Two tasks with a shared variable V

4. Refinements to the Model

This section discusses several enhancements and refinements to the task interaction graph model. The first part of this section shows how to incorporate shared variables into task interaction graphs. The next part discusses how procedures can be incorporated into the model. The following part shows how, in special cases, concurrency graphs can be be made simpler and this section concludes with a discussion of various aspects of TIG's, particularly pseudocode.

4.1 Shared Variables

The use of shared variables greatly complicates the behavior of a program.[7] For example, consider the two tasks shown in Figure 22. The first task assigns the value 1 to the shared variable V. The second task assigns the value 2 to V and then assigns the value of V to U. This program has three possible outcomes depending on the order in which the statements are executed. For example, if statement 1 is executed before statements 2 and 3 then V and U will both end up with the value 2. If statement 1 is executed after statement 2 and before statement 3 then V and U will both end up with the value 1. If statement 1 is executed after statements 2 and 3 then V will end up with the value 1 and U will end up with the value 2.

The problem with the use of shared variables is that there is no way to know how the execution of the statements in the different tasks is going to be interleaved. The solution to this problem is to recognize that the use of shared variables is another means of task interaction and that these interactions divide tasks into regions in the same way that calls and accepts do. The interactions

[7] Note that the use of shared variables described here is more general than is allowed in Ada.

```
task body SHARED_V is
   V : integer;
begin
   loop
      select
         accept PUT(VVAL: in integer) do
            V := VVAL;
         end PUT;
      or
         accept GET(VVAL: out integer) do
            VVAL := V;
         end GET;
      or
         terminate;
      end select;
   end loop;
end SHARED_V;
```

Figure 23: The SHARED_V Task

between these regions can be analyzed to determine how the program behaves. Two ways to show how shared variables divide tasks into regions are considered. The first technique translates a program that uses shared variables into an equivalent program that does not. This is done by creating a new task for each shared variable whose purpose is to keep track of the value of that shared variable. Each occurrence of the shared variable is replaced with a local variable and a call to the new task. This new task keeps track of the value of the shared variable using two accepts, one to assign a new value to the variable and one to return the value of the variable. It is assumed that references to and definitions of shared variables are indivisible operations.

An example of a task to keep track of the value of the shared variable V for the example of Figure 22 is shown in Figure 23. In this task the variable V is local to the task. Figure 24 shows how the original two tasks would be modified to interact with the new task. In each of these tasks the shared variable V is replaced with a local variable V. Whenever this variable is given a new value, as in the statements 1 and 2 of TASK1 and TASK2, the value of this local variable is made available to the other task through a call to SHARED_V.PUT. Whenever this value is referenced, as in statement 3 to TASK2, the value is determined by a call to SHARED_V.GET. This new

```
task body TASK1 is
    V : integer;
begin
    V := 1;
    SHARED_V.PUT(V);
end TASK1;
```

```
task body TASK2 is
    V,U : integer;
begin
    V := 2;
    SHARED_V.PUT(V);
    SHARED_V.GET(V);
    U := V;
end TASK2;
```

Figure 24: Two tasks using SHARED_V

set of three tasks exhibits exactly the same behavior as the original two tasks and has the added advantage that it can now be analyzed using the techniques developed in the first part of this report.

This technique for analyzing programs with shared variables is not without cost however. For example, the TIG for TASK1 would have 3 nodes, the TIG's for TASK2 and SHARED_V would each have 5 nodes. The concurrency graph for this example would have 14 states. The problem with this approach is that each entry call for each use of a shared variable is represented by three regions and each shared variable requires a task which has five regions.

Fortunately, it is possible to model each use of each shared variable with two regions and no additional tasks. The second technique considered here for handling shared variables modifies task interaction graphs to model their uses directly. This technique results in smaller task interaction graphs that more closely model the task interactions caused by the use of shared variables than the first technique. This is made possible by observing that a use of a shared variable divides a task into *two* regions instead of three. Exactly where the task is divided depends on whether the variable is being assigned a new value or its value is being referenced. If the variable is being assigned a new value then the division between the two regions is immediately after the statement in which the assignment occurs. If the variable is referenced, then the division between the two regions is immediately before the statement in which the reference occurs.[8] It would be more accurate to consider that the assignment or reference occurs along the edge between the two regions as is done

[8]Further refinement of this model would be necessary for expressions that contain more than one reference to a shared variable.

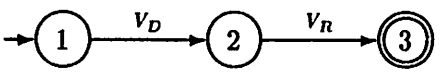

Figure 25: TIG for TASK2

```
C(1) = ENTER(TASK_ACTIVATE);
       task body TASK2 is
           V,U : integer;
       begin
           V := 2;
           EXIT(SHARED_DEF(V),2);

C(2) =     ENTER(SHARED_DEF(V));
           EXIT(SHARED_REF(V),3);

C(3) =     ENTER(SHARED_REF(V));
           U := V;
       end TASK2;
       EXIT(TASK_TERMINATE,φ);
```

Figure 26: Pseudocode for TASK2

for calls and accepts, but it is more convenient to keep the entire statement in which the variable is used in a single region. The choice of region was made to be analogous to the first technique where the PUT occurred after a statement in which a new value was assigned to the shared variable, but the GET occurred before the statement in which the shared variable was referenced. Figure 25 shows the TIG for TASK2 using this new model of shared variable task interaction. The pseudocode for the nodes of this graph is shown in Figure 26.

The rules for creating TIG's for statements that contain shared variables can be added to the translation rules developed in Section 2.1. The first rule to consider is for translating a ⟨*shared-assignment*⟩, a statement that contains an assignment to a shared variable. The TIG for

a ⟨*shared-assignment*⟩ consists of two nodes connected by a single edge. Assume that the shared variable is V. The TIG for this statement is defined as follows.

$$(\{s,t\},\{(s,t)\},s,\{t\},L,C)$$

where

$L(s,t)$ = V_D
$C(s)$ = ⟨*shared-assignment*⟩
EXIT(SHARED_DEF(V),t);
$C(t)$ = ENTER(SHARED_DEF(V));

Note that the code for the ⟨*shared-assignment*⟩ is in the first region.

The second rule is for a ⟨*shared-reference*⟩, a simple statement that contains a reference to a shared variable. The TIG for a ⟨*shared-reference*⟩ is similar to the TIG for a ⟨*shared-assignment*⟩ except that the code for the ⟨*shared-reference*⟩ is in the second region. N, E, S and T are the same as for a ⟨*shared-assignment*⟩. L and C are given below.

$L(s,t)$ = V_R
$C(s)$ = EXIT(SHARED_REF(V),t);
$C(t)$ = ENTER(SHARED_REF(V));
⟨*shared-reference*⟩

The introduction of this new type of interaction requires a new rule for constructing edges between states in a concurrency graph. (This new rule is in addition to the old rule described in Section 3.) 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 such that for $l \neq i$, $n_l = m_l$, and

(*i*) $(n_i,m_i) \in E_i$ and $L_i(n_i,m_i) = S_D$, or

(*ii*) $(n_i,m_i) \in E_i$ and $L_i(n_i,m_i) = S_R$.

Using this definition of TIG's the TIG for TASK1 has two regions, the TIG for TASK2 has three regions and there is no need for the third task. The concurrency graph constructed from these two TIG's has six states, and is shown in Figure 27. The three paths through this concurrency graph

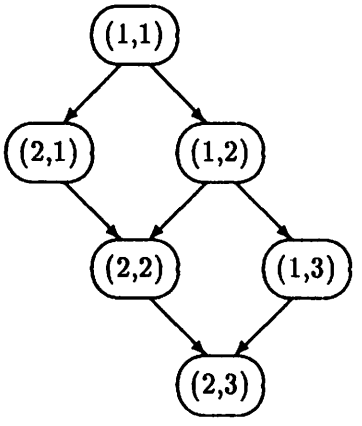

Figure 27: Concurrency Graph for TASK1 and TASK2

correspond to the three possible outcomes of the program. Even though this is an improvement over the first method, the use of shared variables in a concurrent program can add considerable complexity to the concurrency graph and so they should be used with caution.

There is also another technique that is based on the technique used in [Tayl83b]. This is to build the concurrency graph without paying attention to shared variables and then look for tasks that are in regions that can access the same shared variable. In many cases, the state information will be sufficient to answer the desired questions about the uses of the shared variable. This approach would result in a smaller concurrency graph. However, it may not always be possible or easy to calculate the desired answer about the shared variable. The choice of technique will depend on the type of analysis that is being applied.

4.2 Procedures

The calling of or returning from procedures are not, by themselves, task interactions. Of course, procedures may contain task interactions within their declarative regions or bodies.

The use of a procedure within a task may or may not be related to the structure of the task

interactions for the task. There are three cases to consider. In the first case, the called procedure contains no task interactions, i.e., it does not declare any tasks within its declarative part and contains no entry calls or accepts within the procedure body. A call to such a procedure is a ⟨*non-tasking-statement*⟩ so the TIG for a procedure call is a single node and the pseudocode for this node is the procedure call. As is the case with all ⟨*non-tasking-statement*⟩'s, this node represents only part of the region and will be merged with nodes from the preceding and succeeding statements to form the node for the entire region in which the procedure call is made.

In the second case, the called procedure interacts with tasks that are active only while the procedure is active. These tasks are declared within the declarative part of the task and any entry calls or accepts within the body of the procedure are to or from these tasks only. In this case, the procedure and the tasks within it are a self-contained unit that can be analyzed separately from the rest of the program and a concurrency graph for just these tasks can be constructed. Taking advantage of this modularity will simplify the overall analysis. Therefore, a call to such a procedure is treated as if it is a ⟨*non-tasking-statement*⟩.

In the third case, which is the most general, the procedure does not provide any modularity, at least as far as the structure of the task interactions are concerned. The procedure may make entry calls to and accepts from any task. The approach taken in this report to calls to such procedures is to incorporate the TIG of the called procedure into the TIG of the calling task at the point of each procedure call. The TIG of a procedure call is constructed from the TIG of the called procedure by merging the call onto the beginning of start node. If G_1 is the TIG of the called procedure then the TIG of the procedure call is

$$
\begin{aligned}
N &= N_1 \\
E &= E_1 \\
S &= s_1 \\
T &= T_1 \\
L &= L_1 \\
C(n) &= \begin{cases} \text{procedure\_name}(\langle \textit{parameter-list} \rangle) \| C_1(n) & \text{if } n = s_1 \\ C_1(n) & \text{otherwise} \end{cases}
\end{aligned}
$$

Even though this results in a copy of the TIG of the called task at each place that the procedure

is called, this does not mean that analysis of the procedure must be repeated for each of these copies. Many types of analysis will only need to be applied to the procedure once and the results used for each copy. This preserves some of the advantages of modularity provided by procedures.

The task interaction graph for a procedure is constructed from the task interaction graph of the statements in the body of the procedure by adding the procedure header and declarations to the pseudocode for the start node and *"end procedure_name;"* to the pseudocode for the terminal nodes. These additions to the pseudocode do not change the TIG in any significant way and serve only to mark the beginning and end of the procedure.

4.3 Ignoring Intermediate 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 task interaction graph can be reduced the corresponding concurrency graph 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 calls and accepts. Unfortunately at least three nodes are needed to model the general case of a 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 a 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 dining philosophers problem of Section 3.1 will be used to illustrate this simplification of concurrency graphs. Figure 28 shows the simplified TIG's for the tasks in this example. In each

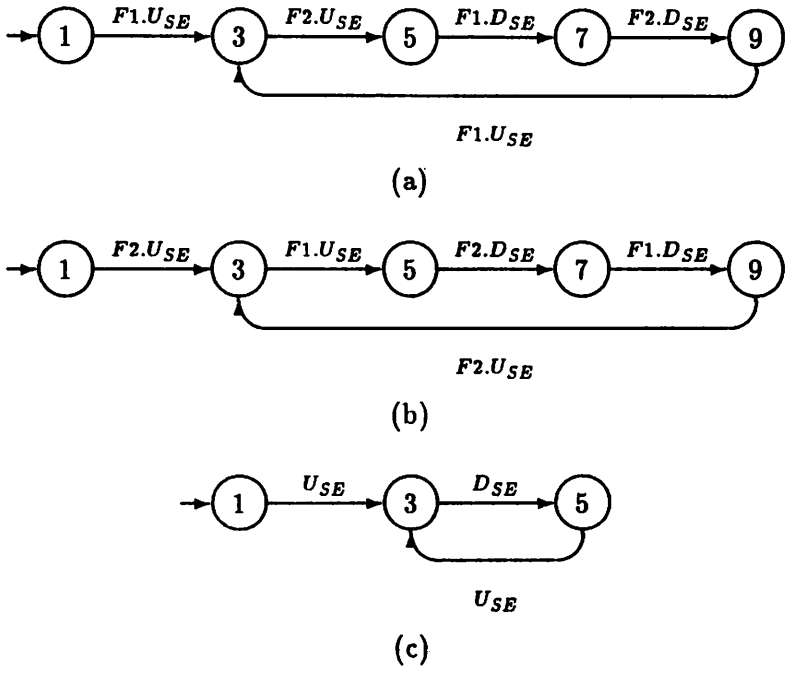

Figure 28: TIG for (a) PHIL1 (b) PHIL2 and (c) FORK1 and FORK2

TIG the intermediate nodes (2,4,6, and 8) have been eliminated. The edges representing the start and end of the calls and accepts have been replaced with a single edge representing the entire call or accept. The subscript *SE* is used in the label on such an edge. The pseudocode for these nodes is almost the same as in the original example except that the interactions are CALL_START_END and ACCEPT_START_END to indicate they represent the entire call or accept. The pseudocode for PHIL1 is shown in Figure 29. The pseudocode for the other tasks is similar. The new concurrency graph for this example is given in Figure 30 and is half the size of the original concurrency graph.

And finally, we return to the example from [Tayl83b] shown in Figure 17. Since neither of the accepts in this example have bodies the concurrency graph for this example can also be simplified. The new concurrency graph is shown in Figure 31.

```
C(1) = ENTER(TASK_ACTIVATE);
       task body PHIL1 is
       begin
           loop
               THINKING;
               EXIT(CALL_START_END(FORK1.UP),3);
               ...
           end loop;

C(3) = ENTER(CALL_START_END(FORK1.UP));
       EXIT(CALL_START_END(FORK2.UP),5);

C(5) = ENTER(CALL_START_END(FORK2.UP));
       EATING;
       EXIT(CALL_START_END(FORK1.DOWN),7);

C(7) = ENTER(CALL_START_END(FORK1.DOWN));
       EXIT(CALL_START_END(FORK2.DOWN),9);

C(9) = loop
           THINKING;
           EXIT(CALL_START_END(FORK1.UP),3);
           ...
           ENTER(CALL_START_END(FORK2.DOWN));
       end loop;
```

Figure 29: Pseudocode for PHIL1

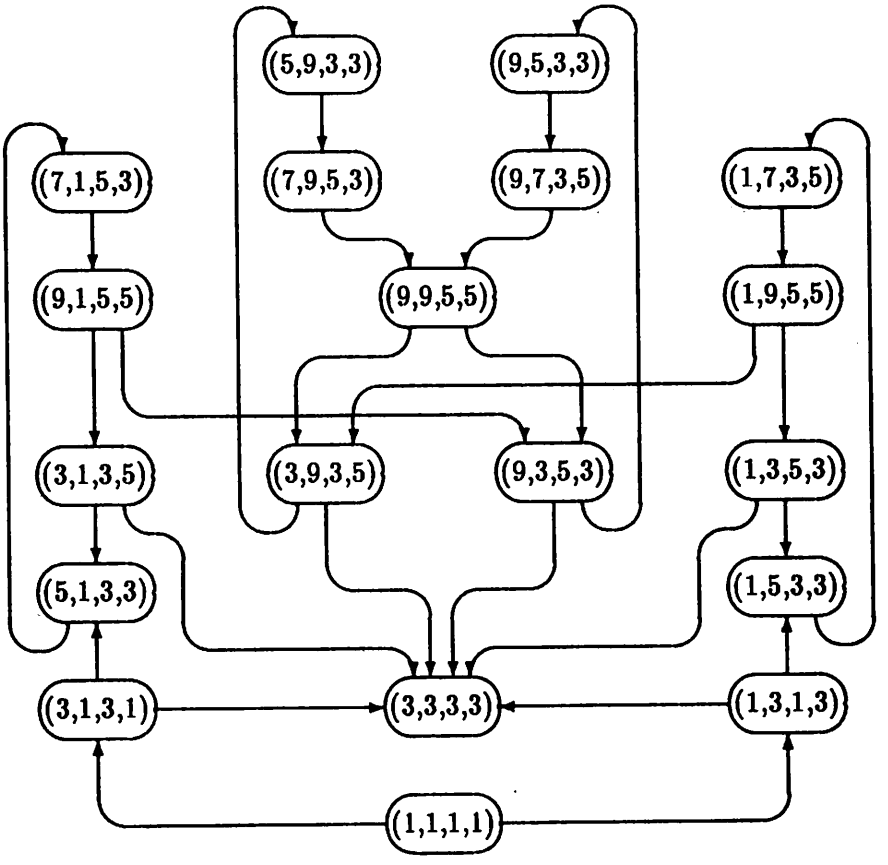

Figure 30: Simplified Concurrency Graph for the Dining Philosophers

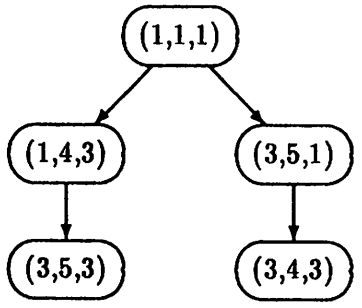

Figure 31: A Simplified Concurrency Graph

4.4 Other Issues

This section concludes with a few comments on task regions and pseudocode.

Task regions, as they have been developed in this report, are the natural embodiment of the behavior of interacting tasks. An individual task region encompasses all possible behaviors of a task starting at one task interaction and ending at the next interaction. Regions contain only one entry point, corresponding to the starting interaction, but may contain many exit points, each reachable by a different path through the region. This single entrance, multiple exit characteristic is a consequence of the desire for task regions to be maximal, in the sense that they contain everything except direct task interactions. One of the benefits of this definition is smaller concurrency graphs.

Task interaction graphs represent tasks using two types of representation. Task interactions are represented using nodes and edges and everything else is represented using pseudocode. It is important to distinguish between characteristics that are properties of one or the other of these representations. Undesirable characteristics of a particular choice of pseudocode can be eliminated by choosing different pseudocode. The underlying model outlined in this report would be unaffected by this change.

For example, the pseudocode described in this report preserves as much of the syntactic structure of the original code as possible in order to show the relationship between the two representations.

In some cases, this was done to avoid lengthy discussions of less important details. For example, ENTER pseudostatements were placed in the middle of loops to preserve the syntactic structure of the loop in order to simplify the discussion of exit statements. If task interaction graphs were to be used in a situation where the entrance in the middle of code caused a problem with the analysis, the loops could be easily unrolled to place the ENTER pseudostatement at the beginning of the region. However, some sort of accommodation would have to be made to handle exit statements. There are several cases that would need to be considered depending on the possible locations of the exit statement. The resulting pseudocode would be equivalent to the pseudocode discussed in this report, but might not be as close syntactically to the original code. A change from one pseudocode to the other would have no effect on the structure of a task and therefore the task interaction graphs would be unchanged.

Some statements in a program are represented in the pseudocode of more than one node. There are two reasons that this occurs. First, it might reflect the different circumstances in which the execution of these statements may occur. For example, in the example in Figure 1 the statement z := 4; is found in the pseudocode of two nodes. Second, code duplication might occur in situations where the execution of the statement might be split across several regions. For example, a loop statement is usually represented in at least two nodes. One of these regions represents the first execution of the loop and contains the code for statements that are executed prior to the first iteration of the loop. Node 1 of Figure 12 is an example of this type of node. A node that represents subsequent executions of the loop would not contain code for the statements that are executed before the first iteration of the loop. Nodes 4 and 5 of Figure 12 are examples of this type of node.

The replication of these statements in different nodes of a TIG separates out the different uses of the statements. It would appear that this approach might require multiple copies of large sections of code. In practice, it should not be necessary to create duplicate copies of large sections of code since pointers to a single copy should be sufficient in most cases. The same technique might also be applied to avoid multiple copies of code for procedures.

5. Conclusion

This report introduces task interaction graphs as a tool for analysis of concurrent programs. Task interaction graphs model tasks using task interactions and task regions as the principal parts. Task interactions are intertask communications that change the local environment of the involved tasks. Two types of task interactions discussed in this report are rendezvous and the use of shared variables.

Task interaction concurrency graphs are constructed from task interaction graphs and offer several advantages over control flow concurrency graphs. Task interaction concurrency graphs are smaller than control flow concurrency graphs and contain much the same information. The more compact representation will make possible the construction of concurrency graphs for larger programs than would otherwise be feasible. Another advantage of task interaction concurrency graphs is that there is an explicit representation of exactly what each state of the concurrency graph represents. In this state, the behavior of each task is described by this pseudocode and is independent of the other tasks.

The treatment of procedures presented in Section 4.2 is essentially the same as in [Tayl83b] except that the start and end of a procedure is not considered to be a task interaction and the copying of the representation of the procedure has been made explicit.

The rules for deadlock detection are more complicated for task interaction graphs than for control flow concurrency graphs. However, the amount of work necessary to apply the rules for deadlock detection to a state is no more than that required to find the adjacent states of that state. The advantages of task interaction concurrency graphs outweigh the small disadvantage of this small amount of additional work.

Task regions are maximal parts of a task that are uninterrupted by task interactions. Since task interactions occur only at region boundaries, a region is unaffected by other tasks and can be considered to have a constant local environment with respect to the other tasks. This makes task regions ideal for using as a basis for extending sequential analysis techniques to concurrent programs. Sequential techniques can be used to analyze each region independently from other regions and other tasks. The results of the analyses of the regions, together with the information

about how the regions interact, can be used for further analysis of the entire program. This general approach will be more tractable than one that uses smaller parts of a task as the basis for the sequential analysis or one that ignores region boundaries altogether.

Task interaction graphs should have a wide variety of applications. One possible application, which is described in [Youn86] using flow graphs, is to combine static concurrency analysis and symbolic evaluation techniques to prune the concurrency graph. The use of task interaction graphs would let the symbolic evaluation be applied to an entire region at a time, offering a substantial improvement over the use of a symbolic execution graph that would contain many more states.

Another application, which will be the subject of future work, is to develop path selection techniques for concurrent programs based on data flow path selection techniques. Existing data flow path selection techniques can be used within regions. Interregional data flow analysis techniques, similar to interprocedural data flow analysis, will also need to be developed.

Future work will also include further refinements to the task interaction graph model such as how to handle multiple references to shared variables in an expression, how to handle calls to functions that cause task interactions, better ways to handle procedure calls, how to handle delay and abort statements, and how to handle nested tasks.

Appendix

Throughout this appendix, G_i will be used to represent the TIG, $(N_i, E_i, s_i, T_i, L_i, C_i)$. Also, whenever two TIG's G_1 and G_2 are combined it will be assumed that $N_1 \cap N_2 = \emptyset$. Unless explicitly mentioned otherwise each edge belongs to its own unique edge group.

The TIG of a $\langle$*statement-list*$\rangle$ consisting of two statements such as $\langle$*statement*$\rangle_1 \langle$*statement*$\rangle_2$ can be constructed from their TIG's, G_1 and G_2.

Let $T_1' \subseteq T_1$ be the set of terminal nodes t, such that control can pass from the part of $\langle$*statement*$\rangle_1$ represented by t to $\langle$*statement*$\rangle_2$. The new TIG is constructed by merging a copy of s_2 onto each terminal node $t \in T_1'$ and adding an edge (t, n) for each edge (s_2, n). Nodes are merged by concatenating the pseudocode for the second node onto the end of the pseudocode for the first node. The original node s_2 and edges (s_2, n) are deleted from the graph. The new graph is defined as follows:

$$
\begin{aligned}
N &= N_1 \cup N_2 - \{s_2\} \\
E &= E_1 \cup (E_2 - \{(s_2, n)\}) \cup \{(t, n) \mid t \in T_1' \text{ and } (s_2, n) \in E_2\} \\
S &= s_1 \\
T &= \begin{cases} (T_2 - s_2) \cup T_1 & \text{if } s_2 \in T_2 \\ T_2 \cup (T_1 - T_1') & \text{otherwise} \end{cases} \\
L(n, m) &= \begin{cases} L_1(n, m) & \text{if } (n, m) \in E_1 \\ L_2(n, m) & \text{if } (n, m) \in E_2 - \{(s_2, n)\} \\ L_2(s_2, m) & \text{if } n \in T_1' \text{ and } (s_2, m) \in E_2 \end{cases} \\
C(n) &= \begin{cases} C_1(n) & \text{if } n \in N_1 - T_1' \\ C_2(n) & \text{if } n \in N_2 - \{s_2\} \\ C_1(n) \| C_2(s_2) & \text{if } n \in T_1' \end{cases}
\end{aligned}
$$

The TIG of a $\langle$*statement-list*$\rangle$ of more than two statements is constructed by repeated application of the above rules.

The next statement to consider is a simple conditional statement.

if $\langle$*expression*$\rangle$ then $\langle$*statement-list*$\rangle_1$ end if;

the TIG of the conditional statement.

Thus, the TIG for this statement is constructed from the TIG G_1 for $\langle \textit{statement-list} \rangle_1$ by adding the start node, s_1 to the set of terminal nodes and by redefining the pseudocode for the start node as follows.

$$
\begin{aligned}
N &= N_1 \\
E &= E_1 \\
S &= s_1 \\
T &= T_1 \cup \{s_1\} \\
L &= L_1 \\
C(n) &= \begin{cases} C_1(n) & \text{if } n \neq s_1 \\ \text{if } \langle \textit{expression} \rangle \text{ then } C_1(s_1) \text{ end if} & \text{if } n = s_1 \end{cases}
\end{aligned}
$$

A more general form of the if statement contains an else clause.

if $\langle \textit{expression} \rangle$ then $\langle \textit{statement-list} \rangle_1$ else $\langle \textit{statement-list} \rangle_2$ end if;

The TIG for this statement is constructed from G_1 and G_2, the TIG's for $\langle \textit{statement-list} \rangle_1$ and $\langle \textit{statement-list} \rangle_2$, by creating a new start node s, that combines the features of the start nodes s_1 and s_2. The nodes s_1 and s_2 are deleted from the graphs along with all edges starting at either s_1 or s_2. For each edge that started at either s_1 or s_2 there is a corresponding edge starting at s. The label on this edge is the same as the label on the original edge. The resulting graph is defined as follows:

$$
\begin{aligned}
N &= N_1 \cup N_2 \cup \{s\} - \{s_1, s_2\} \\
E &= (E_1 - \{(s_1, n)\}) \cup (E_2 - \{(s_2, n)\}) \cup \{(s, n) \mid (s_1, n) \in E_1 \text{ or } (s_2, n) \in E_2\} \\
S &= s \\
T &= \begin{cases} (T_1 - \{s_1\}) \cup (T_2 - \{s_2\}) \cup \{s\} & \text{if } s_1 \in T_1 \text{ or } s_2 \in T_2 \\ T_1 \cup T_2 & \text{otherwise} \end{cases} \\
L(n,m) &= \begin{cases} L_1(n,m) & \text{if } n \neq s_1 \text{ and } (n,m) \in E_1 \\ L_2(n,m) & \text{if } n \neq s_2 \text{ and } (n,m) \in E_2 \\ L_1(s_1,m) & \text{if } n = s \text{ and } (s_1,m) \in E_1 \\ L_2(s_2,m) & \text{if } n = s \text{ and } (s_2,m) \in E_2 \end{cases}
\end{aligned}
$$

$$C(n) = \begin{cases} C_1(n) & \text{if } n \in N_1 - \{s_1\} \\ C_2(n) & \text{if } n \in N_2 - \{s_2\} \\ \text{if } \langle \textit{expression} \rangle \text{ then } C_1(s_1) \text{ else } C_2(s_2) \text{ end if} & \text{if } n = s \end{cases}$$

Next we consider the TIG of a general loop statement

loop ⟨*statement-list*⟩ end loop;

where the ⟨*statement-list*⟩ may contain one or more ⟨*exit-statement*⟩'s. The formal specification for the new TIG follows.

$$\begin{aligned} N &= N_1 \\ E &= E_1 \cup \{(t,n) \mid (s_1,n) \in E_1 \text{ and } t \in T_1 - s_1\} \\ S &= s_1 \\ T &= \{n \mid n \in T_1 \text{ and } C_1(n) \text{ contains a terminate statement}\} \\ &\quad \cup \{n \mid n \in N_1 \text{ and } C_1(n) \text{ contains an } \langle \textit{exit-statement} \rangle\ \} \end{aligned}$$

$$L(n,m) = \begin{cases} L_1(n,m) & \text{if } (n,m) \in E_1 \\ L_1(s_1,m) & \text{if } n \in T_1 \text{ and } (s_1,m) \in E_1 \end{cases}$$

$$C(n) = \begin{cases} \text{loop } C_1(s_1) \ldots \text{end loop} & \text{if } n = s_1 \\ \text{loop } C_1(s_1) \ldots C_1(n) \text{ end loop} & \text{if } n \in T_1 \text{ and } n \neq s_1 \\ \text{loop } \ldots C_1(n) \ldots \text{end loop} & \text{if } n \notin T_1 \text{ and } n \neq s_1 \text{ and} \\ & \quad C_1(n) \text{ contains an } \langle \textit{exit-statement} \rangle \\ C_1(n) & \text{otherwise} \end{cases}$$

The loop ...end loop is used in the pseudocode to indicate both looping from the end of the loop back to the start of the loop and to provide a context for any exit statements. The ellipses (...) indicate code that is omitted because it is part of a different region.

As a further example of looping, the TIG of the statement

while ⟨*expression*⟩ loop $\langle \textit{statement-list} \rangle_1$ end loop;

is considered. In the TIG of this statement N, E, S, and L are the same as for the loop statement. T and C are defined as follows:

$$
\begin{aligned}
T &= T_1 \cup \{s_1\} \\
C(n) &= \begin{cases}
\text{while } \langle \textit{expression} \rangle \text{ loop } C_1(s_1) \ldots \text{end loop} & \text{if } n = s_1 \\
\text{while } \langle \textit{expression} \rangle \text{ loop } C_1(s_1) \ldots C_1(n) \text{ end loop} & \text{if } n \in T_1 \text{ and } n \neq s_1 \\
\text{while } \langle \textit{expression} \rangle \text{ loop } \ldots C_1(n) \ldots \text{end loop} & \text{if } n \notin T_1 \text{ and } n \neq s_1 \text{ and} \\
 & \quad C_1(n) \text{ contains an } \langle \textit{exit-statement} \rangle \\
C_1(n) & \text{otherwise}
\end{cases}
\end{aligned}
$$

Once again, the ellipses indicate code that is part of a different region.

Finally, the three general forms of the select statement are considered. The first form to be considered is:

select ⟨*accept-statement-list*⟩ {or ⟨*accept-statement-list*⟩} end select;

Consider a select statement with exactly k alternatives, and suppose $G_1, G_2, \ldots G_k$ are the TIG's of these ⟨*accept-statement-list*⟩'s. The TIG of this select statement is constructed by combining all the start nodes of G_1, G_2, ..., G_k into a single new start node s. The original start nodes s_1, s_2, ..., s_k are deleted from the graph. All edges leaving s are placed into a single edge group. The new TIG follows.

$$
\begin{aligned}
N &= N_1 \cup N_2 \cup \cdots \cup N_k \cup \{s\} - \{s_1, s_2, \ldots, s_k\} \\
E &= \bigcup_{i=1}^{k} (E_i - \{(s_i, n)\}) \cup \{(s, n) \mid (s_i, n) \in E_i \text{ where } 1 \leq i \leq k\} \\
S &= s \\
T &= T_1 \cup T_2 \cup \cdots \cup T_k \\
L(n,m) &= \begin{cases}
L_i(n,m) & \text{if } n \neq s_i,\ n \neq s, \text{ and } (n,m) \in E_i \\
L_i(s_i,m) & \text{if } n = s \text{ and } (s_i,m) \in E_i
\end{cases} \\
C(n) &= \begin{cases}
C_i(n) & \text{if } n \in N_i - \{s_i\} \\
\text{select } C_1(s_1) \text{ or } C_2(s_2) \text{ or } \ldots \text{ or } C_k(s_k) \text{ end select} & \text{if } n = s
\end{cases}
\end{aligned}
$$

The ellipses indicate the rest of the select alternatives. Note that for a TIG representing an ⟨*accept-statement-list*⟩ the start node cannot be a terminal node so it is not necessary to delete each s_i from T_i in the construction of T. The set of edges $\{(s,m)\}$ form an edge group.

The second form of a select statement to be considered is:

select ⟨*accept-statement-list*⟩ {or ⟨*accept-statement-list*⟩ } or terminate; end select;

Consider a select statement with exactly k alternatives, none of which is a terminate statement and a $(k+1)$-st alternative which is a terminate statement. The only differences between this example and the previous example are that s is added to T and pseudocode for s has the terminate alternative added. Thus, N, E, S, and L are the same as in the previous example and T and C are as follows.

$$T = T_1 \cup T_2 \cup \cdots \cup T_k \cup \{s\}$$

$$C(n) = \begin{cases} C_i(n) & \text{if } n \in N_i - \{s_i\} \\ \text{select } C_1(s_1) \text{ or } \ldots \text{ or } C_k(s_k) \text{ or terminate; end select} & \text{if } n = s \end{cases}$$

The ellipses indicate the rest of the select alternatives.

And finally we consider the third form of a select statement.

select ⟨*accept-statement-list*⟩ {or ⟨*accept-statement-list*⟩} else ⟨*statement-list*⟩ end select;

Consider a select statement with exactly k alternatives, none of which is a terminate statement and one else clause. Suppose that $G_1, \ldots, G_k$ are the TIG's of the select alternatives and that G_{k+1} is the TIG of the statement list in the else clause. This TIG is constructed by incorporating the else clause into the start node. The new TIG follows.

$$N = N_1 \cup N_2 \cup \cdots \cup N_{k+1} \cup \{s\} - \{s_1, s_2, \ldots, s_{k+1}\}$$

$$E = \bigcup_{i=1}^{k+1}(E_i - \{(s_i, n)\}) \cup \{(s, n) \mid (s_i, n) \in E_i \text{ where } 1 \le i \le k+1\}$$

$$S = s$$

$$T = \begin{cases} T_1 \cup T_2 \cup \cdots \cup T_{k+1} \cup \{s\} & \text{if } s_{k+1} \in T_{k+1} \\ T_1 \cup T_2 \cup \cdots \cup T_{k+1} & \text{otherwise} \end{cases}$$

$$L(n,m) = \begin{cases} L_i(n,m) & \text{if } n \neq s_i,\ n \neq s, \text{ and } (n,m) \in E_i \\ L_i(s_i,m) & \text{if } n = s \text{ and } (s_i,m) \in E_i \end{cases}$$

$$C(n) = \begin{cases} C_i(n) & \text{if } n \in N_i - \{s_i\} \\ \text{select } C_1(s_1) \text{ or } \ldots \text{ or } C_k(s_k) \text{ else } C_{k+1}(s_{k+1}) \text{ end select} & \text{if } n = s \end{cases}$$

The ellipses indicate the rest of the select alternatives.

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.

[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.