Detecting Deadlocks in the Ada `accept...do` and `select` Constructs

Stephen P. Masticola

Abstract

We describe the representation of Ada programs containing the `select` and `accept-do` constructs, for use in safe and accurate static detection of deadlock in polynomial time. We describe a *sync hypergraph* program representation, which encompasses remote procedures including synchronization, conditional entries and entry calls, and `else` clauses and timeouts. We present a corresponding execution model for the sync hypergraph abstraction of Ada programs, and give constraints on valid deadlock cycles based on this execution model.

We give full details of a deadlock detection algorithm, including lattice frameworks for deadlock cycle detection and proof of worst-case polynomial time bounds for convergence.

As an intermediate step, we compute an approximate "can't happen together" (CHT) relation between rendezvous statements. This CHT relation has applications in other areas, notably in detection of unexecutable statements and in intertask data flow analysis.

Contents

1 Introduction.

In [MR90], we described a program representation and a polynomial time algorithm for approximate detection of deadlock anomalies in a subset of Ada programs. This initial subset included statically declared tasks and message-passing rendezvous. Later, in [MR91b], we gave further methods to increase the size of the subset of programs that our algorithm could handle, by adding remote procedure calls, conditional entry calls, `select` statements, and *delay* alternatives to the set of Ada constructs we could represent. Space limitations did not permit a full description or complexity analysis of the algorithm there. The purpose of this document is to elaborate on some issues in representation of the program, and to describe in full detail the deadlock algorithm and its worst-case time complexity bounds.

Notation. Notation used in this paper is listed in Appendix A.

2 `Accept...do` construct.

Ada [ANSI83] has features that allow the rendezvous to act as a type of "remote procedure call". The task entry (or `accept`) acts as the body of the called procedure, and the signaling task acts as the caller. The syntax of a remotely callable procedure is as follows:

`accept <signal>[(<parameter list>)] [do <statement>];`

The code of the remote procedure is contained in the `<statement>` portion of the task entry shown. When a signaling task *s* sends a signal that coordinates with this task entry, *s* suspends execution until the body of the `<statement>` is completed. The `<statement>` can itself contain signaling and `accept` statements. Thus, task entries can be nested within each other. For obvious reasons, inner and outer nested entries cannot rendezvous with the same task.

Note that, when the `[do <statement>]` part is omitted, the `accept` construct behaves exactly as described in [MR90]. Thus, the behavior of the `accept...do` construct actually includes the behavior of the `accept` without its `do` part.

2.1 Sync hypergraphs.

We utilize a representation of the program called a *sync hypergraph* to detect deadlock in programs containing `accept-do` constructs. The graph incorporates *sync hyperedges*, which represent the interactions of a call point with a called procedure. A program fragment and its corresponding sync hypergraph are shown in Figure 1.

```
task T1 is
begin
  T2.sig;                          -- (s)
end T1;

task T2 is
begin
  accept sig do                    -- (a_i)
    <Body of procedure block x>    -- (x)
  end sig;                         -- (a_o)
end T2;
```

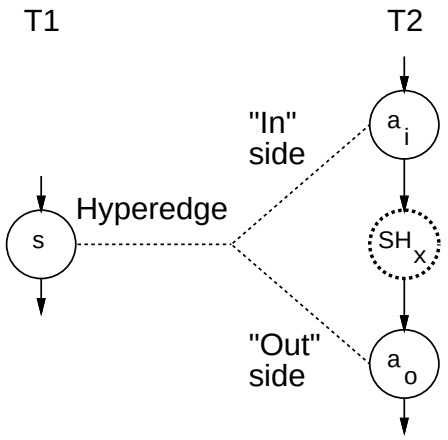

Figure 1: A program fragment and its sync hypergraph.

If the remote procedure contains rendezvous nodes, these must be represented in the sync hypergraph of the program. In the program fragment of Figure 1, the `accept` statement has an

attached procedure block x containing rendezvous statements. We represent a as a pair of nodes a_i and a_o corresponding to the entry ("in") and exit ("out") parts of the remote procedure respectively.

2.1.1 Representing control flow.

For the block x of Figure 1, we form a *sync subgraph* SH_x, including distinguished nodes b_x and e_x corresponding to x's beginning and end. To include SH_x in the complete sync graph, we simply replace b_x with a_i and e_x with a_o and draw sync edges appropriately, as described below. This gives us a way to recursively construct the nodes and control edges where embedded synchronization is present.

In the degenerate case where x has no synchronization, x may be represented by a single directed edge. (The original sync graph structure of [MR90] collapses x, a_i, and a_o into a single node a in this case.)

2.1.2 Representing synchronization.

Synchronization is represented by hyperedges, or edges with three endpoints. The general format of a hyperedge is (s, a_i, a_o). Such hyperedges are placed between every signaling statement s and the in and out nodes formed from each `accept` statement a of the same signal type. (A signal type y is a pair (t_y, n_y) where t_y is the receiving task and n_y is the entry name.)

For convenience, we call (s, a_i) the "in" side of the hyperedge and (s, a_o) the "out" side.

2.2 Execution model.

A signaling statement s can coordinate with an `accept` statement a containing a procedure block x. If x contains synchronization, then execution can advance through x without advancing past s. The previous execution model cannot represent this type of execution wave motion.

We augment the execution model of [MR90] with a new ENGAGED state. The model of rendezvous changes as follows:

- Initially, the execution wave contains one control successor of the beginning node x for each task. T is empty.
- When nodes s and a_i are both are on the execution wave, and are connected by the "in" side of a hyperedge, and s is not in the ENGAGED state, then both are in the READY state.

Any node on the execution wave which is not ENGAGED and is not connected to another node on the wave by a hyperedge is in state WAITING.

- Any pair of READY nodes connected by the "in" part of a hyperedge may be selected nondeterministically for firing.
- When s and a_i are selected for firing, a control successor of a_i is chosen nondeterministically and placed on the execution wave. The state of a_i changes to EXECUTED; the state of s changes to ENGAGED(a_o).
- If s is in state ENGAGED(a_o), and a_o has been chosen for execution, then both s and a_o are marked EXECUTED and their control successors are chosen for execution.

Nodes on the execution wave can be in READY or WAITING states as before, and signaling nodes on the execution wave can also be in the ENGAGED state. A signaling node cannot exit the ENGAGED state (or the execution wave) until the out node of the accept which engaged it is also on the wave, i.e., until a control path from a_i to a_o has been executed.

2.3 Infinite wait anomalies in the `Accept...do` model.

An execution wave which contains no READY nodes, but which contains some node other than the distinguished program end node e, has an infinite wait anomaly. All nodes on the wave are thus WAITING or ENGAGED, or are the distinguished node e.

Lemma 1 *An anomalous execution wave must include at least one node in the WAITING state.*

Proof: Suppose that some execution wave contained only copies of node e and nodes in state ENGAGED. If all nodes are e, then the program has terminated successfully, so at least one must be ENGAGED. Let s be the signaling node on the execution wave which was the last one to become ENGAGED. Let a_i be the accept node s coordinated with when s became ENGAGED. By construction, any control path from a_i to e must include a_o. By the transition rules above, a_o cannot leave the execution wave until s does. Thus, either a_o is on the wave or some node n on a control path from a_i to a_o is. By the transition rules, accept nodes cannot enter the ENGAGED state, so a_o is not on the execution wave. For the same reason, n must be a signaling node in the ENGAGED state. By construction, n cannot be a control predecessor of a_i, so n must have entered the execution wave after s and a_i coordinated. Before n entered the wave, its state was NOT-SEEN. Therefore, n entered the ENGAGED state after s, and s cannot have been the last node to become ENGAGED. □

2.3.1 Characteristics of anomalies.

A *deadlock anomaly* is characterized by a group of tasks on an anomalous execution wave, such that each is waiting for some other task in the group to proceed. A *starvation anomaly* is indicated by a node $n \neq e$ on the wave, such that n cannot coordinate with any other rendezvous node that can be reached in control flow from nodes currently on the wave. Tasks not directly deadlocked or starved may have to wait for deadlocked or starved tasks to complete execution, and thus be unable to proceed.

Naturally, the question arises: Given the new execution model, are deadlocks and starvation still the only possible types of infinite wait anomalies? The answer is yes; the proof is identical to that of Theorem 1 of [MR90].

2.3.2 Deadlocks involving hyperedges.

A deadlock is characterized by a group of tasks that mutually wait on each other. Consider a subset D of the head nodes of deadlocked tasks in an execution wave. Every member of D must be a signaling node s or an accept-in node a_i. (If any accept-out node a_o appears on the execution wave, it can rendezvous with some signaling node s which is ENGAGED to a_o's corresponding accept-in node.)

In the deadlock cycle D, there are only three ways in which one task $T1$ can wait on another task $T2$. These *wait modes* are shown in Figure 2.

In the first mode (Figure 2(a)), node s of task $T1$ is WAITING, and can rendezvous with a_i and a_o of task $T2$. Node a_i is NOT-SEEN, since it cannot execute until t (and possibly other intervening nodes in $T2$) have executed. Here, a_i is a tail node and s is a head node.

The second mode (Figure 2(b)) is really the reverse of mode (a). Node a_i in task T1 is WAITING, and can rendezvous with s in T2, but s is NOT-SEEN. Modes (a) and (b) correspond to modes which would occur if a_i and a_o had no intervening rendezvous nodes, i.e., the modes of the simplified model in which `accept` statements have no associated `do` blocks. Here, a_i is a head node and s is a tail node.

The third mode (Figure 2(c)) can occur only when `accept-do` statements are permitted. Here, s is ENGAGED to a_i, and therefore cannot complete until its corresponding accept-out node a_o executes. However, a_o is prevented from completing because some intervening node t in task T2 is WAITING on or ENGAGED to some NOT-SEEN node. Note that mode (c) includes cases where

Wait Mode	m	n	Resulting Edge
(a)	s	a_o	(a_o, s)
(b)	a_i	s	(s, a_i)
(c)	s	a_o	(a_o, s)

Table 1: Modes of the *Waits* relation.

s is embedded in an `accept-do`, or where there is an `accept-do` surrounding a, or one inside a and surrounding t. Here, a_o is a tail node and s is a head node.

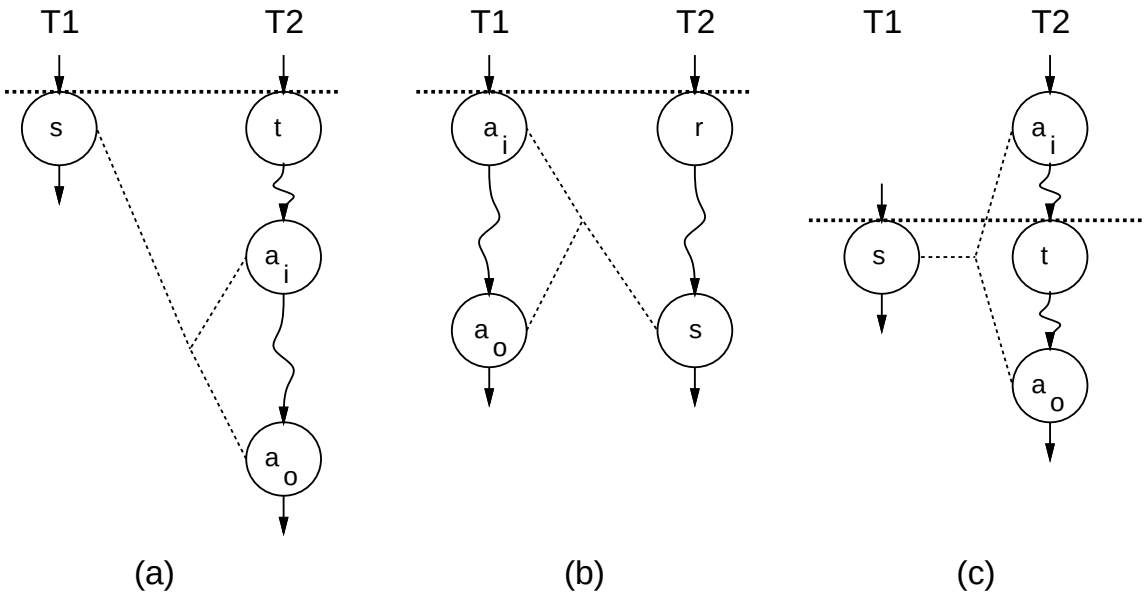

Figure 2: Deadlocks involving hyperedges.

2.3.3 "Waits on" relationship and hyperedges.

We define a relation $Waits(E, m, n)$ which means, "m cannot complete execution in execution wave E, but m could complete execution if n were on E, and n is a control successor of some node in E other than m." Informally, m waits on n in E. This relationship allows us to determine which transitions along sync hyperedges are valid in a deadlock cycle.

Table 2.3.3 summarizes, for each wait mode, which node waits on which other node.

In tracing deadlock cycles, we trace forward on control edges and from tail to head nodes (n to m) wherever m waits on n. Thus, the edges in the "Resulting Edge" column represent the

permissible transitions over hyperedge (s, a_i, a_o) when tracing a deadlock cycle.

2.4 Hyperedge cycle location graphs.

In cycle detection, we use the concept of a *hyperedge cycle location graph*, or HCLG, corresponding to the sync hypergraph. The HCLG can be constructed so that any cycles must satisfy constraint 1 of [MR90], i.e., that any cycle that enters a task should traverse at least one control edge in that task. Such a construction eliminates cycles which are spurious under constraint 1.

Figure 3 shows the transformation for signaling nodes and `accept-do` structures. Node s is split into engage and disengage nodes, s_e and s_d respectively. Edges between tasks correspond to sync hyperedges. Note that any path entering and then exiting a task in the HCLG must traverse at least one edge in that task, such that the edge corresponding to a control edge in the sync hypergraph. Subgraph x corresponds to an embedded `do` block, and is recursively expanded into subgraph x' in the HCLG.

Deadlock task interactions in modes (a) and (b) correspond to edges (a_o, s_d) and (s_e, a_i) respectively. Interactions in mode (c) have some node in x' as a head node, and traverse (a_o, s_d).

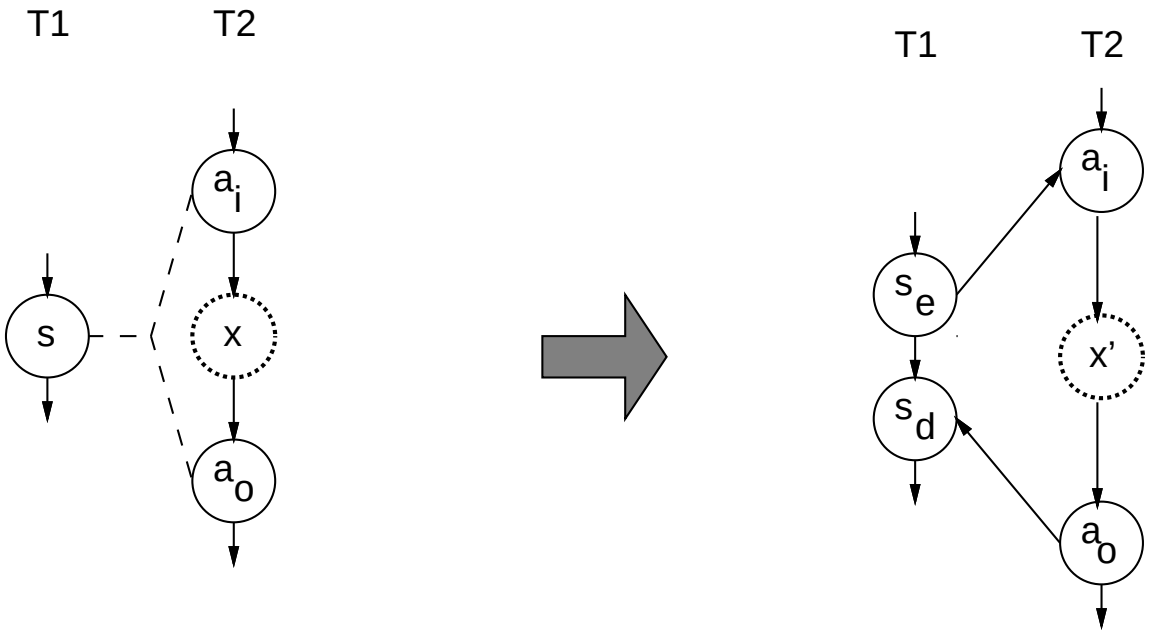

Figure 3: Hyperedge cycle location graph (HCLG) transform.

2.5 Scope depth and feasible deadlocks.

In [MR90], constraint 2 states that no two head nodes of a deadlock cycle may rendezvous with each other. Constraint 4 says that there must be some way to execute the nodes of the deadlock so that no node outside the deadlock can break the deadlock. The structure of the `accept-do` construct weakens both of these constraints. We outline a reinterpretation of constraints 2 and 4 in the presence of `accept-do` statements containing rendezvous nodes.

Constraint 2 must change because of mode (c) coupling. In Figure 2, node s may have the same signal type as node t. No rendezvous will occur, because s is already ENGAGED to a_o.

Similarly, 4 must change when `accept-do` is used. There may be no way in which to execute a deadlock such that some pair of nodes which can break the deadlock may be of the right signal type to rendezvous — but the sending node may always be in state ENGAGED, and thus may be unable to break the deadlock.

Let the *scope depth* (denoted $Scope(n, y)$) of a node n be the depth to which it is embedded in `accept-do` constructs of signal type y. $Scope(n, y)$ can easily be computed during the parse of the program. For any feasible execution wave E, the number of ENGAGED signaling nodes of type y equals the sum of the scope depths $Scope(n, y)$ of all nodes n on the wave. Call this summation $Scope_{set}(E, y)$; call the number of signaling nodes of type y in E, $SigN_{set}(E, y)$. Let predicate $Accept?(E, y)$ where $y = (t_y, m_y)$ be true if $E[t_y]$ is an accept node of signal name m_y.

In any execution wave E, for all signal types y,

$$Scope_{set}(E, y) \leq SigN_{set}(E, y) \tag{1}$$

since all signaling nodes required to increase scope depth must still be ENGAGED and therefore present on the execution wave. Also, if E is anomalous, then

$$Accept?(E, y) \Rightarrow Scope_{set}(E, y) = SigN_{set}(E, y) \tag{2}$$

since otherwise some accept-in node of type y in E could rendezvous with a non-ENGAGED signaling node. Since t_y is the only task that can contribute to $Accept?(E, y)$, we know that $Scope_{set}(E, y) = Scope(E[t_y], y)$. Therefore,

$$Accept?(E, y) \Rightarrow Scope(E[t_y], y) = SigN_{set}(E, y). \tag{3}$$

For the special case where $Scope_{set}(E, y) = 0$ for all E and y (i.e., no embedded rendezvous are permitted), then we substitute in Equation (1) to obtain

$$SigN_{set}(E, y) \geq 0. \tag{4}$$

This is clearly true for all execution waves E. If E is anomalous, then we can substitute in Equation (2) to obtain

$$Accept?(E, y) \Rightarrow SigN_{set}(E, y) = 0. \tag{5}$$

This implies the original constraint 2 of [MR90], that no pair of head nodes in a deadlocked execution wave (and hence a deadlock cycle) can rendezvous.

We would like to derive from Equation (1) and Equation (3) those conditions which can be checked given *only* knowledge of the deadlock cycle and the total number of active tasks. Let D be the set of head nodes of a deadlocked execution wave E. Clearly,

$$Scope_{set}(D, y) \leq SigN_{set}(D, y) + |E| - |D| \tag{6}$$

since otherwise the total number of signaling nodes of type y which could possibly be on E would not be sufficient to drive D to the given scope depth. A more restrictive condition is

$$\sum_{y \in Sigs} [Scope_{set}(D, y) - SigN_{set}(D, y)] \leq (|E| - |D|). \tag{7}$$

since there must be sufficient signaling nodes on E to drive all of the nodes in D to their scope depth. This is a condition on the feasibility of D.

For the condition on accept nodes of Equation (3), we have that

$$Accept?(D, y) \Rightarrow Scope(D[t_y], y) = SigN_{set}(E, y). \tag{8}$$

Therefore,

$$Accept?(D, y) \Rightarrow Scope(D[t_y], y) - SigN_{set}(D, y) = SigN_{set}(E - D, y) \tag{9}$$

where $0 \leq SigN_{set}(E - D, y) \leq |E| - |D|$. Therefore, it must also be true that

$$Accept?(D, y) \Rightarrow 0 \leq Scope(D[t_y], y) - SigN_{set}(D, y). \tag{10}$$

If Equation (10) is false for some D, then two nodes of D can rendezvous and D is thus not a deadlock.

2.6 Impact to constraints.

We examine the constraints of [MR90] one by one to determine whether they hold, and how they are changed, by the new model of synchronization.[1]

[1] The original constraints are in italics; changes are in plain typeface.

1. *Each node in D is transitively coupled to all others in D.*

   We have shown how each of the wait modes is modeled by hyperedges in the sync hypergraph.

   Thus, there is a cycle in the sync graph, which includes the nodes of D, and such that, for every node $r \in D$ in task u:

   (a) *The path enters node r through a sync edge.*

   Paths in the sync hypergraph can only go between tasks by traversing sync hyperedges. In a valid deadlock cycle, the hyperedges are taken from a signaling node to an accept-in node, or from an accept-out node to a signaling node.

   (b) *The path traverses at least one control flow edge in task u.*

   When a path enters a task in the HCLG, it cannot exit the task without traversing at least one edge corresponding to a control edge in the sync hypergraph.

   (c) *The path leaves u via a sync edge and does not re-enter u.*

   Section 3.1.1 of [MR90] shows that, in straight-line code without conditional branching, any cycle which enters a task more than once corresponds to one which enters the task only once. In code with conditional branching (but no loops), any path that enters a task T more than once either corresponds to some other path which enters T only once, or includes a pair of nodes which are not co-executable.

   The above observation is based on reachability in control flow; if head node n can be reached from head node m, then any path which enters the task at both m and n also corresponds to one which enters only at m. If m and n are mutually unreachable in control flow, then they are not co-executable. This observation is unchanged in the sync hypergraph.

2. *No two head nodes of the deadlock cycle can rendezvous with each other.*

   This constraint is on the head nodes of a deadlock cycle alone. The nodes can rendezvous with each other only if the signaling node is not always ENGAGED. Equation (10) expresses a subset of this condition which is determinable from only the proposed deadlock and the number of tasks. We repeat Equation (10) here:

$$Accept?(D, y) \Rightarrow 0 \leq Scope(D[t_y], y) - SigN_{set}(D, y).$$

3. *All head nodes may execute at the same time.*

(a) *The nodes in D must represent statements whose instances can happen together.*

(b) *All nodes in D are* co-executable *in the sense of Callahan and Subhlok [CS88].*

In the presence of `accept-do` statements, we have new information on co-executability of the nodes in D. It must be possible to have enough ENGAGED nodes of each signal type on the execution wave to drive the nodes in D to their scope depth with respect to all signal types. This implies Equation (7), which we repeat here:

$$\sum_{y \in Sigs} [Scope_{set}(D, y) - SigN_{set}(D, y)] \leq (|E| - |D|)$$

.

4. *The nodes of D must be able to execute concurrently with a set of remaining nodes $E - D$ such that none of the nodes in $E - D$ can rendezvous with each other or with any member of D.*

   Equation 7 is also relevant to this constraint, since it determines whether D can execute, given the size of $E - D$.

3 `Select` construct.

Ada has a *selective wait* mechanism that allows a task to select from a number of available `accept` statements. The selective wait mechanism is available through use of the `select` construct, whose syntax is:

```
select     [when <guard>] <accept-alt> [<sequence of statements>]
       [or [when <guard>] <accept-alt> [<sequence of statements>]]...
       [else <sequence of statements>]
end select;
```

where `<accept-alt>` is of one of the following forms:

```
accept    <signal> [do <sequence of statements>] -- or --
delay     <expression>                           -- or --
terminate
```

The selective wait may contain an `else` clause, a set of `delay` alternatives, or a `terminate` alternative, but no pair of these constructs are permitted in the same selective wait statement.

The guard is a Boolean expression. Any of the sister `accept` clauses without a guard, or whose guard is *open* (true) may rendezvous when execution reaches the selective wait and some other task has made the proper entry call. If all guards are *closed* (false), and there is no `else` clause, then a runtime exception is raised.

We refer to the `accept` statements of a conditional entry construct as *sisters* of each other. The optional `else` statement, if present, would execute if none of the sister accepts was available for a rendezvous. Thus, the `accept` clauses of a `select...else` construct cannot participate as the head nodes of a deadlock. (A rendezvous inside the `else` clause may be a head node in a deadlock cycle, but only if none of the open `accept` statements can rendezvous.)

Similarly, an open or unguarded `delay` alternative, if present, would execute if none of the open sisters could rendezvous before the specified delay period had expired. Thus, no set of sisters with an unguarded `delay` alternative can be the head nodes of a deadlock. However, the sequence of statements following a `delay` statement may contain rendezvous, and may ordinarily participate in a deadlock as either head or tail nodes.

3.1 Representing conditional entries.

Figure 4 shows the translation of a selective wait statement into a sync hypergraph. The `accept` clauses `u1` through `um` are all unguarded, while `accept` clauses `g1` through `gn` have guard clauses.

A pair of nodes is constructed for each entry. The nodes of each `accept` clause are marked as not co-executable with those of its sisters and with the nodes of the `else` clause and `delay` alternatives. Other markings, which are described below, denote limits on the number of signaling nodes of various types in any valid deadlock cycle containing nodes in the `select` statement.

```
-- Control predecessors
select
  -- Unguarded entries
  accept u1
    do <seq_u1>
  end u1;
  <seq_v1>
or
  ...
or
  accept um
    do <seq_um>
  end um;
  <seq_vm>
or
  -- Guarded entries
  when <guard_1>
    accept g1
      do <seq_g1>
    end g1;
    <seq_h1>
or
  ...
or
  when <guard_n>
    accept gn
      do <seq_gn>
    end gn;
    <seq_hn>
or
  when <guard_d1>
    delay <time>;
    <seq_delay1>
or
  ...
or
  when <guard_dk>
    delay <time>;
    <seq_delayk>
else
  <seq_else>
end select;
-- Control successors
```

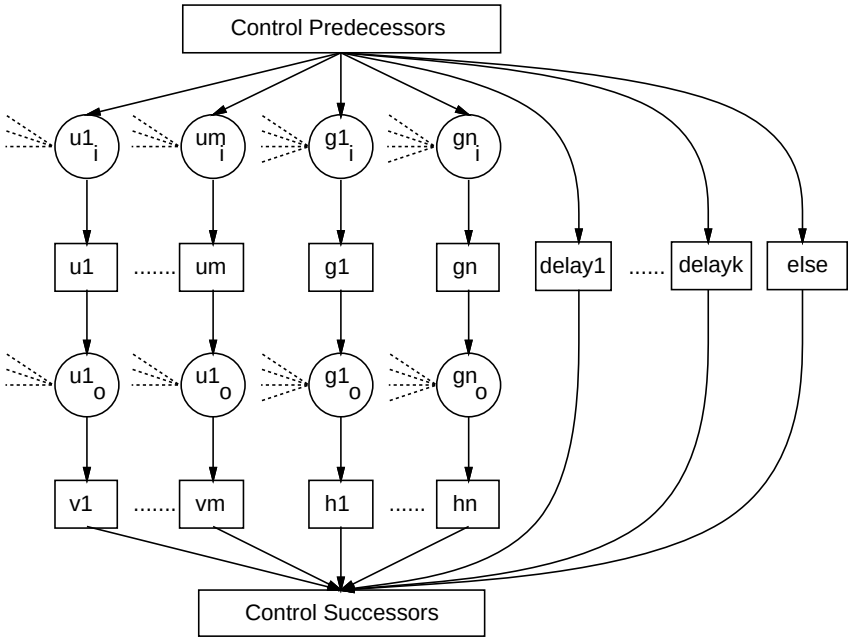

Figure 4: Representing conditional entries in the sync hypergraph.

Sections 3.2 and 3.3 explain the reasons for the construction of the sync hypergraphs for `select` statements, and detail some assumptions made during deadlock analysis.

3.2 Conditional entries without `else` or `delay`.

The three modes of interaction previously outlined for `accept-do` statements also apply to `select` statements. In mode (a), the `select` statement has not been reached in execution flow, but some node s on the wave might be able to complete if it were. In mode (b), the `select` statement has been reached in execution, but none of its open `accept` clauses can rendezvous. In mode (c), one of the open `accept` clauses $a_i ... a_o$ has been engaged, but a node t contained in that clause cannot proceed.

3.2.1 Guard clauses.

Guard clauses for the `accept` statements are optional. The unguarded `accept` statements are always open.

In special cases, it may be possible to perform some kind of case analysis on the guard clauses, to determine which may be open and which may be closed at any given time in program execution. In general, performing an exact analysis of this sort is an intractable problem, so we must make the most conservative assumptions possible about which guards are open.

The most conservative assumption here is the one which gives the greatest possibility of deadlock, i.e., that the minimum possible number of sisters of each hypothesized head node are available for rendezvous. We assume that all unguarded sisters of a hypothesized head node are always open, but that the only open guarded sister is the hypothesized one (if the hypothesized head node is guarded).

3.2.2 Commitment and sync hyperedge paths.

We examine possible sync hyperedge paths according to task interaction mode, to determine what assumptions about open guards are precise and conservative. In this analysis, node a_i represents one of several `accept` alternatives and a_o represents the end of the `do` block of the *one* alternative selected for execution (if any are). Each of the alternative `accept` statements may be either guarded or unguarded.

In mode (a) task interactions, a deadlock path may go from any of the alternatives of a_i to s. To be conservative, we must assure that the maximum number of possible cycles will be followed. We therefore assume that any of the `select` alternatives of a_i would be open, if the `select` could be reached in execution. That is, any of the `select` alternatives may be used as a tail node.

In mode (b) task interactions, if any of the `accept` alternatives is open and READY to coordinate, there cannot be a deadlock. Constraint 2 leads us to discard any proposed deadlock cycle in which some signaling node can rendezvous with any of the unguarded alternatives. For conservativeness, we should not discard proposed deadlock cycles in which a signaling node can rendezvous with a guarded alternative, since the alternative may be closed by the guard. (See Section 3.2.1.)

In mode (c) task interactions, we have marked a signaling node ENGAGED and partially executed the `do` clause of an `accept` statement. In the `select` construct, we need to make sure that the `do` clause executed corresponds to the `accept` statement which was actually engaged. We must therefore construct the sync hypergraph so that this will always be the case.

To represent the possible control edge paths in deadlock cycles, we insert control edges from each control predecessor of the `select` statement to the accept-in node of each `accept` alternative. Likewise, we insert control edges from the last nodes of each `accept` alternative to each control successor of the `select` statement. (See Figure 4.)

3.2.3 Scope depth and conditional entries.

If any of the "in" nodes of any of the sister entries of a selective wait statement are on the execution wave, then the "in" nodes of *all* sisters are effectively on the wave. Thus, no valid deadlock cycle can contain both the "in" node of a `select` clause and a signaling node that can rendezvous with one of its unguarded sisters.

Say that we have hypothesized a deadlock cycle D including accept-in node a_i, which has an unguarded sister node u_i of type y. If $Scope(u_i, y) < SigN_{set}(D, y)$, then u_i can rendezvous and D is not a true deadlock. Note that $Scope(u_i, y) = Scope(a_i, y)$, since both nodes are nested inside the same set of `accept-do` constructs.

Let $UngSis?(D, y)$ be a a predicate which is true iff D contains an accept-in node with type y, or a sister node of any unguarded accept-in node of type y. Remembering that task t_y contains all accept nodes of type y and all sisters of such nodes, we can rephrase Equation (10) as:

$$\forall y \overset{\text{such that}}{} UngSis?(D, y) : 0 \leq Scope(D[t_y], y) - SigN_{set}(D, y) \tag{11}$$

3.3 Conditional entries with `else` clauses or `delay` alternatives.

The `else` clause will typically not include any coordinations with other tasks. However, [ANSI83] does not rule out this possibility, so it must be considered for the sake of completeness. We first

examine the impact of `else` clauses without rendezvous on possible deadlocks, and then look at the effects of including rendezvous statements in the `else` clause.

3.3.1 `Else` clauses and `delay` alternatives without rendezvous statements inside the clause.

Consider a conditional entry's accept node a, as shown in Figure 2. In mode (a) interactions, s can rendezvous with one of the `accept` clauses of the unexecuted conditional entry. In mode (c), one of the `accept` clauses has already been committed to execution, so s would still wait on a_o.

When a conditional entry with an `else` clause is present on the execution wave, mode (b) task interactions do not lead to deadlock. In such a case, execution will fall through to the `else` part. Thus, the hyperedge need never be traversed from s to a_i.

To represent `select` statements with an `else` clause or `delay` alternative with no rendezvous, we insert a control edge from each predecessor of the `select` statement to each successor. We also note that the accept-in node of each `accept` alternative cannot be a head node. (The sync hypergraph of Figure 4 has this construction if the `else` clause body contains no rendezvous.)

3.3.2 `Else` clauses and `delay` alternatives containing head nodes.

A *frontier node* in the sync hypergraph of a clause is defined as a node which is reached in control flow only from the beginning of the clause. A frontier node of an `else` clause or `delay` alternative may be the head node of a deadlock cycle only when none of the open `accept` clauses is ready to rendezvous when the `select` statement is executed (or before the delay period has expired). This may happen only when none of the open `accept` clauses can rendezvous with any of the signaling nodes on the execution wave at the time the `select` clause was executed, or within the delay period following the start of execution.

We might be tempted to conclude from this that the frontier nodes of an `else` clause or `delay` alternative cannot be in a deadlock cycle along with any node that could rendezvous with an unguarded `accept` alternative. However, this is not true. It is possible, though not intuitive, that a frontier node may be co-executable with a signaling node that can rendezvous with one of the unguarded sister `accept` statements. For an example, see Figure 5. Suppose that, at the time the `accept` statement in task T1 started execution, node n in task T2 was waiting to rendezvous. At that time, no rendezvous to any of the open sisters was available, so the `else` statement started execution, and control flow reached f_1. After that, in T2, n completed its rendezvous and s started

execution. Even though s would have prevented f_1 from executing if it had been executing before the `accept` statement started, s can execute simultaneously with f_1.

It is possible that there may be a deadlock with a head node in an `else` or `delay` clause and a tail node that is not in the same clause. Therefore, we insert control edges from the last nodes in the `else` or `delay` clause to the control successors of the `select` statement. (See Figure 4.)

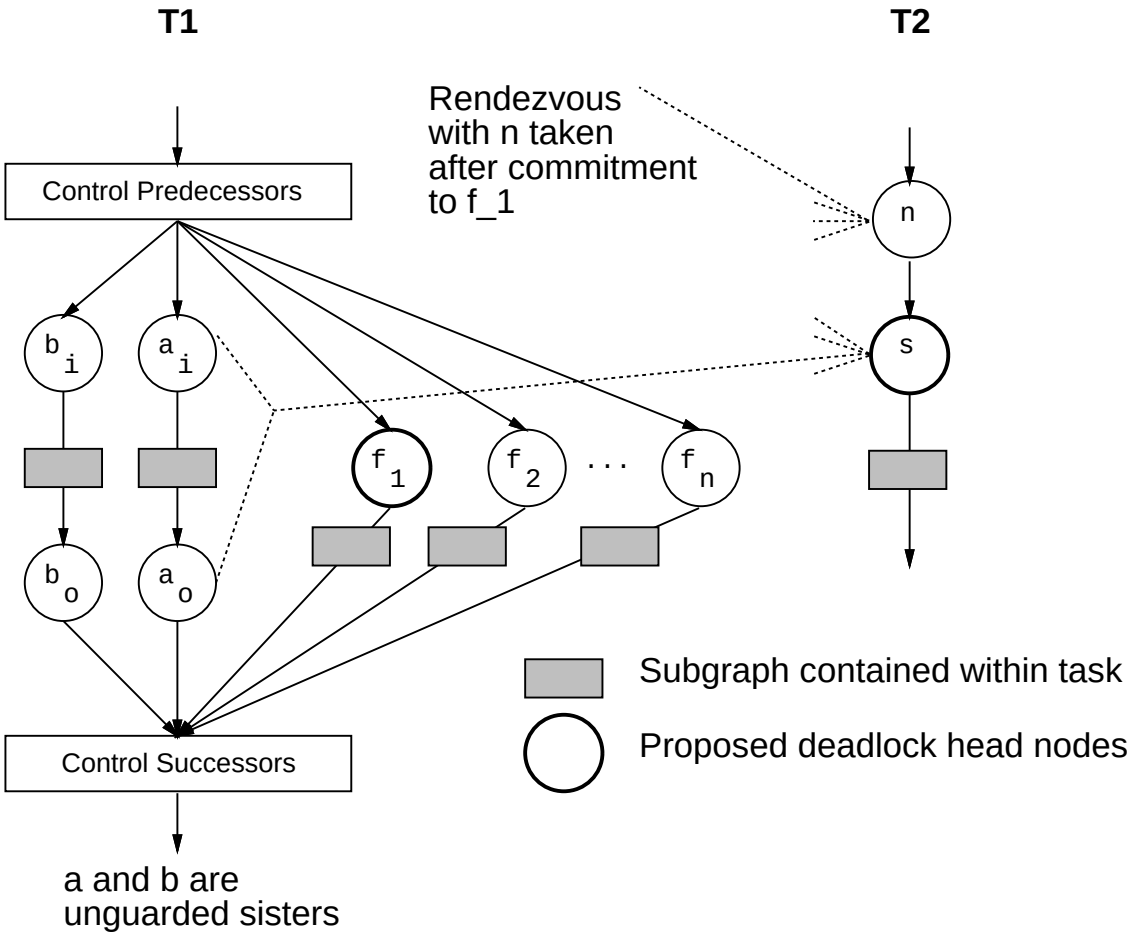

Figure 5: Co-executability and `else` clauses of selective wait statements.

The `else` clause or `delay` alternative of a conditional entry is not co-executable with any of the conditional entries themselves. We mark the possible head nodes of the `else` clause or `delay` alternative in this way.

3.3.3 `Else` clauses and `delay` alternatives containing tail nodes.

A tail node represents a rendezvous node which could break a deadlock, if that tail node were present on the execution wave. From this viewpoint, and also from the conservative viewpoint of

forming the greatest number of deadlock cycles, any rendezvous nodes inside `else` clauses or `delay` alternatives should be permitted to act as tail nodes wherever it is feasible that they can do so, even if the head node is not contained within the `else` clause. Thus, we should construct the sync hypergraph so that there are control paths from the control predecessors of the conditional entry to the nodes of the `else` clause or `delay` alternative. (See Figure 4.)

3.4 Conditional and timed entry calls.

A *conditional entry call* mechanism also exists in Ada. It allows an entry call to be executed, depending on whether an `accept` is available. Its syntax is:

```
select <entry call> [<sequence of statements>]
       else <sequence of statements>
end select;
```

The *timed entry call* is a similar construct of the form:

```
select <entry call> [<sequence of statements>]
       or delay <expression> <sequence of statements>
end select;
```

3.4.1 Representing conditional entry calls.

Figure 6 shows how a conditional or timed entry call is represented in the sync hypergraph. The `else` part is treated as a conditional statement which may be executed only if the entry call cannot be accepted. Thus, the entry call and subsequent nodes are marked as not co-executable with the nodes of the `else` clause. The conditional entry call is marked as such, so that it will not be included as a head node in deadlock cycles.

3.4.2 Conditional entry calls and deadlock.

If the entry call cannot rendezvous, then execution will proceed to the `else` part of the conditional call, or the delayed sequence of statements in the timed entry call. The entry call thus cannot possibly be a head node in a deadlock cycle. However, rendezvous nodes embedded in the `else` or `delay` part might be head nodes, if they do not themselves have `else` clauses. The same is true for rendezvous embedded within the sequence of statements following the call. Therefore, the construction of conditional entry calls is similar to that of conditional entries.

The nodes of the `else` clause are not co-executable with the conditional entry call or any nodes

```
-- Control predecessors
select
  t.s;              -- Node s
  <seq_cond_call>;
else
  <seq_else>;
end select;
-- Control successors

       -- or --

-- Control predecessors
select
  t.s;              -- Node s
  <seq_cond_call>;
or delay <expr>
  <seq_else>;
end select;
-- Control successors
```

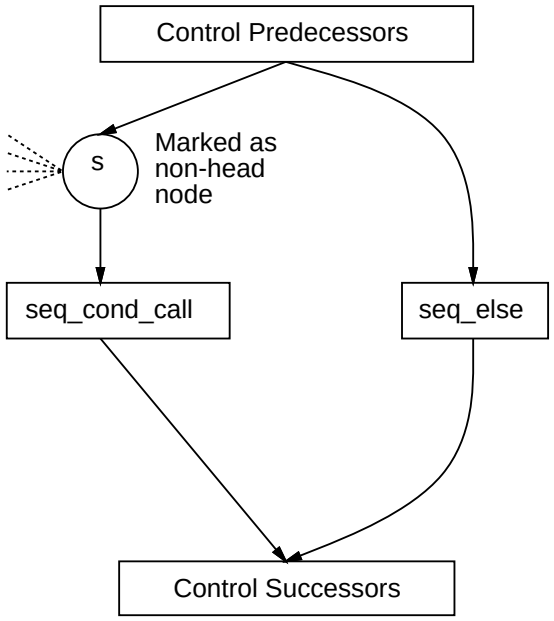

Figure 6: Representing conditional entry calls in the sync hypergraph.

in the sequence of statements that follow it. However, for reasons similar to those of 3.3.2, a frontier node of an `else` clause in a conditional entry call may be co-executable with an `accept` statement which can rendezvous with the conditional call. Therefore, the construction of the sync hypergraph for conditional entry calls is similar to that for conditional entries.

3.5 Impact to constraints.

If a proposed deadlock cycle D is a valid deadlock cycle, then:

1. *Each node in D is transitively coupled to all others in D. Thus, there is a cycle in the sync graph, which includes the nodes of D, and such that, for every node $r \in D$ in task u:*

   (a) *The path enters node r through a sync hyperedge, going along a sync hyperedge as follows:*

   i. *From an accept-out node to a signaling node.*

   Since conditional entry calls are never deadlock head nodes, the signaling node in this case may not be a conditional entry call.

   ii. *From a signaling node to an accept-in node.*

   The accept-in node may not have a `else` clause. If it does, it cannot be a head node in a deadlock cycle.

(b) *The path traverses at least one control flow edge in task u.*

(c) *The path leaves u via a sync hyperedge (as above) and does not re-enter u.*

2. *No two head nodes of the deadlock cycle can rendezvous with each other. This implies Equation (10):*

$$Accept?(D, y) \Rightarrow 0 \leq Scope(D[t_y], y) - SigN_{set}(D, y).$$

Equation (11) generalizes Equation (10) as follows:

$$\forall y \overset{\text{such that}}{} UngSis?(D, y) : 0 \leq Scope(D[t_y], y) - SigN_{set}(D, y)$$

3. *All head nodes may execute at the same time.*

(a) *The nodes in D must represent statements whose instances can happen together.*

(b) *All nodes in D are* co-executable *in the sense of Callahan and Subhlok [CS88]. This implies Equation (7):*

$$\sum_{y \in Sigs} [Scope_{set}(D, y) - SigN_{set}(D, y)] \leq (|E| - |D|).$$

4. *The nodes of D must be able to execute concurrently with some set of remaining nodes $E - D$ such that none of the nodes in $E - D$ can rendezvous with each other or with any member of D.*

If any node a_i of D is a conditional entry, then all conditional entries entries in its `select` statement, which are unguarded or have open guards, are available for rendezvous. If one of these is an unguarded entry $u_i = (t, y)$, and $SigN_{set}(D, y) > Scope(u_i, y)$, then D is always broken by a rendezvous with u_i, even though u_i does not necessarily appear in D.

4 Searching for valid deadlock cycles.

This section describes the method used to safely approximate a search for valid deadlock cycles in the sync hypergraph. Scope depth is used to eliminate invalid deadlock cycles, but the algorithm is to be kept polynomial in the size of the sync hypergraph.

We consider two subproblems separately. Either a proposed set of deadlock head nodes D contains at least one accept-in node or it contains only signaling nodes. In the first case, we can conduct our searches starting only at accept nodes, and apply the constraint of Equation (11) to eliminate spurious cycles. In the second case, we can detect deadlocks by searching for cycles that include only signaling head nodes, and eliminate the propagation needed for mode (b) task interactions.

4.1 Steps in deadlock detection.

Figure 7 shows the steps in the process of deadlock detection. As input, the deadlock detection algorithm is given an Ada source program. The program is parsed, and the *parse form sync hypergraph* is generated. From the parse form sync hypergraph, two other sync hypergraphs are generated. One of these sync hypergraphs, the *CHT form*, is in a form used by *CHT* analyses; the other hypergraph, called the *reach form*, is used by the reachability (deadlock detection) analysis. Structural details of these graphs are given in Section 4.1.1.

The parser also generates the following ancillary information, not shown on the diagram:

- Sets of sisters and unguarded sisters of accept-in nodes.
- A set of nodes which can't be head nodes, due to the presence of an `else` clause or an unguarded `delay` alternative.
- The scope depth of each accept-in node, with respect to its own signal type.
- A map from loop nodes in the parse form sync hypergraph to the hypergraphs of the loop bodies they represent.
- A table of all signal types, and the nodes which reference each signal type.

From the *CHT* form sync hypergraph, we obtain *CHT* information using the following analysis techniques:

- *Limited B4 analysis*, which finds pairs of nodes such that all instances of one node must complete before any instance of the other can start.
- *Pinning analysis*, which can determine (in some cases) which nodes must execute together.
- *Critical section analysis*, in which critical section structures are found in the sync hypergraph, and are used to infer *CHT* information.
- *Remote procedure analysis*, based on calls to remote procedures that include rendezvous statements in their bodies.
- *Iteration* of all the above techniques, to eliminate sync hyperedges between nodes that can't happen together. Iteration improves the precision of the solution.

Following *CHT* analysis, the parse form sync hypergraph is transformed into the reachability form sync hypergraph. During the translation process, the *CHT* information is also translated to correspond to the reachability sync hypergraph.

The next step is the iterative solution of the reachability problem for all possible head nodes. The result of this analysis is, for each node n in the program, the set $RHead(n)$, which is a conservative approximation of the potential head nodes which may include n on some deadlock cycle (as well as some other information about the paths from those head nodes to n). To find $RHead(n)$, we solve $SACnt_h(n)$ for each possible head node h. For convenience, we do not explicitly construct a hyperedge cycle location graph, but instead maintain separate "engage" and "disengage" sets for signaling nodes in the reachability form sync hypergraph.

The final step in deadlock detection is to inspect $RHead(n)$ for all n. If n is included in $RHead(n)$, then there may be a deadlock involving n as a head node. If no node is included in its own *RHead* set, then the program cannot deadlock.

4.1.1 Sync hypergraph formats.

The parse form sync hypergraph output by the parser is designed to be easily converted into either the reach form or *CHT* form hypergraphs. Figure 8 shows the parse form sync graph that represents the following program fragment:

```
[ 1] FOR i IN 1..x LOOP          -- LS1
[ 2]   t1.s1;                    -- S1
[ 3]   FOR j IN 1..y LOOP        -- LS2
```

```
[ 4]     t2.s2;                 -- S2
[ 5]     FOR k IN 1..z LOOP     -- LS3
[ 6]       t3.s3;               -- S3
[ 7]     END LOOP;              -- LE3
[ 8]   END LOOP;                -- LE2
[ 9]   t4.s4;                   -- S4
[10] END LOOP;                  -- LE1
```

The parse form sync graph actually consists of a set of sync hypergraphs, one for the body of each loop and one for the main program. Each loop is represented in the body of the containing loop or program by a *loop node*. A map is maintained from each node to the sync hypergraph of the loop body it represents. For instance, loop node `L1` in hypergraph `G0` represents the loop body of sync hypergraph `G1`. Similarly, loop node `L2` represents loop body `G2`. A control edge is placed from each predecessor of the loop node to each successor, since the bodies of the loops that the loop nodes represent might not necessarily be executed.

Before attempting to detect deadlock, we must determine which pairs of nodes in the sync hypergraphs represent rendezvous statements such that no instances of the statements can happen together. This problem is called the *can't happen together*, or *CHT*, problem. The *CHT* form sync hypergraph is used in *CHT* analysis. In the *CHT* form, all the control paths between rendezvous nodes are represented as control edges, and there is a one-to-one mapping between nodes and statements (except for the distinguished begin and end nodes b and e, which do not correspond to statements). For each task t, a *task end node* e_t is included, as well as the program end node e; task end nodes are used to distinguish the end of a single task from the termination of the program. Figure 9 shows a fragment of a program and the *CHT* form sync hypergraph that represents it.

After *CHT* analysis is completed, the parse form sync hypergraph is translated to one in reachability form, as in Figure 10. This is done by replacing all loop nodes with the loop bodies they represent. (Outermost loops are replaced by two serially concatenated copies of the loop bodies they represent; the nodes of the second copy are added to the set of non-head nodes.)

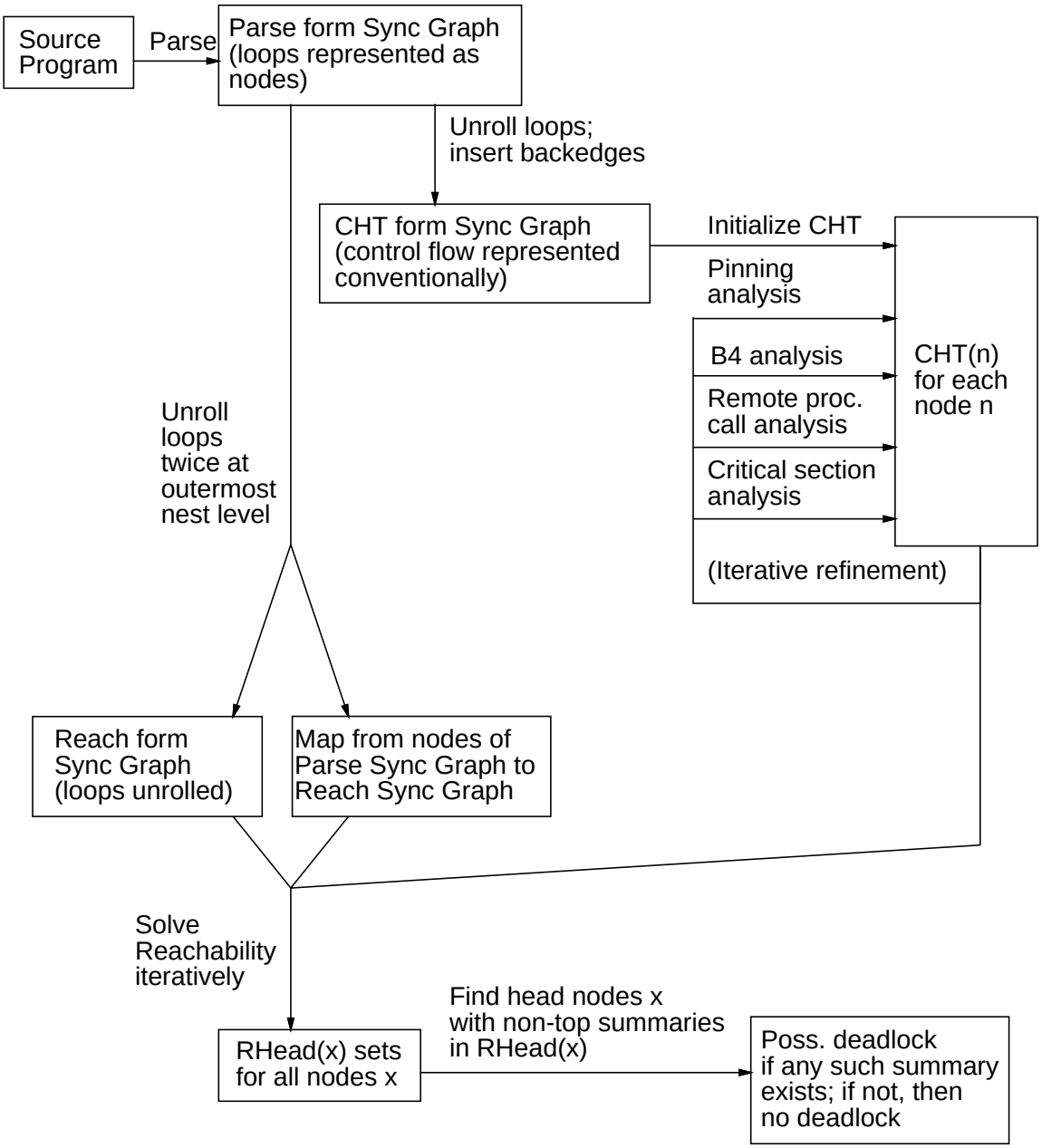

Figure 7: Process of static deadlock detection.

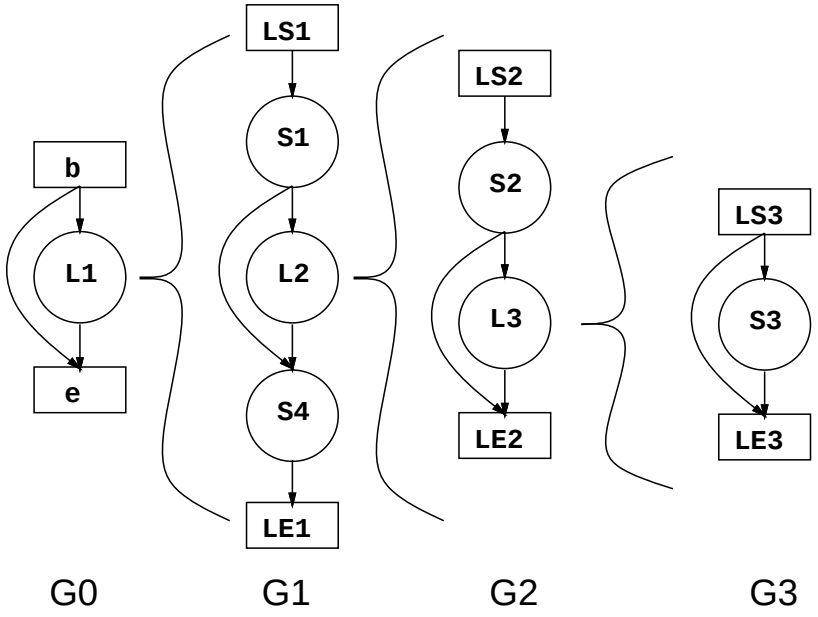

Figure 8: Representation of loops in the parse form sync hypergraph. Sync hyperedges are not shown.

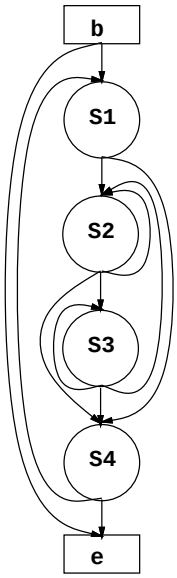

Figure 9: Representation of loops in *CHT* form sync hypergraph.

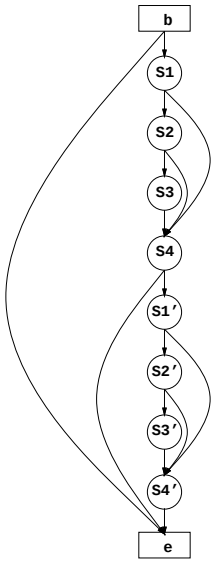

Figure 10: Representation of loops in reachability sync hypergraph.

5 Lattice frameworks and properties for deadlock detection.

A *lattice framework* [Hec77] is a quadruple $D = (G, L, F, M)$ where $G = (N, E)$ is a graph, L is a lattice, F is a set of monotone functions $L \rightarrow L$ such that F is closed under meet and composition, and M is a mapping function $E \rightarrow F$ from the edges of G to functions in F. Lattice frameworks are commonly used to represent iterative problems in data flow analysis, and are convenient for analyzing the time for convergence of iterative data flow algorithms. In this section, we describe some properties of lattice frameworks, which we will later use in the timing analysis of the the *B4* and *RHead* problems.

5.1 Motivating the analysis.

In the sections that follow, we analyze the 1-semiboundedness and distributivity properties of the function spaces for lattice frameworks. We are interested in these properties for the following reasons:

- *Applicability of complexity results.* The Kam-Ullman time complexity result is applicable only to lattice frameworks whose function spaces are both distributive and 1-semibounded. (See Section 5.4.)
- *Solution time of path methods.* The k-boundedness properties of the function spaces are important in bounding the execution time of Tarjan's path problem methods. (See Section 5.3.)
- *Accuracy.* If the function spaces are distributive, then the maximum fixed point of an iterative algorithm is the meet-over-all-paths solution of the problem, i.e., the iterative algorithm is perfectly accurate.

5.2 General lattice framework properties.

We present the following lemmas for later use in proving properties of our lattice frameworks.

5.2.1 Closure properties.

Lemma 2 *In a lattice, if* $a \sqsubseteq b$ *and* $c \sqsubseteq d$, *then* $a \sqcap c \sqsubseteq b \sqcap d$.

Proof:

$$\begin{aligned} a &\sqsubseteq b \\ a \sqcap c &\sqsubseteq b \sqcap c \\ c &\sqsubseteq d \\ b \sqcap c &\sqsubseteq b \sqcap d \\ a \sqcap c &\sqsubseteq b \sqcap d. \square \end{aligned}$$

Lemma 3 *The set of monotone functions on a lattice is closed under composition and meet.*

Proof: A monotone function f is such that

$$a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b).$$

If two functions f_1 and f_2 are monotone, then

$$a \sqsubseteq b \Rightarrow f_1(a) \sqsubseteq f_1(b)$$

and

$$a \sqsubseteq b \Rightarrow f_2(a) \sqsubseteq f_2(b).$$

By Lemma 2,

$$a \sqsubseteq b \Rightarrow f_1(a) \sqcap f_2(a) \sqsubseteq f_1(b) \sqcap f_2(b).$$

By the definition of monotonicity,

$$a \sqsubseteq b \Rightarrow f_1(f_2(a)) \sqsubseteq f_1(f_2(b)). \square$$

Lemma 4 *The set of distributive functions on a lattice is closed under function composition and meet.*

Proof: A function f is distributive if, for all $a, b \in L$, $f(a \sqcap b) = f(a) \sqcap f(b)$. Suppose f_1 and f_2 are distributive. Then

$$\begin{aligned} f_1(f_2(a \sqcap b)) &= f_1(f_2(a) \sqcap f_2(b)) \\ &= f_1(f_2(a)) \sqcap f_1(f_2(b)). \end{aligned}$$

Therefore, the composition of two distributive functions is also distributive, and the set of distributive functions is closed under composition.

Also, let $f = f_1 \sqcap f_2$.

$$\begin{aligned} f(a \sqcap b) &= f_1(a \sqcap b) \sqcap f_2(a \sqcap b) \\ &= f_1(a) \sqcap f_1(b) \sqcap f_2(a) \sqcap f_2(b) \\ &= f(a) \sqcap f(b). \end{aligned}$$

Therefore, the meet of two distributive functions is also distributive, and the set of distributive functions is closed under meet. □

Lemma 5 *The set of nondecreasing functions on a lattice is closed under composition and meet.*

Proof: A function f is nondecreasing iff $a \sqsubseteq f(a)$. Say f_1 and f_2 are nondecreasing. Then

$$a \sqsubseteq f_1(a)$$

and

$$a \sqsubseteq f_2(a).$$

Therefore,

$$a \sqsubseteq f_1(a) \sqcap f_2(a)$$

and, without loss of generality,

$$a \sqsubseteq f_2(a) \sqsubseteq f_1(f_2(a)).\square$$

Note that a function f may be nondecreasing but not monotone; e.g., $f(x) = 2x$ if x is even, $3x$ if x is odd.

Lemma 6 *The set of nondecreasing monotone functions on a lattice is closed under composition and meet.*

Proof: Direct from Lemmas 3 and 5. □

Lemma 7 *The set of nondecreasing distributive functions on a lattice is closed under composition and meet.*

Proof: Direct from Lemmas 4 and 5. □

Lemma 8 *Nondecreasing functions are 1-semibounded.*

Proof: A function is 1-semibounded if, for all $x, y \in L$, $f(y) \sqsupseteq y \sqcap x \sqcap f(x)$ [MR91a]. If f is nondecreasing, then $f(y) \sqsupseteq y \sqsupseteq y \sqcap x \sqcap f(x)$. □

5.2.2 Convergence time for iterative lattice problems.

Lemma 9 *If a lattice L has a finite height h, and is used in a monotone data flow framework $D = (G, L, F, M)$ where $G = (N, E)$, then the worklist iterative algorithm [Hec77] applied to D where each node in N has an initial value of $\perp$ terminates after no more than $h|N|^2\lceil\log_2(|N|)\rceil$ meet operations and $h|N|^2$ edge function evaluations in the worst case. The cost to maintain the worklist during this time is $\mathcal{O}(h|N|\log(|N|))$ in the worst case. This result also holds where each node in N has an initial value of $\top$.*

Proof: Suppose each node in N is given an initial value of $\perp$. All nodes in N are initially placed on the worklist, and can only increase in value or remain at $\perp$. If all nodes remain at $\perp$, then the worklist algorithm terminates after $|N|$ node visits and $|E|$ edge function evaluations. If any node n increases in value, then all its children are placed on the worklist. The process continues until the worklist is empty.

If any node m has its value v_m increased, then for all edges $e = (m, n)$ leaving n, $f_e(v_m)$ must also be increased (by the definition of monotonicity for f_e.) Since v_n is the meet of $f_e(v_{m_i})$ for all incoming edges (m_i, n), increasing any m_i will thus increase n or leave it unchanged. Since $v_n = \perp$ initially for all $n \in N$, each v_n may thus only increase or remain unchanged throughout the execution of the worklist algorithm.

Each node n may have its value increased a maximum of h times. In the worst case, each time a node's value increases, we may have to evaluate $|N|$ edge functions (if N is a complete graph and self-loops are allowed) for edges to n's children. We must also meet $\lceil \log_2(|N|) \rceil$ values at each child node of n to decide whether to place that child on the worklist. (Here we do incremental updates to the values of the children of n and retain partial meet values for edges entering each node in a height balanced tree structure.) Since the graph G contains a total of $|N|$ nodes, we must thus do a maximum of $h|N|^2$ edge function evaluations and $h|N|^2\lceil \log_2(|N|) \rceil$ meet operations.

Assuming that the worklist is maintained as a heap, each insert operation and each delete operation will require a maximum of $\mathcal{O}(\log(|N|))$ time. We must do an insert every time a node increases in value, and a delete every time we evaluate the edge functions out of a node whose value has increased. Thus we must do a total of $\mathcal{O}(h|N|\log(|N|))$ work during execution of the worklist algorithm to maintain the heap.

We may make the same statement if we initialize all v_n to $\top$, in which case each v_n will be monotonically nonincreasing during the execution of the worklist algorithm. □

5.3 Tarjan's path problem methods.

Tarjan [Tarj81a] gives an algorithm for solving global flow analysis problems which are represented as instances of the *all pairs path expression* problem. In this representation, *path expressions* $P(u, v)$, or regular expressions representing all paths from each node u to node v in the graph $G = (V, E, s)$[2], are constructed. Then the path expressions are mapped into the global flow analysis problem of interest, which is solved by substituting $\perp$ at the source vertex s of G. The result for each vertex

[2]We use Tarjan's notation system in this section.

v of G is a solution to the data flow problem at v.

Constructing the set of path expressions in a general graph takes $\mathcal{O}(m\alpha(m,n)+t)$ time [Tarj81b], where $m = |V|$, $n = |E|$, t is the time required to find the path expressions for the *dominator strong components* of G, and α is the inverse Ackermann function. Deriving the dominator tree of G takes time $\mathcal{O}(m\alpha(m,n))$ [LT79]. From these, the path expressions of the dominator strong components of G must be found by applying the algorithm recursively to each component. Assuming that no graph G of interest is such that all of its vertices are involved in the same dominator strong component and that $\alpha(m,n)$ is essentially constant, the time needed to construct the path expressions is bounded by $\mathcal{O}(m^2)$. The length of the path expression produced by this procedure is $\mathcal{O}(m\alpha(m,n)+l)$, where l is the sum of the lengths of the path expressions of the dominator strong components of G. Thus, the length of the path expression is bounded by $\mathcal{O}(m^2)$ under the same assumptions.

The mapping from the path expressions to the data flow problem may require construction of *pseudotransitive closure functions*, $f^{@}(P)$ as an approximation of the transitive closure function $f^{*}(P)$ for the path expressions P. For data flow frameworks which are *k-bounded*, we can compute an approximation to $f^{@}(P)$ in $\mathcal{O}(\log(k))$ time, or we can compute $f^{*}(P)$ itself in $\mathcal{O}(k)$ time. Since k is constant for a given lattice framework, we may regard this time as a constant multiplier on the execution time of the algorithm. The substitution step thus takes $\mathcal{O}(km^2)$ time, assuming that the time to meet is constant in the parameters of G.

Lemma 10 *Under the assumption that Tarjan's algorithm terminates, a solution to a 1-semi-bounded (or 2-bounded) lattice problem can be found in time*

$$\mathcal{O}(m^2(t_f + t_\sqcap))$$

in the worst case, where t_f is the time for one edge function application and $t_\sqcap$ is the time for a meet operation.

Proof: The time to construct the path expressions for a program is $\mathcal{O}(m^2)$, under the assumption that the inverse Ackermann function is essentially constant. The path expression itself is of length $\mathcal{O}(m^2)$. Therefore, mapping the data flow problem to the path expression will require $\mathcal{O}(m^2)$ meet operations and edge function applications, when the data flow problem is k-bounded and that k is constant in m and n.

1-semiboundedness implies 2-boundedness [MR91a]. Therefore, the lemma holds. □

We note that Tarjan's methods may have high overhead costs, and are not frequently used in practice. However, they allow us to reduce the asymptotic upper bound on the complexity of the

lattice problems used in deadlock detection.

5.4 Kam-Ullman "rapid" problems in irreducible graphs.

Kam and Ullman [KU76] describe a version of Kildall's iterative node-listing algorithm which halts in at most $d(G)+3$ iterations over the nodes of a graph G, where $d(G)$ is the *loop connectedness* of G, or the largest number of back edges found in any cycle-free path in G (with respect to the depth-first spanning tree of G from which the node listing is derived). The functions in the lattice framework are required to be both 1-semibounded and distributive for the Kam-Ullman time complexity result to hold (and to be monotone and closed under composition and meet). The lattice frameworks for which the Kam-Ullman result hold are called *rapid*, and include such classical data flow problems as reaching definitions and live variables.

For reducible control flow graphs, $d(G)$ is equal to the maximum loop nesting depth of any statement in the program. However, we cannot in general assume that the sync hypergraph is reducible; in fact, if the program can deadlock, then its sync hypergraph must be irreducible (because it contains the deadlock cycle and a path from the program begin node to each head node in the cycle). For general graphs G, we conservatively assume that G may be fully connected, and that $d(G)$ is therefore proportional to the number of nodes in G.

Lemma 11 *Given a rapid lattice framework $D = (G, L, F, M)$ where $G = (V, E)$ is a general graph, Kildall's algorithm converges in $\mathcal{O}(|V||E|)$ edge function evaluations and $\mathcal{O}(|V|^2 \log(|V|))$ meet operations.*

Proof: From Theorem 2 of [KU76], Kildall's algorithm halts in no more than $d(G)+2$ iterations on a rapid problem. In the initialization step of Kildall's algorithm, the edge functions into each node are initially evaluated and a meet of the edges into each node is taken. Thus there are $|E|$ edge function evaluations and, in the worst case, $\mathcal{O}(|V|\log(|V|))$ meet operations. The same work is done on each iteration. Thus, a total of $\mathcal{O}(d(G)|E|)$ edge function evaluations and $\mathcal{O}(d(G)|V|\log(|V|))$ meet operations are performed. Since $d(G) = |V| - 1$ in the worst case, we have $\mathcal{O}(|V||E|)$ edge function evaluations and $\mathcal{O}(|V|^2 \log(|V|))$ meet operations in the worst case on a general graph. □

Unfortunately, we cannot use the results of Lemma 11 in the timing analysis of the lattice frameworks used in deadlock analysis, since none of these frameworks turns out to use a distributive function space.

6 Computing *CHT.*

In this section, we discuss the methods of obtaining *CHT* information. These include *B4* analysis, pinning analysis, critical section analysis, and remote procedure call analysis.

6.1 *B4* analysis.

Given a pair of statements in the program, it may be possible to show that all instances of one statement must always happen before any instance of the other. We do this by establishing that there must always be a chain of control edges and/or synchronizations from one to the other, if both statements occur. This form of analysis is called *B4 analysis*, and is closely related to the *SCPreserved* analysis in loopless programs performed by Callahan and Subhlok [CS88].

More recently, Duesterwald and Soffa [DS91], [Dues91] formulated a *before* analysis of Ada-like programs; our analysis is similar to theirs in principle, although they use a different graph representation and execution model. Our sync hypergraphs extend their representation by including the `accept-do` construct, selective wait, and conditional entry calls[3]. Though [Dues91] proposes to eliminate spurious sync edges using their ordering relation, they do not mention doing so iteratively, or in combination with other refinements.

However, [DS91] represents recursion and process creation, while we require that subroutines be fully inlined and all processes created at the start of the program. While we find little or no use of recursive process creation in practice, we hope to eliminate these restrictions in future research.

Lastly, while [CS88] and [DS91] presented their analyses as sets of data flow equations, we present *B4* more formally here as a lattice framework, and derive its convergence time and accuracy properties based on lattice-theoretic arguments.

6.1.1 *B4* lattice framework.

A *B4 summary* is defined as a pair (C, S) where C and S are sets of nodes in a *CHT* form sync hypergraph. Thus the lattice space L_{B4} is:

$$L_{B4} = 2^{N_{CHT}} \times 2^{N_{CHT}}.$$

Intuitively, in a *B4* summary (C, S) for node x, C represents the *B4* information propagating from x's control predecessors, while S represents *B4* information from the synchronizations immediately

[3] While the authors refer to selective wait, they do not provide examples of its representation.

preceding the start of any instance of x. The meet operation is element-wise set intersection; thus, the $\sqsubseteq$ operator is element-wise set containment. For convenience, denote $(\emptyset, \emptyset)$ as $B\not4_\perp$ and (N_{CHT}, N_{CHT}) as $B\not4_\top$.

Lemma 12 *The height of the $L_{B\not4}$ lattice space is $2|N_{CHT}| + 1$.*

Proof: The $\sqsubseteq$ operator is defined for $L_{B\not4}$ as element-wise set containment. If $a = (C_a, S_a), b = (C_b, S_b) \in L_{B\not4}$ are such that $a \sqsubset b$, then either $C_a \subset C_b$ and $S_a \subseteq S_b$, or $C_a \subseteq C_b$ and $S_a \subset S_b$. Thus, there must be at least one element in S_b not contained in S_a, or at least one element in C_b not contained in C_a. The longest possible ordered chain of elements of $L_{B\not4}$ would thus start with $B\not4_\perp$, and be such that either C_i or S_i (but not both) has one more node than C_{i-1} (respectively S_{i-1}). The length of such a chain is thus the total maximum cardinality of C and S plus one. □

6.1.2 Completor edges.

Before defining the function space for the $B\not4$ lattice framework, it is useful to discuss the propagation of $B\not4$ information along sync hyperedges. Consider node x in Figure 11. We might expect to obtain some information on the synchronizations that precede x by examining nodes connected to the sync hyperedges reaching its control predecessors, a_i^4 and s^5.

In doing so, we can derive more information from s^5 than we can from a_i^4. This is because x is nested inside the accept statement corresponding to a_i^4. When x is reached in control flow, some such signaling node (say s^1) is in state ENGAGED(a_o^4), and would thus actually be *concurrent* with x. On the other hand, s^5, and the accept node pair it does rendezvous with, must complete execution before x starts.

For the sake of a simple and uniform treatment of rendezvous completion, we choose to examine only the most immediate control ancestors of x which have completed their rendezvous before x is reached in control flow. This set of control ancestors is called the *completors* of x, denoted *Completors*(x). In Figure 11, *Completors*(x) = $\{s^5, s^6, a_o^3\}$. Since one of these must have completed rendezvous before x is reached, we know that $n \in B\not4(x)$, since n is in the $B\not4$ sets (in boxes) of all the nodes which complete rendezvous along with the completors of x.

For each node $v \in$ *Completors*(x), and each accept-out or signaling node w which may complete a rendezvous with v, we can imagine a directed *completor edge* (w, x). The presence of such an edge indicates that w may be a "synchronization ancestor" of x; the intersection of the $B\not4$ sets of all such synchronization ancestors must be in $B\not4(x)$. We denote the set of completor edges as $E_{Completors}$.

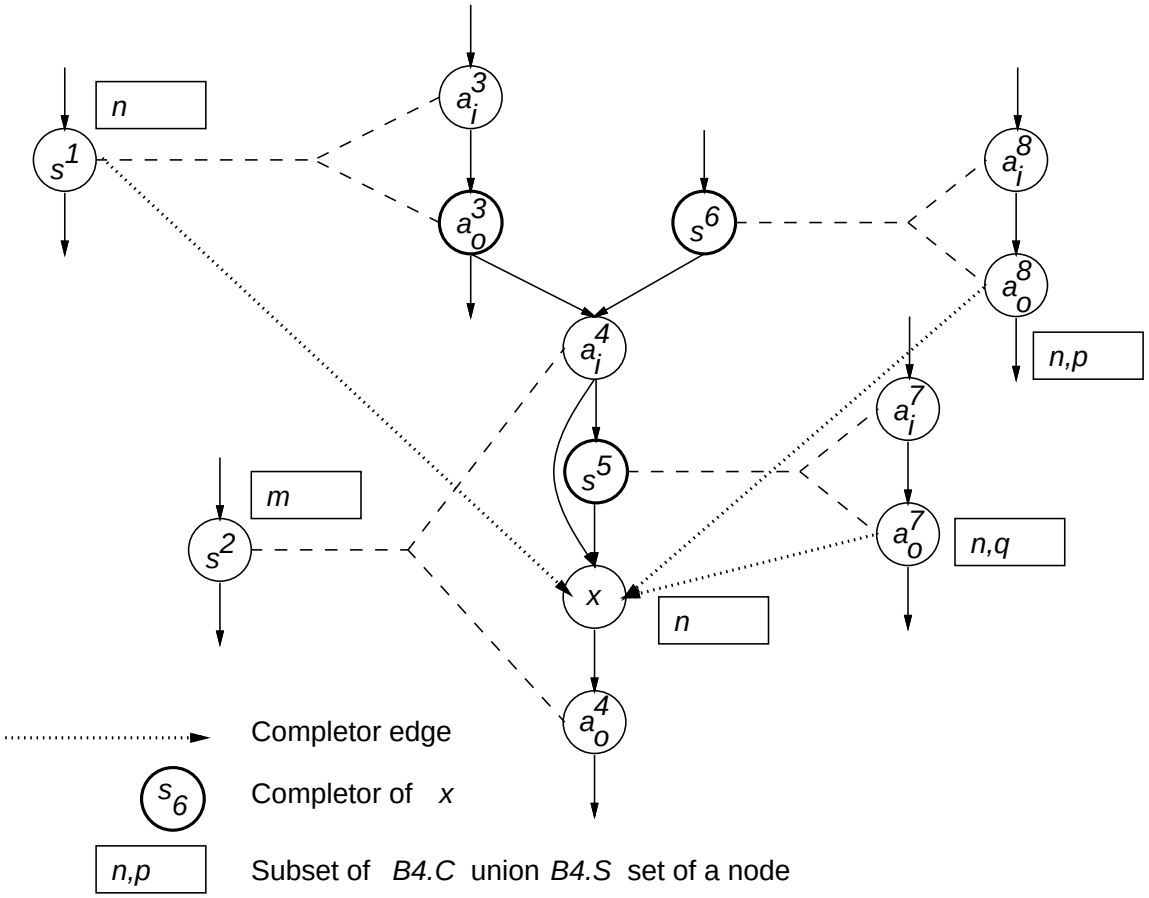

Figure 11: Completors of node x and completor edges to x.

If $b \in Completors(x)$, then there is a path from the start of the program to x along which no rendezvous is completed. In this special case, we establish a completor edge (b, x) to propagate an empty $B4$ set to x through the completor edges. Thus, no nodes are known to complete before x because of rendezvous.

We may, of course, be able to derive further information from the nodes which rendezvous with the `accept` statements within which x is nested. For instance, we know that m must be in $B4(x)$, since m is in $B4(s^2)$ and s^2 is the only node that can rendezvous with a_i^4. However, making use of this information would require introducing a potentially unbounded group of "engagor edge" types, one for each scope level. We feel that doing so would complicate the lattice beyond reason. This potential source of information may be a subject for later investigation.

As the estimate of CHT is refined, we can refine the set of completor edges using new CHT information. If $m \in Completors(n)$ and $p \in CHT(m)$, then $(p, n) \notin E_{Completors}$.

6.1.3 Function space.

We map a control or completor edge $e = (m, n)$ to a function f_e as follows:

$$f_e((C, S)) = (C \cup S \cup K_e, N_{CHT})$$

if e is a control edge, or

$$f_e((C, S)) = (N_{CHT}, C \cup S \cup K_e),$$

if e is a completor edge. For either edge type,

$$K_e = CReach(m) \cup \{m\} - \{p : m \in CReach(p)\},$$

i.e., the set of nodes q such that there is a control path from q to m (possibly of zero length), but no control path of length > 0 from m to q.

Lemma 13 *The edge functions f_e are nondecreasing and monotone.*

Proof: By the definition of the $\sqsubseteq$ operator as element-wise set containment, each f_e is clearly nondecreasing. Also, for any $a = (C_a, S_a), b = (C_b, S_b) \in L_{B4}$ such that $a \sqsubseteq b$, by definition of $\sqsubseteq$, $C_a \subseteq C_b$ and $S_a \subseteq S_b$. Therefore, for f_e of the form

$$f_e((C, S)) = (C \cup S \cup K_e, N_{CHT}),$$

we have

$$f_e(a) = (C_a \cup S_a \cup K_e, N_{CHT})$$

and

$$f_e(b) = (C_b \cup S_b \cup K_e, N_{CHT}).$$

We know that, since $C_a \subseteq C_b$ and $S_a \subseteq S_b$,

$$C_a \cup S_a \cup K_e \subseteq C_b \cup S_b \cup K_e.$$

Therefore, $f_e(a) \sqsubseteq f_e(b)$ and so f_e is monotone. An analogous argument applies for f_e of the form for completor edges, $f_e((C, S) = (N_{CHT}, C \cup S \cup K_e)$. $\Box$

Lemma 14 below shows that the $B4$ lattice framework is not rapid. We therefore cannot use the time complexity results of Lemma 11 for the $B4$ problem.

Lemma 14 *The functions f_e are not distributive in general.*

Proof: Consider $a = (\emptyset, \{n\})$ and $b = (\{n\}, \emptyset)$. Let $f_e((C, S)) = (C \cup S \cup K_e, N_{CHT})$ where $n \notin K_e$. Thus, $a \sqcap b = (\emptyset, \emptyset)$ and $f_e(a \sqcap b) = (K_e, N_{CHT})$. But $f_e(a) = f_e(b) = (K_e \cup \{n\}, N_{CHT}) = f_e(a) \sqcap f_e(b)$. In this instance, $f_e(a \sqcap b) \sqsubset f_e(a) \sqcap f_e(b)$.

A similar argument holds for edge functions of the form $f_e((C,S)) = (N_{CHT}, C \cup S \cup K_e)$. □

Lemma 15 below shows that the $B4$ lattice framework correctly models the semantic notion we wish to capture.

Lemma 15 *Let $B4_i$ be the i'th distinct value of $B4$, during execution of an iterative algorithm based on the lattice framework for $B4$ (with $B4_0(n) = (\emptyset, \emptyset)$ for all n). If $B4_i(n) = (C,S)$ and $m \in C \cup S$, then all instances of m must complete before any instance of n can start.*

Proof: By induction on i. As a notational convenience, let $UnionB4((C,S)) = C \cup S$.

Basis: Since $B4_0(n) = (\emptyset, \emptyset)$, the condition of the lemma is trivially satisfied.

Induction step: Suppose the lemma holds for $B4_{i-1}$. Let $B4_i(n) = (C,S)$, and suppose that $m \in C \cup S$.

If $m \in UnionB4(B4_{i-i}(n))$ then the condition of the lemma is satisfied by the induction hypothesis.

If $m \notin UnionB4(B4_{i-1}(n))$, then at least one of the following must be true:

1. $m \in C$. Thus,

$$m \in \bigcap_{p \in CPreds(n)} UnionB4(B4_{i-1}(p)) \cup k_{(p,n)}.$$

   For all control predecessors p of n, either $m \in UnionB4(B4_{i-1}(p))$, or $m \in k_{(p,n)}$, or both. If $m \in UnionB4(B4_{i-1}(p))$ for a particular predecessor p, then all instances of m must complete before any instance of p starts, by the induction hypothesis. Therefore, for any instance of n entered through p, all instances of m must have completed before any instance of n starts.

   If, for a particular predecessor p, $m \in k_{(p,n)}$, then there is a control path from m to p, but no path of length > 0 from p to m. Therefore, either $m = p$ and m has only one instance, or all instances of m must complete before any instance of p start. In either case, for any instance of n entered through p, all instances of m must complete before any instance of n starts.

   Thus, if $m \in C$, then all instances of m must complete before any instance of n can start If this is true, then obviously $m \in CHT_{perf}(n)$ and $n \in CHT_{perf}(m)$.

2. $m \in S$. Thus,

$$m \in \bigcap_{(p,n) \in E_{Completors}} UnionB4(B4_{i-1}(p)) \cup k_{(p,n)}.$$

   For all completor edges (p,n), either all instances of m complete before any instance of p starts, or $m = p$ and m has a single instance. Thus, n cannot be entered in control flow until after all instances of m have terminated.

Thus, if $m \in S$, then all instances of m must complete before any instance of n can start.

3. n either has no control edges entering it, or has no completor edges entering it. This only occurs if n is unreachable, i.e., no instances of n ever execute. Regardless of whether m executes or not, the conditions of the lemma are vacuously satisfied.

Therefore, if $m \in C \cup S$, then all instances of m must complete before any instance of n can start. □

The function space $F_{B4} = L_{B4} \rightarrow L_{B4}$ is the transitive closure of all possible edge functions under meet and function composition.

Lemma 16 *F_{B4} is closed under meet and function composition, and is 1-semibounded.*

Proof: Any set of nondecreasing monotone functions is closed under meet and function composition, by Lemma 6, and is 1-semibounded by Lemma 8. □

Lemma 17 *The time for either a meet operation in L_{B4} or an edge function evaluation for $f_e \in F_{B4}$ is $\mathcal{O}(|N|)$.*

Proof: If sets of nodes are represented as bit maps of length $|N|$, then set union or intersection takes $\mathcal{O}(|N|)$ time. Evaluation of f_e takes two set union operations; meet requires two set intersection operations. □

Lemma 18 *The total time for convergence of a B4 lattice problem using the worklist iterative algorithm is $\mathcal{O}(|N_{CHT}|^3 \log |N_{CHT}|)$ in the worst case.*

Proof: Direct from Lemmas 9, 12, and 17. □

Lemma 19 *The total time for solution of a B4 lattice problem using Tarjan's path expression algorithm is $\mathcal{O}(|N_{CHT}|^3)$ in the worst case.*

Proof: Direct from Lemmas 10 and 17. □

6.1.4 Using *CHT* information to eliminate completor edges.

Recall from Section 6.1.2 that completor edges (w, x) are drawn from nodes w which can complete rendezvous with the completors of x. If w cannot happen together with any completor of x with which w can rendezvous, then the completor edge (w, x) may be eliminated. Doing so will improve the accuracy of the *B4* solution.

Incremental update of *B4* sets. The following lemma shows that the *B4* solution does not decrease as we eliminate completor edges. This allows us to update *B4* sets incrementally after eliminating completor edges, rather than re-solving *B4* from the beginning.

Lemma 20 *If* $E_{Completors}{}' \subseteq E_{Completors}$, $B4_i$ *is the i'th distinct value of an iterative solution of* $B4$ *on the graph* $(N, E_C, E_{Completors})$, *and* $B4'_i$ *is the i'th distinct value of an iterative solution on the graph* $(N, E_C, E_{Completors}{}')$, *then* $B4(n)' \sqsupseteq B4(n)$ *for all nodes* n.

Proof: By induction on i.

Basis: $i = 0$. $B4'(n) = B4(n) = (\emptyset, \emptyset)$ for all n.

Inductive step: Suppose $B4'_{i-1}(n) \sqsupseteq B4_{i-1}(n)$ for all n. By monotonicity of the edge functions (Lemma 13), $f_e(B4'_{i-1}(n)) \sqsupseteq f_e(B4_{i-1}(n))$ for any node n and any edge e.

Since $E_{Completors}{}' \subseteq E_{Completors}$, $B4'_i(n)$ is a meet of at most as many summaries as $B4_i(n)$. Each of those summaries in $B4'_{i-1}$ must be at least as great as the corresponding summaries in $B4_{i-1}$. Thus, $B4'_i(n) \sqsupseteq B4_i(n)$. □

6.1.5 Augmenting *CHT* information using *B4* analysis.

Once we have found a solution to the $B4$ lattice problem, we can immediately extract CHT information from it. Given the $B4$ and CHT sets of a program, we form

$$CHT' = CHT \cup B4 \cup B4^T.$$

The update operation thus requires two array union operations and one array transpose.

Algorithm 1 *Augment CHT information using B4 information.*

Input:

- The CHT form sync graph $G = (N, E_C, E_S)$.
- The set *CompleteRend*. *CompleteRend*(n) is the set of nodes which can complete a rendezvous with n. If n is a signaling node, then *CompleteRend*(n) is the set of accept-out nodes of the same signal type; if n is an accept-out node, then *CompleteRend*(n) is the set of signaling nodes of the same signal type. Otherwise, *CompleteRend*(n) is empty.
- $CHT \subseteq CHT_{perf}$.

Output:

- CHT' such that $CHT' \supseteq CHT$ and $CHT' \subseteq CHT_{perf}$.

Procedure:

1. Using G and CHT, construct $E_{Completors} = Completors \times (CompleteRend - CHT)$.
2. Let $G' = (N, E_C, E_{Completors})$.
3. Find the least fixed point $B4$ of the lattice framework $(G', L_{B4}, F_{B4}, M_{B4})$. M_{B4} is the mapping function described in Section 6.1.3.
4. $CHT' = CHT \cup B4 \cup B4^T$.

Lemma 21 (Correctness of $B4$.) *If $CHT \subseteq CHT_{perf}$, then $CHT \subseteq CHT' \subseteq CHT_{perf}$ following Algorithm 1.*

Proof: $CHT' = CHT \cup B4 \cup B4^T$. From Lemma 15, $B4 \cup B4^T \subseteq CHT_{perf}$. Thus, $CHT \subseteq CHT_{perf} \Rightarrow CHT' \subseteq CHT_{perf}$. Obviously, $CHT \subseteq CHT'$. Therefore, $CHT \subseteq CHT' \subseteq CHT_{perf}$. □

Lemma 22 (Time bounds for $B4$ analysis.) *The total time required for $B4$ analysis using Algorithm 1 is $\mathcal{O}(|N_{CHT}|^3 \log(|N_{CHT}|))$ in the worst case using the worklist iterative algorithm to solve the lattice problem, or $\mathcal{O}(|N_{CHT}|^3)$ using Tarjan's algorithm.*

Proof:

The set of completor edges $E_{Completors}$ can be constructed at a cost of $\mathcal{O}(|N_{CHT}|^3)$ time. From Lemmas 18 and 19, the time to solve the $B4$ lattice problem is $\mathcal{O}(|N_{CHT}|^3 \log(|N_{CHT}|))$ using the worklist algorithm, or $\mathcal{O}(|N_{CHT}|^3)$ using Tarjan's algorithm. The time to perform the Boolean array transpose and union operations to form CHT' is $\mathcal{O}(|N_{CHT}|^2)$.

Thus, the total time required for $B4$ analysis is $\mathcal{O}(|N_{CHT}|^3 \log(|N_{CHT}|))$ in the worst case using the worklist iterative algorithm, or $\mathcal{O}(|N_{CHT}|^3)$ using Tarjan's algorithm. □

6.2 Widening and pseudotransitivity.

The CHT relation is not transitive. However, it admits two useful properties, which we call *widening* and *pseudotransitivity*.

Theorem 1 (Widening.) *If there exists a set S such that some member of S must always execute together with n, then the nodes in $\bigcap_{n' \in S} CHT(n')$ cannot happen together with n.*

Proof: Obvious. □

Thus, if we can discover some set of nodes S, such that some member of S must always happen together with n, we can improve the CHT information about n. In general, a small S yields the

greatest chance for adding information to CHT. Pinning, critical section, and remote procedure analysis make use of Theorem 1 by identifying such sets. Additionally, we can sometimes expand the CHT set without additional structural analysis:

Corollary 1 (Pseudotransitivity.) *The nodes in the set*

$$\bigcap_{n' \in N_{CHT} - CHT(n) - \{n\}} CHT(n')$$

cannot happen together with n.

Proof: The set $N_{CHT} - CHT(n) - \{n\}$ contains all nodes that may happen together with n. This becomes the set S of Theorem 1. □

Corollary 1 allows us to take the *pseudotransitive closure* of the CHT relation, in the obvious way. However, we do not do this explicitly. In pinning, critical section, and remote procedure analysis, we attempt to identify, for each node n, some set $S \subset N_{CHT} - CHT(n) - \{n\}$ satisfying Theorem 1. If no such set can be identified for n, we use $N_{CHT} - CHT(n) - \{n\}$ to refine CHT by pseudotransitivity as a by-product of the analysis.

We can further sharpen the accuracy of Corollary 1 by considering pseudotransitivity on a task-wise basis:

Corollary 2 (Task-wise pseudotransitivity.) *For any task T and node $n \notin T$, the nodes in the set*

$$\bigcap_{n' \in T - CHT(n)} CHT(n')$$

cannot happen together with n, where $CHT(n) \subseteq CHT_{perf}(n)$.

Using Corollary 2 will usually require one pseudotransitivity computation per task. In certain cases (e.g., Section 6.3), CHT is refined on a task-wise basis, so we can use task-wise pseudotransitivity without an excessive time penalty. In general, though, using Corollary 2 would increase the cost of a refinement algorithm by a factor of $\mathcal{O}(|Tasks|)$ time.

Algorithm 2 *Widening CHT using sets of nodes which must happen together with other nodes. Denoted $CHT' = Widen(CHT, S, T)$.*

Input:

- The CHT array.
- An array S, such that, for each node n, either $S(n) = \emptyset$ or some element of $S(n)$ must happen together with each node n.

- A vector T of nodes, such that, for each node $n \in N - \{b, e\}$, n must execute concurrently with members of T. To apply task-wise pseudotransitivity, T will be the set of nodes of a single task. To apply the more general pseudotransitivity of Corollary 1, $T = N$.

Output:

- The array $CHT' \supseteq CHT$.

Procedure:

1. $Mask \Leftarrow (S \times ONE)$.

2. $Tarray \Leftarrow (ONE_{Vector} \times T) \cap (-Mask)$.

   $Tarray$ is an array which duplicates T on every row for which S has no set bits.

3. $MustHT \Leftarrow (S \cup Tarray) - CHT - I$.

   For each node n, $MustHT(n)$ is a set of nodes other than n such that some instance of at least one member of $MustHT(n)$ must (and can) happen together with any instance of n. If $S(n) = \emptyset$, then $MustHT(n) = T - CHT(n) - \{n\}$, to apply pseudotransitivity. If $S(n) \neq \emptyset$, then $MustHT(n) = S(n) - CHT(n) - \{n\}$, a set smaller than (and thus refined over) the set implied by the pseudotransitivity relation on n.

4. $NewCHT \leftarrow -(MustHT \times (-CHT))$.

   $NewCHT(n)$ is the set of nodes which can't happen together with any node that must happen together with n.

5. $CHT' \leftarrow CHT \cup NewCHT \cup NewCHT^T$.

   Since $NewCHT$ is not necessarily symmetric, we must union CHT with both $NewCHT$ and its transpose. Obviously, $CHT'(n) \supseteq CHT(n)$ for all n.

Lemma 23 (Correctness of the widening algorithm.) *If $CHT \subseteq CHT_{perf}$ and the parameters S and T are correctly specified to Algorithm 2, then $CHT \subseteq CHT' \subseteq CHT_{perf}$.*

Proof: From the discussion in the algorithm definition. □

Lemma 24 (Time bounds for the widening algorithm.) *Algorithm 2 requires $\mathcal{O}(|N_{CHT}|^3)$ time to execute in the worst case.*

Proof: The most expensive operation in Algorithm 2 is a multiplication of two $|N_{CHT}| \times |N_{CHT}|$ arrays; this requires $\mathcal{O}(|N_{CHT}|^3)$ in the worst case. □

Since Algorithm 2 must be used iteratively in some cases, it is also important to know the cost of updating $Widen(CHT, S, T)$ incrementally, as a function of the number of clear bits in any "old" version of CHT that are set in the subsequent "new" version.

Lemma 25 (Incremental time bounds for the widening algorithm.) *The cost of incrementally updating a one-bit change in the CHT parameter of* $Widen(CHT, S, T)$ *is* $\mathcal{O}(|N_{CHT}|)$.

Proof: Suppose that bit (m, n) is zero in the old CHT array and one in the new CHT array. We need not update $Mask$, since it depends only on S. However, we will need to clear bit (m, n) in $MustHT$. Similarly, we will have to clear bit (m, n) in $-CHT$.

$NewCHT$ may be updated by updating row m and column m alone. This update requires $\mathcal{O}(|N_{CHT}|)$ bit operations. From this, we may recompute CHT' in $\mathcal{O}(|N_{CHT}|)$ bit operations. Thus, it takes $\mathcal{O}(|N_{CHT}|)$ bit operations to update a one-bit change to the CHT parameter. □

6.3 Pinning analysis on successors of a rendezvous.

Figure 12 shows the sync graph of a simple program. Here, m can rendezvous only with p, and n can rendezvous only with q. For the sake of a simple explanation, we show each accept clause as a single node in the present discussion.

Suppose that we have hypothesized n as a head node. $CPreds(n) = \{m\}$, so some successor of a rendezvous partner of m must have been placed on the execution wave after m completed its rendezvous.

The distinguished program begin node b is not in $CPreds(n)$. Let the set $Partners(n)$ be the set of signaling and accept-in nodes which are the most immediate successors of nodes which can rendezvous with $CPreds(n)$. In Figure 12, $Partners(n) = \{q\}$. Since no node in the sync graph other than n can rendezvous with q, and since q has no `else` clause, unguarded `delay` alternative, or sister which can rendezvous with nodes other than n, we know that q must remain on the execution wave until it does rendezvous with n. Furthermore, since q can rendezvous with n, we know that n cannot be a head node of a valid deadlock cycle.

Formalizing the above arguments, we have the following:

Lemma 26 *Whenever a node* n *is on the execution wave, at least one node* $p \in Partners(n)$ *must be placed on the wave at the same time as* n*, in the execution model.*

Note that the partner need not remain on the wave, as Figure 13 shows. Here, although q

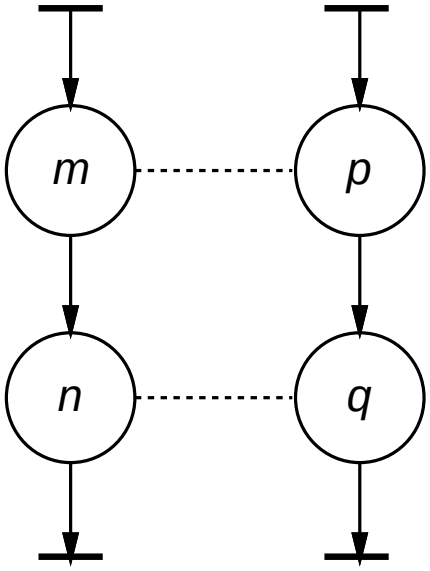

Figure 12: A simple sync graph. The thick horizontal bars represent the bounds of loop bodies.

appears on the execution wave concurrently with n, it may rendezvous with r and leave the wave. Therefore, q need not be present at all times n is. Node q is thus a "vanishing partner" of n.

The reasons an accept-in or signaling node n may leave the wave include the following:

- n completes a rendezvous.
- n has a sister that completes a rendezvous.
- n has a `delay` alternative, for which the guard (if any) is open, and the delay period expires.
- n has an `else` clause which is taken. This may happen even if n can rendezvous with another node m on the wave. To see why this may happen in an actual program, consider what would happen if there was a long interval between the time n is reached in control flow and the time m is reached.

Nodes that have a `delay` alternative or an `else` clause are called "slippery" nodes, since they can slip off the execution wave without rendezvousing. In contrast, nodes that cannot slip off the execution wave without a rendezvous are called "sticky." In the construction of the sync hypergraph, if m can slip off the execution wave and allow n to execute, then there are control edges from m's predecessors to n.

Lemma 27 *If a node n is on the execution wave, at least one member of Partners(n) must appear on the wave until the first rendezvous occurs with n or a node in Partners(n).*

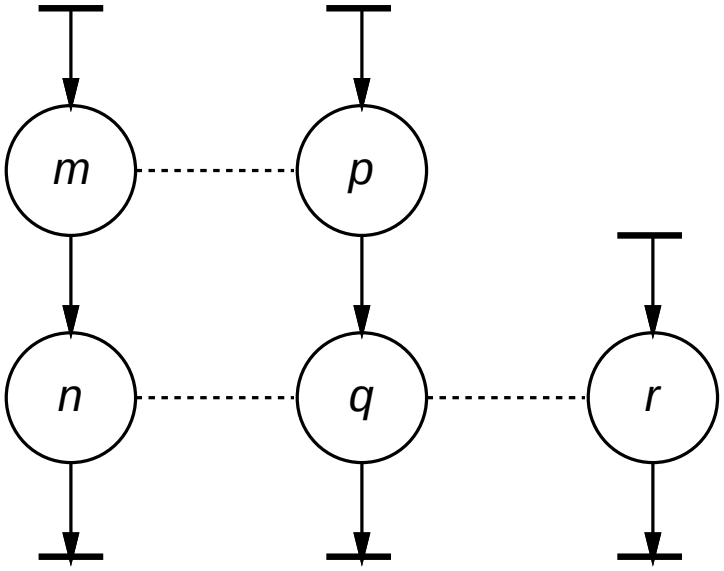

Figure 13: A "vanishing partner" of n.

This, plus information about which nodes can rendezvous, gives us some information which can be used to iteratively add information to CHT:

Lemma 28 *If all nodes in $Partners(n)$ can rendezvous only with n or with sisters of n, or with nodes in $CHT(n)$, then some member of $Partners(n)$ is on the execution wave at all times that n is.*

If the above condition is true, we say that *n pins its partners to the wave.* Thus,

Lemma 29 *If n pins its partners to the wave and*

$$p \in \bigcap_{q \in Partners(n)} CHT(q)$$

then $p \in CHT(n)$.

We can further derive information about possible head nodes:

Lemma 30 *If n (or one of its unguarded sisters) can rendezvous with all nodes $q \in Partners(n)$ (or with an unguarded sister of each q), and if n pins its partners to the wave, then n cannot be a head node.*

Proof: If such is the case, then at any time n is on the wave, then so is another node that can rendezvous with n. □

Node	*CPreds*	*Partners*	Head node?	*CHT*
b	$\emptyset$	$\emptyset$	No	r, s, t, u, v, w
r	b	u	No	b, s, t, v, w
s	r	v, e	Yes	b, r, t
t	s	w	No	b, r, s, u, v
u	b	r	No	b, s, t, v, w
v	u	s, e	Yes	b, u, w
w	v	t	No	b, r, s, u, v

Table 2: Pinning analysis - iteration 1. The "Head node?" column for a node n contains "No" if n's partners all do rendezvous only with n and members of $CHT(n)$, "Yes" otherwise.

6.3.1 Example of pinning analysis.

Figure 14 shows a sync graph for which we will conduct pinning analysis. Table 6.3.1 shows the results of the first iteration of pinning analysis for Figure 14. Initially, the *CHT* set for every node contains the other nodes in its task.

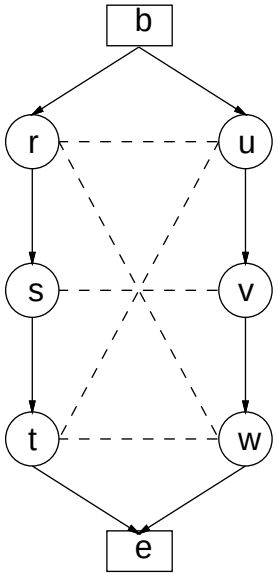

Figure 14: Pinning analysis example.

After the first iteration, we notice that $w \in CHT(r)$ and $u \in CHT(t)$. This gives us the opportunity to eliminate two sync edges which correspond to those rendezvous that are never taken. Figure 15 shows the resulting sync graph, and Table 6.3.1 shows the results of the second

Node	*CPreds*	*Partners*	Head node?	*CHT*
b	$\emptyset$	$\emptyset$	No	r, s, t, u, v, w
r	b	u	No	b, s, t, v, w
s	r	v	No	b, r, t, u, w
t	s	w	No	b, r, s, u, v
u	b	r	No	b, s, t, v, w
v	u	s	No	b, r, t, u, w
w	v	t	No	b, r, s, u, v

Table 3: Pinning analysis - iteration 2.

iteration of pinning analysis.

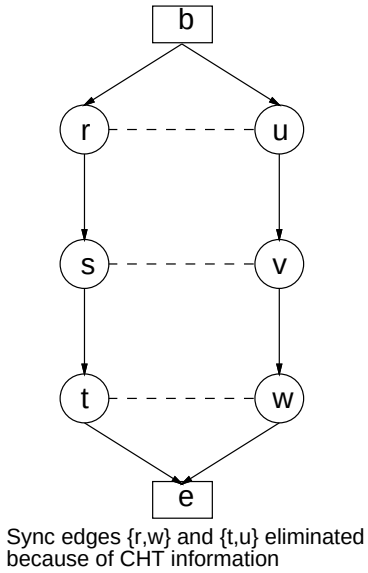

Figure 15: Pinning analysis example after second iteration.

We could have stopped pinning analysis after the first iteration, since at that time, only s and v were possible head nodes, and no valid deadlock cycles could have been found. In general, though, it seems more prudent to continue pinning analysis until stabilization, before attempting to find deadlock cycles.

6.3.2 Pinning analysis when the begin node is in $CPreds(n)$.

One special case of pinning analysis occurs when we are analyzing some node n, and the distinguished begin node b is in $CPreds(n)$. For example, see Figure 16. Here, $CPreds(n) = \{b, m\}$. If we regard b as a barrier synchronization between all tasks, then we might say that $Partners(n) = \{m, p, r, s\}$. Since r and s can rendezvous with each other, and can happen together with n, we can conclude that n does not pin its partners to the wave.

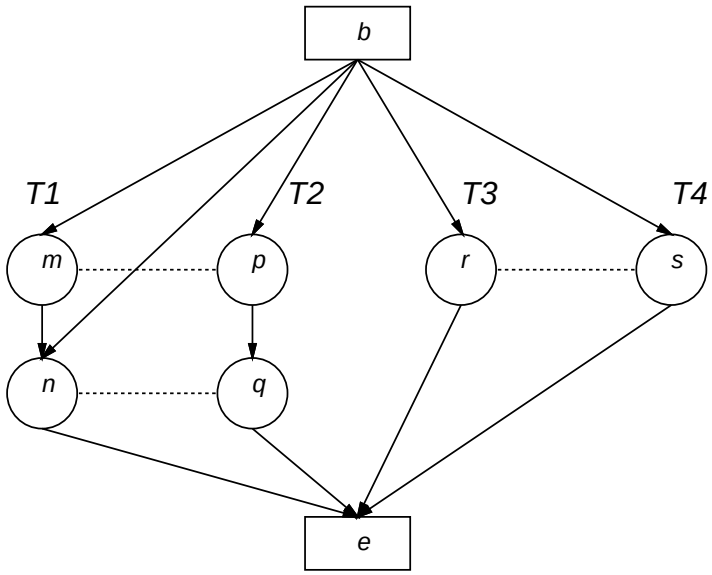

Figure 16: Pinning analysis where the distinguished begin node b is in $CPreds(n)$.

This, however, is too liberal a definition of the term "partner," and consequently we lose useful information. In this case, we do not find out that either p or q must be on the wave whenever n is. Since both p and q can rendezvous with n, n cannot be a head node.

We can solve this problem by partitioning the *Partners* set by task, as in the following definition:

$$Partners(n, T \neq Task(n)) = \left[\bigcup_{(m,n)\in E_c, \{m,p\}\in E_s, (p,q)\in E_c} q \right] \cup \left[\bigcup_{(b,n)\in E_c, \{b,q\}\in E_c, q\in T} q \right]$$

where the union over an empty set is taken to be empty. Semantically, $Partners(n, T)$ is the set of nodes which either:

- May start together with n when n is the control successor of a rendezvous node; or,
- Are in task T, and may start together with n when n is the control successor of b.

Lemma 31 *If the members of $Partners(n, T)$ can rendezvous only with n or with nodes that can't happen together with n, then n pins $Partners(n, T)$ to the wave.*

Proof: Whenever n starts execution, it must follow either the begin node or a rendezvous by one of its predecessors. In either case, some member of $Partners(n, T)$ is always placed on the wave whenever n starts. If all of these partners must rendezvous either with n or with some node that can't happen together with n, then any partner that starts with n must remain on the execution wave until n leaves it. □

6.3.3 Pinning and the `accept-do` and `select` constructs.

To this point, we have considered pinning analysis without reference to the `accept-do` or *select* constructs. In particular, we have used single accept nodes in our explanation, instead of accept in/out pairs. In this section, we examine these constructs in detail.

Predecessors of a node. If a node n has a signaling node s as its predecessor, the successor of an accept-out node a_o which could have been engaged to s must be a partner of n. Similarly, if n has an accept-out node a_o as a predecessor, then its partners must be the successors of signaling nodes s which could have been engaged to a_o.

However, if n has an accept-in node a_i as its predecessor, i.e., n is an accept-out node or is the first node executed in an accept-do clause, then the signaling node s which a_i engaged with must still be on the wave. In a sense, s is the partner of n; although s did not start at the same time n did, s is on the wave at the same time n is.

Although this last case is more explicitly handled by remote procedure call analysis, it has been seen experimentally that including s as n's partner, rather than the nodes which can rendezvous with $Completors(n)$, often has some accuracy benefits in pinning analysis of nodes inside accept-do clauses. In particular, if a_i is the only predecessor of n, then more accurate CHT information may be propagated from s to n than from $(Completors \times CompleteRend)(n)$ to n.

Sisters. If node n pins an accept-in node a_i with any sisters, the sisters (by definition) must be able to happen together with n. By construction of the sync hypergraph, the sisters of a_i must also be included as partners of n, since the sisters of a_i have the same control predecessors as a_i. Thus, we need not pay special attention to the sisters of partners when we perform pinning analysis.

6.3.4 Algorithm for pinning analysis.

Algorithm 3 *Pinning analysis.*

Input:

- The following arrays, of dimensions $|N_{CHT}| \times |N_{CHT}|$:
  - *Starters*.
  - *CPreds*.
  - *TakeOff*. *TakeOff* is, for each node n, the set of nodes which may take n off the execution wave (without necessarily completing a rendezvous). For signaling nodes, $TakeOff(n)$ includes only accept-out nodes of the same signal type; for accept-in or accept-out nodes, $TakeOff(n)$ includes signaling nodes of the same type as n. $TakeOff(n) \supseteq CompleteRend(n)$.
  - *CompleteRend*.
  - *InClause*. $InClause(n)$ is, for each accept-in node n, the set of nodes contained within the do clause of n (including n's accept-out node). If n is not an accept-in node, then $InClause(n)$ is empty.
  - *Sisters*.
  - *StartRendSig*. $StartRendSig(n)$ is the set of signaling nodes that can start a rendezvous with n. If n is an accept-in node, $StartRendSig(n)$ is the set of signaling nodes of the same signal type as n. If n is not an accept-in node, then $StartRendSig(n)$ is empty.
- The following arrays, of dimensions $|Tasks| \times |N_{CHT}|$:
  - *TaskFrontNodes*. $TaskFrontNodes(t)$ is the set of nodes in task t which may be placed on the execution wave at the start of the program.

Output: The array $CHT' \supseteq CHT$, of dimension $|N_{CHT}| \times |N_{CHT}|$.

Procedure:

1. $CHT' \Leftarrow CHT$.
2. For each task t do:
   (a) $Partners \Leftarrow (CPreds \times (CompleteRend - CHT') \times Starters) \cup (CPreds \times (StartRendSig - CHT')) - CHT'$.

The term $(CPreds \times (StartRendSig - CHT'))$ is a refinement that allows accept-out nodes to pin the signaling nodes which engage the accept clause.

(b) $Partners_t \Leftarrow Partners$.

(c) For each node n such that $b \in CPreds(n)$:

i. $Partners_t(n) \Leftarrow Partners(n) \bigcup TaskFrontNodes(t)$.

If n may be the front node of a task, its sets of partners include the front nodes of each other task. For each task t, we refine $Partners_t$ this way.

(d) $Pins \Leftarrow -(((Partners_t \times (TakeOff - CHT')) - I - CHT') \times ONE) \cap (Partners_t \times ONE)$.

$(Partners_t \times (TakeOff - CHT)) - I - CHT$ is, for each node n, the set of nodes which can happen together with both n and the partners of n, and which can take n's partners off the wave. The final result is an array where n's entire row is set if n has partners, and if no node other than n, which can happen together with n and any partner q of n, can take q off the wave.

(e) $PartnerClause_t = ((Partners_t \cap AInSameType) \times InClause) \cup Partners_t$.

That is, $PartnerClause_t$ contains, for each node n, all partners of n, and all nodes nested in any accept-do clause of any of n's partners that n can rendezvous with. $AInSameType$ is an array such that $a_i \in AInSameType(s)$ for signaling nodes s which may start a rendezvous with accept-in nodes a_i.

(f) $Partners_{Pinned} = (Pins \cap PartnerClause_t)$.

If n pins its partners, $Partners_{Pinned}(n)$ is the set of nodes in the partner's clause.

(g) $CHT' \Leftarrow Widen(CHT', Partners_{Pinned}, t)$.

Lemma 32 (Correctness of pinning analysis.) *If $CHT \subseteq CHT_{perf}$ and the other parameters of Algorithm 3 are correct, then $CHT \subseteq CHT' \subseteq CHT_{perf}$.*

Proof: From the discussion in the algorithm definition, each node n such that $Partners_{Pinned}(n) \neq \emptyset$ must execute concurrently with some member of $Partners_{Pinned}(n)$. This defines a set S which can be widened using Algorithm 2. The parameter T of Algorithm 2 is an arbitrary task. Therefore, we have correctly specified the parameters S and T of Algorithm 2.

From Lemma 23, if $CHT \subseteq CHT_{perf}$ and the parameters S and T are correctly specified for Algorithm 2, then $Widen(CHT, S, T) = CHT' \subseteq CHT_{perf}$. Therefore, if $CHT \subseteq CHT_{perf}$ and the other parameters of Algorithm 3 are correct, then $CHT' \subseteq CHT_{perf}$. □

Lemma 33 (Time bounds for pinning analysis.) *One application of pinning analysis requires* $\mathcal{O}(|Tasks||N_{CHT}|^3)$ *time.*

Proof: Algorithm 3 executes a loop which iterates $|Tasks|$ times. Within this loop are a fixed number of array copies, unions, subtractions, transposes, negations, and multiplications, and one call to *Widen*. Most of these operations require $\mathcal{O}(|N_{CHT}|^2)$ time; multiplication and the call to *Widen* are the exceptions, requiring $\mathcal{O}(|N_{CHT}|^3)$ time.

Within the task loop is an iteration over $|N_{CHT}|$ nodes, to create $Partners_t$. Each iteration of the inner loop requires at most $|N_{CHT}|$ time; the inner loop total time is thus $|N_{CHT}|^2$.

The total time per iteration of the task loop is thus $\mathcal{O}(|N_{CHT}|^3)$ time. An application of pinning analysis thus requires $\mathcal{O}(|Tasks||N_{CHT}|^3)$ time. □

6.3.5 Propagating pinning information.

It is possible, under certain circumstances, to propagate pinning information from a node to at least some of its successors. While we do not currently make use of this technique, we describe it here by means of an example. The effectiveness of propagating pinning information is a subject for future study.

Consider the situation of Figure 17. Here, node m pins a set of its partners $\{a, b, c\}$ to the wave, and does not rendezvous with any of them or their sisters. Thus, at least one member of $\{a, b, c\}$ must remain on the wave when m completes execution.

Node n is dominated by m, and there is no node on any path from m to n (including m but not necessarily n) that can happen together with any node that can rendezvous with a, b, or c. Thus, at least one member of $\{a, b, c\}$ must be on the wave when n starts to execute. If n can't happen together with any node that can rendezvous with a, b, c, or their sisters (except possibly n itself or one of n's sisters), then n pins a, b, and c to the wave. Therefore, we can add the same set of nodes to $CHT(n)$ which were added to $CHT(m)$ as a result of pinning analysis of m.

6.4 Strict intervals.

We define a *strict interval* as a single entry, single exit region in control flow with additional properties which limit control flow within the region. Strict intervals are the structural basis of critical sections and regions of mutual exclusion in several synchronization paradigms, including the Ada rendezvous. Thus, detecting strict intervals is an important part of deriving CHT from a program.

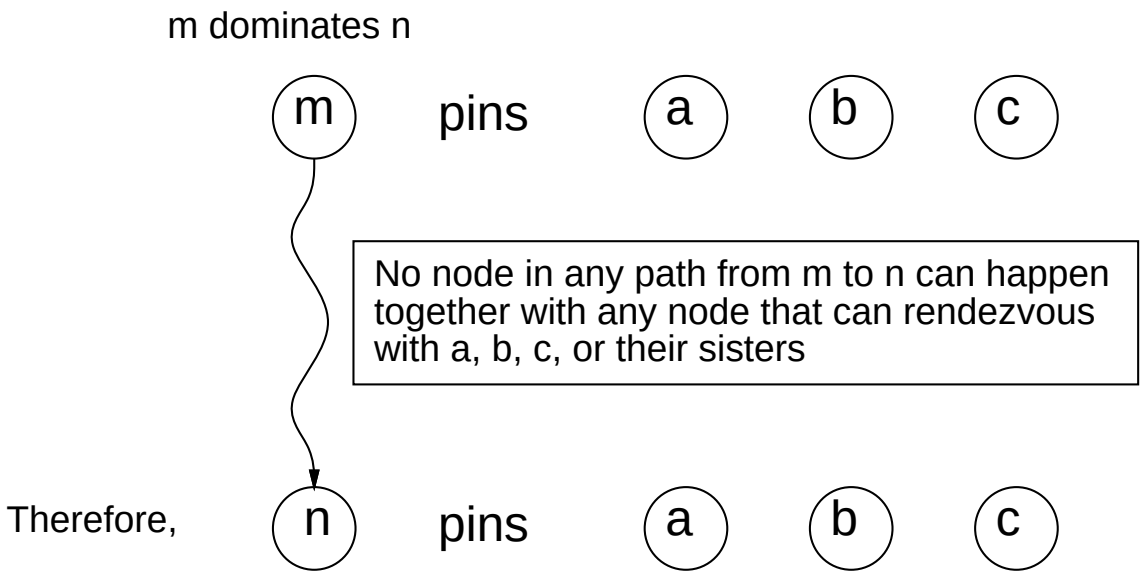

Figure 17: Propagation of pinning information to the successors of a node m

Suppose that G is a directed graph, and $\mathcal{P}$ is a subset of some partition of the nodes in G (i.e., $\mathcal{P} = \{P_1, ...P_k\}$ such that each $P_i \subseteq N$ and $\forall i \forall j \neq i : P_i \cap P_j = \emptyset$). Define $StrictInt(G, \mathcal{P})$ to be the set of all strict intervals in G relative to the partition $\mathcal{P}$. A strict interval is denoted by the pair (n_n, n_x), where $n_n \in P_n$, $n_x \in P_x \neq P_n$, $P_x, P_n \in \mathcal{P}$. The strict interval represented by (n_n, n_x) is the set of nodes which can be reached from n_n without passing through n_x (i.e., the union of all $n_i, 1 \leq i \leq k$, such that a path $(n_n, n_1, ..., n_k = n_x)$ exists in G). The following conditions are also required of strict intervals:

- Local conditions on each strict interval:
  - n_n and n_x form a single-entry, single-exit region in G, with n_n as the entry node and n_x as the exit node.
  - $\{n_n\}$ cuts[4] $(\{n_x\}, CPreds(n_x))$ in G. I.e., there are no cycles including n_x that omit n_n.
  - $\{n_x\}$ cuts $(\{n_n\}, CPreds(n_n))$ in G.
  - $\{n_x\}$ cuts $(\{n_n\}, (P_n \cup P_x) - \{n_n, n_x\})$ in G. I.e., no node in $P_n \cup P_x$ lies within the strict interval, except for n_x.
- Global conditions on all strict intervals:

[4] M cuts (N, P) iff, for all $n \in N - M$ and $p \in P - M$, any directed path from n to p contains some $m \in M$.

- $\forall(n_n \in P_n)\exists(n_x \in P_x) : (n_n, n_x) \in \mathit{StrictInt}(G, \mathcal{P})$. Every node in P_n is the entry of a strict interval that exits at a node in P_x.
- $\forall(n_x \in P_x)\exists(n_n \in P_n) : (n_n, n_x) \in \mathit{StrictInt}(G, \mathcal{P})$. Every node in P_x is the exit of a strict interval that enters at a node in P_n.

Suppose that the nodes within P_n and P_x form the entry and exit nodes, respectively, of a set S of strict intervals. Then we have the following useful properties:

- Strict intervals in S can only be entered through a node in P_n.
- When any node in P_n completes execution, control must enter a strict interval in S.
- No node in P_n can be within a strict interval in S.
- All nodes in P_x lie inside of a strict interval in S.
- Once control has entered a strict interval in S, it can only leave the strict interval by executing a node in P_x.
- Once a node in P_x has completed inside a strict interval in S, control must leave the strict interval.
- Strict intervals in S do not overlap.

6.4.1 Algorithm to find strict intervals.

Algorithm 4 produces a set of strict intervals, given a graph and a partition of its nodes.

Algorithm 4 *Identify strict intervals.*

Input:

- A directed graph $G = (N, E)$.
- A partition $\mathcal{P}$ of the nodes of N.

Output: The set $\mathit{StrictInt}(G, \mathcal{P})$.

Procedure:

1. Construct the control dominator and postdominator trees for G.

2. Identify pairs of nodes $(n_n \in P_n, n_x \in P_x), P_n, P_x \in \mathcal{P}$ such that:

   - $P_n \neq P_x$;
   - $n_n \in DOM(n_x)$;
   - $\forall n' \overset{\text{such that}}{} (n' \in DOM(n_x) \cap (P_n \cup P_x)) \wedge (n' \neq n_n) : \{n_n\}$ cuts $(\{n'\}, \{n_x\})$;
   - $n_x \in POSTDOM(n_n)$;
   - $\forall n' \overset{\text{such that}}{} (n' \in POSTDOM(n_n) \cap (P_n \cup P_x)) \wedge (n' \neq n_x) : \{n_x\}$ cuts $(\{n_n\}, \{n'\})$.

   Place these pairs (n_n, n_x) in S. S is the set of *candidate strict intervals.*

3. If there is any node $n_n \in P_n$ such that there is no pair $(n_n, n_x \in P_x) \in S$, then remove all pairs (n'_n, n'_x) from S such that $n'_n \in P_n, n'_x \in P_x$.

4. If there is any node $n_x \in P_x$ such that there is no pair $(n_n \in P_n, n_x) \in S$, then remove all pairs from S (n'_n, n'_x) such that $n'_n \in P_n, n'_x \in P_x$.

5. Return S.

Lemma 34 (Correctness of the strict interval algorithm.) *The set StrictInt$(G, \mathcal{P})$ returned by Algorithm 4 is the set of those strict intervals that satisfies the definition given in Section 6.4.*

Proof: Step 2 places in set S exactly those node pairs which satisfy the local conditions on strict intervals for graph G and partition $\mathcal{P}$. Steps 3 and 4 eliminate from S those candidate strict intervals that do not satisfy the global conditions for G and $\mathcal{P}$. Neither of the two elimination steps cause any valid strict intervals to be eliminated, nor does any step cause any candidate strict intervals which were valid under the previous steps to become invalid under them. Therefore, upon completion of the algorithm, S contains exactly those strict intervals that satisfy the definition of Section 6.4. □

Lemma 35 *Using Algorithm 4, the time to find all strict intervals for a given graph and node partition is $\mathcal{O}(|N|^3)$.*

Proof: The dominator and postdominator trees of a flow graph can be found in time $\mathcal{O}(|E|\log(|N|))$ by the simpler algorithm of [LT79]. From this, the initial pairs $(n_n, n_x) \in S$

may be constructed in $\mathcal{O}(|N|^2)$, by traversal of these trees.[5] We ignore pairs of nodes which are in the same partition.

At this point, we can eliminate unsatisfactory candidate strict intervals (n_n, n_x) by performing depth-first search along forward control edges from n_n and along backward control edges from n_x. We eliminate the interval from S if there is a forward path from n_n to any node $n' \neq n_x \in P_n \cup P_x$ that does not include n_x. Likewise, we eliminate the interval if there is a backward path from n_x to any node $n' \neq n_n \in P_n \cup P_x$. Doing these searches takes $\mathcal{O}(|N|)$ time per pair in S. There may be up to $\mathcal{O}(|N|^2)$ elements in S (or, more accurately, $\mathcal{O}(|N||\mathcal{P}|)$, since each node in P_n may be in an interval with only one node in P_x, or vice versa). Step 2 thus takes a total of $\mathcal{O}(|N|^3)$ time.

Finally, for each remaining pair $(n \in P_n, x \in P_x)$, we examine P_n to see if it contains any nodes n' which is not a member of a pair $(n', x' \in P_x)$. We similarly examine P_x. This examination may be done in $\mathcal{O}(|N|)$ time per pair, using bit vectors, for a total of $\mathcal{O}(|N|^3)$ (again more accurately $\mathcal{O}(|N||\mathcal{P}|^2)$). The total execution time of Algorithm 4 is thus $\mathcal{O}(|N|^3)$. □

6.5 Critical section analysis.

A *critical section*, for our purposes, is a set of regions of different tasks, such that the nodes of at most one region can execute simultaneously. Critical sections may be used, for example, to enforce mutual exclusion of access to a shared variable or other resource.

There are many ways of implementing this type of mutual exclusion, some of which involve the use of memory state information, and some of which do not. We examine a fairly popular instance of the latter in detail in this section. We develop a method of identifying a particular form of critical section structures in a sync hypergraph, and of using these to obtain *CHT* information. We assume that critical section calls may be nested, that critical section bodies may call other critical sections, etc. This model overlaps with, but does not subsume and is not subsumed by, the models handled by the methodology presented in Section 6.3.

[5] A somewhat more clever way to do this is to construct a single graph, with two types of edges, corresponding to edges in the dominator and postdominator trees of signaling and accept-out nodes only. A candidate strict interval would then be found by finding a cycle consisting of a dominator and a postdominator edge. However, doing this does not decrease the order-of-magnitude complexity of Step 2.

6.5.1 An example of a critical section.

We present here a simple example of a critical section. A more general form of critical sections is presented in Section 6.5.2.

Consider the sticky node n of Figure 18, which is reachable only after a rendezvous between its predecessor m and the predecessor of p has been executed. Suppose further that there are successors s_i of p, such that, along all control paths from p to s_i, all nodes are known to be in $CHT(x)$ for any x that can rendezvous with n or any of its sisters. In this case, n *must happen together* with each s_i. The contrapositive is of interest, because any node that is in $CHT(n)$ is also in $CHT(s_i)$. This is generalized and stated more formally below:

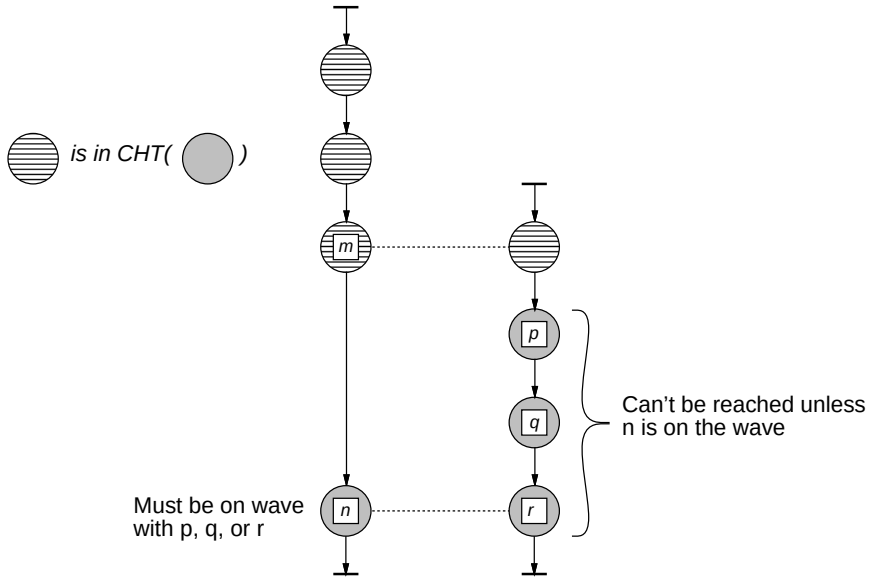

Figure 18: A critical section.

Lemma 36 *Assume that n is a sticky node with a partner p, and let q be a node reachable in control flow from p. If the following conditions hold:*

- *n postdominates $Completors(n)$.*
- *For all $x \neq n$, $p \notin Partners(x)$.*
- *On all control paths $s_1 = p, ..., s_k = q$:*
  - *No node n' that can rendezvous with n or with any sister of n can happen together with any s_i.*
  - *Each s_i is dominated by $Partners(n)$.*

Then n must happen together with all members of all paths from p to s.

Proof: Since n is a sticky node, it must remain on the wave until some node that can rendezvous with n reaches the wave. Since $p \notin Partners(x)$ for $x \neq n$, p can only be on the wave as a result of a rendezvous between some member of $Completors(p)$ and some member of $Completors(n)$. Since n postdominates $Completors(n)$, p can only be placed on the wave concurrently with n.

Assume that p is placed on the wave, and that the members of some chain of successors $s_1 = p, ..., s_k = q$ of p such that each $s_i \in CHT(x)$ for any x that can rendezvous with n or any of n's sisters. Then, whenever the entire chain executes, n must be on the wave at all times during the execution of the chain. Since $Partners(n)$ dominates each s_i, each s_i can only execute as a result of executing the entire chain. Therefore, n must remain on the wave at least until q completes execution. □

Corollary 3 *If there is a sticky node r such that:*

- *r is a child of some node q as above,*
- *r is dominated by members of $Partners(n)$,*
- *r can only be reached from members of $Partners(n)$ along paths of nodes that cannot rendezvous with n or its sisters,*
- *r can only rendezvous with n or its sisters, or with nodes in $CHT(n)$,*

then r must happen together with n.

Proof: When r reaches the wave, n must be on the wave. r can only leave the wave by rendezvousing with n or one of its sisters. □

6.5.2 Critical section structures.

A *critical section structure* is a set of specialized single entry-single exit regions which generalize the above simple case, as in Figure 19. (The term "critical section structure" comes from the fact that such structures are often used to implement critical sections in the rendezvous paradigm.)

Critical section structures are denoted by the pair (C, B) of sets of node pairs. C is the set of *call bodies* of the critical section, i.e., pairs (c, r), corresponding to the calls to, and returns from, a critical section. B is the set of of *critical section bodies* of the critical section, i.e., node pairs (n, x), corresponding to the entry and exit nodes of each critical section body. Because of its structural characteristics, the critical section structure has the properties that:

- Only one of its critical section bodies may execute at any time;
- Only one of its call bodies may execute at any time;
- Any call body of the structure must execute simultaneously with a critical section body of the same structure;
- Any critical section body of the structure must execute simultaneously with a call body of the same structure.

We will initially assume that $|B| = 1$; later, in Section 6.5.3, we expand this to $|B| \geq 1$. Also note that a node may be contained within many critical section structures, and that the structures are not required to overlap in any given way, e.g., by nesting.

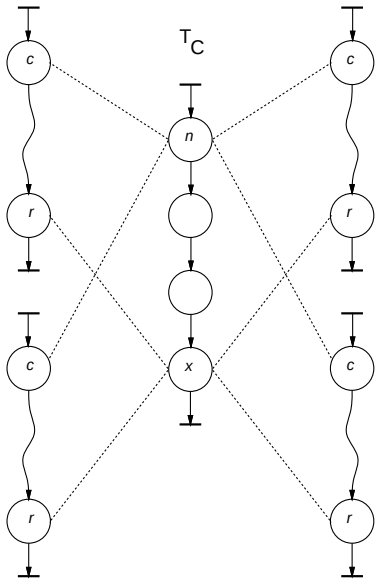

Figure 19: A critical section structure.

Denote as $SameType(n)$ the set of nodes of the same signal type and node type as n, and:

$$N_C = \bigcup_{(c,r)\in C}\{c\};$$
$$N_R = \bigcup_{(c,r)\in C}\{r\};$$
$$N_N = \bigcup_{(n,x)\in B}\{n\};$$
$$N_X = \bigcup_{(n,x)\in B}\{x\}.$$

The pair of nodes $(n, x \neq n)$ delineates a critical section structure $(C, B = \{(n, x)\})$ if:

- n and x are either signaling or accept-out nodes.
- $Scope(n, y_n) = Scope(x, y_x) = Scope(n, y_n) = Scope(x, y_x) = 0$, where y_n and y_x are the signal types of n and x respectively.
- n dominates x.
- x postdominates n.
- n and x are of different signal types.
- n and x are sticky.
- For all $(c, r) \in C$, c and r are either signaling or accept-out nodes.
- For all $(c, r) \in C$, $Scope(c, y_n) = Scope(c, y_x) = Scope(r, y_n) = Scope(r, y_x) = 0$.
- For all $(c, r) \in C$, c and r are sticky.
- For all $(c, r) \in C$, c can rendezvous only with n.
- For all $(c, r) \in C$, r can rendezvous only with x.
- If n can rendezvous with some c, then $\exists (c, r) \in C$.
- If x can rendezvous with some r, then $\exists (c, r) \in C$.
- Any cycle in control flow that includes n must also include x, and vice versa.

  Equivalently,

  $$\{x\} \text{ cuts } (\{n\}, CPreds(n)) \wedge \{n\} \text{ cuts } (\{x\}, CPreds(x)).$$

- For all $(c, r) \in C$, any cycle in control flow that includes c must also include r, and vice versa. Equivalently,

  $$\forall (c, r) \in C : \{c\} \text{ cuts } (\{r\}, CPreds(r)) \wedge \{r\} \text{ cuts } (\{c\}, CPreds(c)).$$

- For every pair $(c, r) \in C$, c dominates, and is postdominated by, r. Equivalently,

  $$\forall (c, r) \in C : \{c\} \text{ cuts } (\{b\}, \{r\}) \wedge \{r\} \text{ cuts } (\{c\}, \{e\}).$$

- All control paths from any $c_i \in N_C$ to any $c_j \in N_C$ must contain some $r \in N_R$. This prevents any task from issuing a call to a critical section while control is within a call body for the same critical section. It likewise prevents nesting or overlapping call bodies to the same critical section.

  Equivalently, N_R cuts $(\{c_i\}, N_C)$.

- All control paths from any $r_i \in N_R$ to any $r_j \in N_R$ must contain some $c \in N_C$. This prevents any task from issuing a return from a critical section while control is outside any call body for the same critical section. It also prohibits nesting or overlapping call bodies to the same critical section.

  Equivalently, N_C cuts $(\{r_i\}, N_R)$.

For $(c, r) \in C$, define the *call body* $CallB(c, r)$ as the interval (c, r) minus c. Also, define the *critical section body* $CSB(n, x)$ as the interval (n, x) minus n. The set

$$\left[\bigcup_{(c,r)\in C} \{c, r\}\right] \cup \left[\bigcup_{(n,x)\in B} \{n, x\}\right]$$

is called the set of *bracket nodes* for the critical section, since these are the nodes that delimit the call bodies and critical section bodies in control flow.

Lemma 37 *If m dominates n, then m dominates each q in interval (m, n).*

Proof: Suppose some node q is reachable from a path from m which does not include n, but m does not dominate q. Then there is a path from some predecessor of m to n (through q) that does not include m. Therefore, m does not dominate n, contrary to the assumption. □

Lemma 38 *If n postdominates m, then n postdominates all nodes q in interval (m, n).*

Proof: Suppose some node q is reachable from a path from m which does not include n, but n does not postdominate q. Then there is a path from m to some successor of n (through q) that does not include n. Therefore, n does not postdominate m, contrary to the assumption. □

Lemma 39 *All nodes in any critical section call body $CallB(c, r)$ are dominated by c and postdominated by r.*

Proof: Direct from the definition of the call body, and from Lemmas 37 and 38. □

Lemma 40 *All nodes in any critical section body $CSB(n, x)$ are dominated by n and postdominated by x.*

Proof: Identical to Lemma 39. □

Theorem 2 (Critical section structures.) *Each call body $CallB(c,r)$ must happen together with critical section body $CSB(n,x)$, and $CSB(n,x)$ must happen together with exactly one call body at any time.*

Proof: We will consider these contradictory possibilities in order:

1. $CallB(c,r)$ can start at some time when $CSB(n,x)$ does not,
2. $CSB(n,x)$ can start at some time when no call body does,
3. Two (or more) call bodies can execute simultaneously,
4. $CallB(c,r)$ may terminate while $CSB(n,x)$ does not,
5. $CSB(n,x)$ may terminate when no call body does.

Possibility 1. Consider first the possibility that when $CallB(c,r)$ is entered, $CSB(n,x)$ is not. $CallB(c,r)$ cannot be entered except through c, since c dominates all nodes in $CallB(c,r)$. Node c must rendezvous with n for control to enter $CallB(c,r)$. All successors of n are members of $CSB(n,x)$, and all successors of c are members of $CallB(c,r)$. Therefore, if control enters $CallB(c,r)$, it will also enter $CSB(n,x)$ at the same time. So the first possibility cannot hold.

Possibility 2. Similarly, $CSB(n,x)$ cannot be entered except through n, which must rendezvous with some $c \in N_C$ to enter $CSB(n,r)$. Each c dominates a call body $CallB(c,r)$. So if control enters $CSB(n,x)$, it will also enter some call body at the same time. So the second possibility cannot hold.

Possibility 3. Now consider the third possibility, i.e., that two call bodies $CallB(c,r)$ and $CallB(c',r')$, execute simultaneously, and are in fact the first pair of call bodies that do. Each of the call nodes for these bodies must rendezvous with n, in order for the bodies to start execution. Since n can rendezvous only with one node at any given state transition, no two call bodies can start executing simultaneously. Therefore, $CallB(c,r)$ and $CallB(c',r')$ must start at different times. Suppose, without loss of generality, that $CallB(c,r)$ starts before $CallB(c',r')$ does. $CallB(c',r')$ cannot start until c' does rendezvous with n. But n cannot be reached from within $CSB(n,x)$, and x postdominates $CSB(n,x)$ and can rendezvous only with members of N_R. Therefore, while $CallB(c,r)$ continues to execute, some $r'' \in N_R$ must rendezvous with x to allow n to be reached in control flow, before any other $CallB(c',r')$ can start.

Suppose that $r'' = r$. Since r postdominates all members of $CallB(c,r)$, and there are no cycles inside $CallB(c,r)$ that contain r, this means that $CallB(c,r)$ stops, contrary to the hypothesis that it continues executing until $CallB(c',r')$ starts. Therefore, $r'' \neq r$. But r'' is a member of $CallB(c'',r'')$. Thus, at least one member of $CallB(c'',r'')$ executes simultaneously with $CallB(c,r)$, and must have started executing before $CallB(c',r')$ did. This violates our hypothesis that $CallB(c',r')$ was the first call body to executes simultaneously with $CallB(c,r)$. Thus, at most one call body may execute simultaneously with $CSB(n,x)$, and the third possibility cannot hold.

Possibility 4. Now, suppose that $CallB(c,r)$ exits before $CSB(n,x)$ does. $CallB(c,r)$ is postdominated by node r, so $CallB(c,r)$ must be exited following completion of r, which must rendezvous with x. (Note that r must be sticky, or it could not postdominate the call body, by construction of the sync hypergraph.) x postdominates $CSB(n,x)$, and there is no cycle within $CSB(n,x)$ that includes x. So after the rendezvous between r and x, control is no longer in $CSB(n,x)$. So $CallB(c,r)$ cannot exit before $CSB(n,x)$ does, and the fourth possibility cannot hold.

Possibility 5. Finally, consider the possibility that $CSB(n,x)$ exits before $CallB(c,r)$ does. $CSB(n,x)$ cannot be exited except through node x, since x postdominates all nodes in $CSB(n,x)$. x must rendezvous with some node in N_R to allow control to leave $CSB(n,x)$. Since no two call bodies can execute simultaneously, x can only rendezvous with r when control leaves $CSB(n,x)$; thus, control will leave $CallB(c,r)$ simultaneously. Therefore, the fifth possibility cannot hold. □

Corollary 4 *If $CallB_i \neq CallB_j$ are call bodies for the same critical section, then no node in $CallB_i$ can happen together with any node in $CallB_j$.*

Proof: Direct from Theorem 2. □

Corollary 5 *If $CallB_i$ is a call body and CSB_j is its critical section body, then no node in $Task(CSB_j) - CSB_j$ can happen together with any node in $CallB_i$.*

Proof: Direct from Lemma 2. □

6.5.3 Relaxing the structural requirements on critical sections.

In this section, we show how some of the requirements listed in the definition of critical section structure in Section 6.5 can be relaxed.

Eliminating the stickiness check. The call and return nodes of a critical section, and the entry and exit nodes of a critical section body, were previously required to be sticky. This was done to

enforce the requirement that call bodies be entered in execution only through the rendezvous with the call nodes, and left only through the rendezvous with the return nodes. Similar requirements restrict the entry and exit nodes of a critical section body. However, it turns out that the reason for the stickiness requirement is actually enforced by the control dominance characteristics of the sync hypergraph.

Lemma 41 *If x dominates y in control flow in the sync hypergraph and x is slippery, then y can only be entered following a rendezvous between x and some other node.*

Proof: By construction of the sync hypergraph, if x is slippery, then there are then control edges from all control predecessors of x to the front nodes of the `else` clause or unguarded `delay` alternative of the clause C containing x, and/or to the control successors of x if either of these clauses are empty. In this case, x does not dominate any of the successors of C, because there is a path from the begin node b which reaches them without going through x. x does not dominate any nodes in the `else` clause or unguarded `delay` alternative, for the same reason.

The only nodes which x may dominate are those in the sequence of statements following x, or those embedded within x if x is an accept-in node of an `accept-do` construct. These can only be executed following the initiation of a rendezvous with x. □

Lemma 42 *If y is slippery, then y cannot postdominate any node in control flow in the sync hypergraph.*

Proof: By construction, if y is slippery, there are edges from the control predecessors of y to the front nodes of the `else` clause or unguarded `delay` alternative of the clause C containing y, and/or to the control successors of y if either of these clauses are empty. Thus, there are paths from the predecessors of y to the successors of y which do not include y, and y cannot postdominate its predecessors. Therefore, y cannot postdominate any node in the sync hypergraph. □

So, if x and y delineate a single entry, single exit region, then the nodes in the region (other than x itself) can only be reached through a rendezvous with x, and the region can only be left through a rendezvous with y. The stickiness checks for the entry and exit nodes are not necessary.

Multiple critical section bodies. It is possible, though uncommon, for more than one critical section body to receive the same calls. Fortuitously, such cases are easy to identify in Ada.

Lemma 43 *If two different critical section bodies, $CSB(n, x)$ and $CSB(n', x')$, are such that both n and n' can rendezvous with $c_1, c_2 \in N_C$, and both x and x' can rendezvous with $r_1, r_2 \in N_R$, then either n and n' (and therefore x and x') are in the same task, or c_1 and c_2 (and therefore r_1 and*

r_2) are in the same task.

Proof: Only accept nodes can rendezvous with nodes from more than one task, because entry calls are restricted (by the semantics of Ada) to call only the entries of one task. This implies that either the two critical section bodies or the two call bodies are in the same task. □

Lemma 44 *No two critical section bodies $CSB(n,x)$ and $CSB(n',x')$ in which n and n' (and therefore x and x') have the same signal type can happen together.*

Proof: Two such bodies must be contained in the same task, and are therefore disjoint. □

Theorem 3 *If a critical section structure contains multiple critical section bodies, then each call body must happen together with one of the critical section bodies, and each critical section body must happen together with exactly one call body at any time.*

Proof: Very similar to Theorem 2. □

Corollary 6 *If $CallB_i$ is a call body and $CSB_1, CSB_2, ...CSB_k$ are its critical section bodies, then, for all $j \in [1...k]$, no node in $Task(CSB_j) - CSB_1 - CSB_2 ... - CSB_k$ can happen together with any node in $CallB_i$.*

Proof: Direct from Theorem 3. □

Due to these results, we can treat critical sections with multiple bodies the same way we would treat critical sections with a single body. We can therefore compute *CHT* for critical section structures with multiple critical section bodies, as long as we are careful to insure that all the critical section bodies are all contained within a single task.

6.5.4 Scope depth and critical sections.

In Section 6.5.2, the bracket nodes of a critical section structure were forbidden to have a scope depth > 0 with respect to the signal type of any of the bracket nodes. If this restriction is relaxed, then it becomes possible for multiple critical section bodies to execute at once.

Lemma 45 *Suppose $(n,x) \in B$ for some candidate critical section structure (C,B). If $Scope(x, y_x) > 0$, then either multiple call bodies may execute simultaneously, or (C,B) does not correspond to the definition of a critical section structure.*

Proof: Under these assumptions, some accept-do construct of type y_x encloses x. Either the same construct encloses n, or it does not.

If the former is the case, then let x' be the accept-out node of the structure enclosing n and x. In order for n to execute, some signaling node must be ENGAGED to x' when n executes. This signaling node must be some r such that $(c,r) \in C$. If n makes a rendezvous with c' such that

$(c', r') \in C$, then some member of $CallB(c', r')$ executes simultaneously with r. Thus, nodes in two call bodies may execute simultaneously.

If the latter is the case, then there must be some accept-in node of the same signal type as x, which lies on a path from n to x that does not include x. In this case, (C, B) is not properly a critical section structure. □

Other forms of nesting have even worse implications:

Lemma 46 *Suppose $(n, x) \in B$ for some candidate critical section structure (C, B). If n is an accept-in node of type y_n and $Scope(n, y_n) > 0$, then either (C, B) must deadlock with one call body active, or (C, B) does not properly define a critical section structure (ignoring rules of scope depth).*

Proof: Let n' be the accept-in node of the `accept-do` construct of type y_n most immediately enclosing n. There is at least one path from n' to n, by construction of the sync hypergraph. If there is a path from n' to n which includes no nodes in N_X, then the candidate critical section structure (C, B) does not meet the definition of a critical section structure.

On the other hand, suppose any path from n' to n includes some node in N_X. Then n' is postdominated by some such node x', and all paths from n' to n include x'. Whenever x' executes, the node which rendezvoused with n' is still ENGAGED, and no node from that task (or any other) is available to rendezvous with x'. Thus, the structure deadlocks with one call body active. □

We could conceivably relax the rules about critical section structures to allow nesting of bracket nodes inside `accept-do` constructs of other bracket nodes. However, since this construct never appeared in the test data, the value of doing so is probably low.

6.5.5 Identifying critical sections.

The following algorithm can be used to identify critical sections. It relies on doing inexpensive tests first.

Algorithm 5 *Identify critical section structures.*

Input: The *CHT* form sync hypergraph $G = (N_{CHT}, E_{C_{CHT}} E_{S_{CHT}})$.

Output: A set of critical section structures, CRITSTRUCTS.

Procedure:

1. Create a partition $\mathcal{P}$ such that each $P_i \in \mathcal{P}$ contains all nodes of a particular signal type and polarity in G.

2. Create the control flow graph $G' = (N_{CHT}, E_{C_{CHT}})$.

3. Set CANDBODIES to $StrictInt(G', \mathcal{P})$.

4. For each pair $(a, b) \in$ CANDBODIES found in step 3, form a *candidate critical section structure* (M, N) such that:

   - M is the set of pairs $(a', b') \in$ CANDBODIES, such that:
     - $SigType(a) = SigType(a')$;
     - $NodeType(a) = NodeType(a')$;
     - $SigType(b) = SigType(b')$;
     - $NodeType(b) = NodeType(b')$.
   - N is the set of pairs $(a'', b'') \in$ CANDBODIES, such that:
     - $SigType(a) = SigType(a'')$;
     - $NodeType(a) \neq NodeType(a'')$;
     - $SigType(b) = SigType(b'')$;
     - $NodeType(b) \neq NodeType(b'')$.

   In short, for all pairs $(a', b') \in M$ and $(a'', b'') \in N$, a' can rendezvous with a'' and b' can rendezvous with b''.

   Place all the candidate critical section structures into the set CANDSTRUCTS.

5. Eliminate from CANDSTRUCTS all pairs (M, N) such that $|M| = 0$ or $|N| = 0$.

6. Eliminate from CANDSTRUCTS all pairs (M, N) such that $M \cup N$ contains a pair (m, n) such that:

   - $Scope(m, y_m) > 0$;
   - $Scope(m, y_n) > 0$;
   - $Scope(n, y_m) > 0$; or,
   - $Scope(n, y_n) > 0$.

7. For each pair (M, N) in CANDSTRUCTS, let B be whichever of M or N has all nodes of its pairs as members of the same task, and let C be the other set. (If both sets meet this qualification, choose arbitrarily.) Place the resulting pair (B, C) in CRITSTRUCTS.

Lemma 47 (Correctness of critical section identification.) *Algorithm 5 correctly identifies the critical section structures of a program.*

Proof: From the discussion, following step 6, CANDSTRUCTS contains only pairs (M, N) such that either (M, N) or (N, M) meets the definition of a critical section structure (as defined in Section 6.5.2 and elaborated in Sections 6.5.3 and 6.5.4). Step 7 identifies which of these two alternatives enforces the requirement that the critical section bodies be in each others' *CHT* sets. All steps maintain validity of the results of the previous steps. Thus, at the termination of the algorithm, CRITSTRUCTS is the set of critical section structures of the program. □

Lemma 48 *Algorithm 5 uses $\mathcal{O}(|N_{CHT}|^3)$ time in the worst case.*

Proof: The partitioning of step 1 can be done in $\mathcal{O}(|N_{CHT}|)$ time; creating G' from G can be done in constant time. Using Algorithm 4, the CANDBODIES set of step 3 can be created in $\mathcal{O}(|N_{CHT}|^3)$ time.

Step 4 can be accomplished by sorting the node pairs in CANDBODIES, keyed on the signal and node types of a and b. The sort will take $\mathcal{O}(|N_{CHT}|^2 \log(|N_{CHT}|))$ time. The result of the sorting will be a list with at most $4|Sigs|(|Sigs| - 1)$ members. For each member M of this list, we find its corresponding N by looking up the pair with the same signal types and opposite polarity. The lookup process thus takes $\mathcal{O}(|Sigs|^2 \log(|Sigs|))$ time. So the total time for Step 4 is $\mathcal{O}(|N_{CHT}|^2 \log(|N_{CHT}|))$ time, since the number of nodes is at least the number of signals used by all nodes in the program. During the construction of the CANDSTRUCTS set, we can also eliminate the candidate structures with a zero-size set (Step 5).

Therefore, the total time to construct CRITSTRUCTS is $\mathcal{O}(|N_{CHT}|^3)$ in the worst case. □

6.5.6 Expanding *CHT* information using critical section structures.

Algorithm 6 *Expanding CHT information using critical section structures.*

Input:

- The CRITSTRUCTS set, as generated by Algorithm 5.
- *CHT*.

Output:

- $CHT' \supseteq CHT$.

Procedure:

1. $CHT' \Leftarrow CHT$.
2. For each critical section structure (B, C) in CRITSTRUCTS do:
   (a) $CSBNodes \Leftarrow \bigcup_{CSB(n,x) \in B} CSB(n, x)$.
   (b) $CallNodes \Leftarrow \bigcup_{CallB(c,r) \in C} CallB(c, r)$.
   (c) $CHT' \Leftarrow CHT' \cup ((CallNodes^2 \cup CSBNodes^2) - I)$.
   Nodes in call bodies or critical section bodies of the same structure can't happen together.
   (d) $MustHT \Leftarrow ((CSBNodes \times CallNodes) \cup (CallNodes \times CSBNodes))$.
   This finds sets of nodes which must happen together because they are in a call or critical section body of the same critical section structure.
   (e) $CHT' \Leftarrow Widen(CHT', MustHT, N)$.

Lemma 49 (Correctness of Algorithm 6.) *If $CHT \subseteq CHT_{perf}$ and the other input parameters of Algorithm 6 are correctly specified, then $CHT \subseteq CHT' \subseteq CHT_{perf}$.*

Proof: From Theorems 1 and 2. □

Lemma 50 *Algorithm 6 requires $\mathcal{O}(|N_{CHT}|^4)$ setup time and $\mathcal{O}(|N_{CHT}|^5)$ time per iterative refinement, in the worst case.*

Proof: Computing *CSBNodes* and *CallNodes* for each critical section structure can be done while the critical section structure is being constructed. Computing *MustHT* for each critical section structure can be done in $\mathcal{O}(|N_{CHT}|^2)$ time, and need only be done once in the life of the program. Step 2c likewise requires $\mathcal{O}(|N_{CHT}|^2)$ time, and need only be done once in the life of the program.

The call to *Widen* of step 2e requires $\mathcal{O}(|N_{CHT}|^3)$ time per critical section structure; since it is the step that uses widening to refine CHT, it must be done every time the refinement is done.

There may be up to $\mathcal{O}(|N_{CHT}|^2)$ critical sections, each containing $\mathcal{O}(|N_{CHT}|)$ nodes. Such program structures are realizable. Thus, Algorithm 6 requires $\mathcal{O}(|N_{CHT}|^4)$ setup time and $\mathcal{O}(|N_{CHT}|^5)$ time per iterative refinement, in the worst case. □

Note that the worst case time assumes a pathological case which is not likely to occur. In practice, we expect only a small number of critical section structures. However, due to the large asymptotic time bound on widening, we may wish to restrict the number of times iterative refinement is done on critical sections to a constant number, and pay some penalty in loss of *CHT* information. However, we will assume here that we wish to do full iterative refinement of *CHT* using critical sections, and will investigate the effects of limiting iterative refinement in our experimental work.

Lemma 51 (Time bounds for the critical section algorithm.) *The time to find critical section structures and use them to expand CHT information is* $\mathcal{O}(|N_{CHT}|^5)$ *in the worst case.*

Proof: From Lemmas 48 and 50. □

However, we may do better than this by using an incremental version of Algorithm 2 (the widening algorithm):

Lemma 52 (Incremental time bounds for the critical section algorithm.) *The time to incrementally expand CHT information is* $\mathcal{O}(|N_{CHT}|^3)$ *per bit change in CHT in the worst case.*

Proof: Each call to the incremental version of *Widen* requires $\mathcal{O}(|N_{CHT}|)$ time. If there are $\mathcal{O}(|N_{CHT}|^2)$ critical section structures, a total of $\mathcal{O}(|N_{CHT}|^3)$ will be required by the incremental expansion. □

6.5.7 Deadlocks with critical section nodes as head nodes.

Example. Figure 20 shows a case in which two tasks each use nested calls to critical sections. The program here has two critical section bodies, (e, f) and (g, h). (e, f) is called by (b, c) and (k, l), while (g, h) is called by (a, d) and (j, m). The system cannot deadlock.

Table 6.5.7 shows the *CHT* relation computed between nodes of the program of Figure 20.

After unrolling the loops, we have a possible cycle (b, c, f, e', b), since b and f can happen together. (The fact that they can is a true *CHT* relation rather than an approximation, since the execution wave $[b, f, h, l]$ is feasible. The cycle happens to violate Constraint 4.) However, if we do pinning analysis, we find that f must always happen together with some node that can rendezvous with f. Therefore, f cannot be a head node, and (b, c, f, e', b) is not a valid deadlock cycle. For the same reason, (k, l, f, e', k) is not a valid deadlock cycle.

We notice a more serious problem with the cycle (a, b, c, d, h, g', a), which also violates Constraint 4. Here, a and h are the head nodes, and they also can happen together (in waves $[a, e, h, m]$ and $[a, f, h, l]$), and pinning analysis does not show that either a or h must have a rendezvous available.

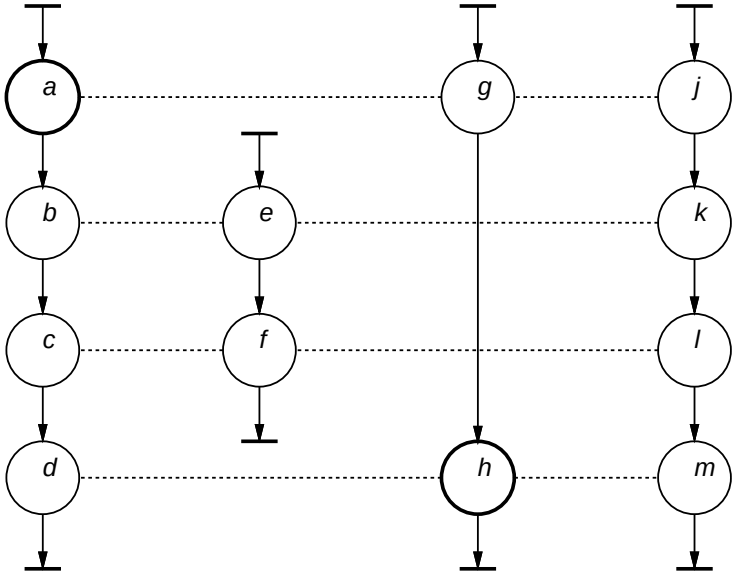

Figure 20: A program with nested calls to critical sections.

Without further information, therefore, deadlock cycle (a, b, c, d, h, g', a) must be considered valid.

6.5.8 Bracket node propagation and deadlock cycle detection.

The following theorems and lemmas allow us to eliminate (a, b, c, d, h, g', a) in Figure 20 as a spurious deadlock cycle, without constructing a critical section call graph.

Theorem 4 (Bracket nodes in a deadlock.) *In a deadlock, the nodes on the execution wave for those tasks which participate in a critical section structure cannot consist solely of the bracket nodes of the structure.*

Proof: Suppose the opposite, i.e., that there is a deadlock such that, for all tasks participating in a critical section structure, the nodes on the execution wave are all bracket nodes of the structure. Either one of the critical section bodies is executing or none are.

If no critical section body is executing, then, by hypothesis, the entry node n of one of the critical section bodies is on the execution wave. Since any call body must happen together with some critical section body, no call body can be executing. Therefore, the call node c of some call body must be on the execution wave. In that case, the wave cannot be deadlocked, because n and c can rendezvous.

Now suppose that some critical section body is executing. By hypothesis, its exit node x must be on the execution wave. The critical section body must happen together with some call body. Also by hypothesis, the call body's return node r must be on the wave. But x and r can rendezvous,

Node	*CHT* Set
a	b,c,d
b	a,c,d,f,g,k,l,m
c	a,b,d,e,g,k,l,m
d	a,b,c,f,g,k,l,m
e	c,f,l
f	b,d,e,g,k,m
g	b,c,d,f,h,k,l,m
h	g
j	k,l,m
k	b,c,d,f,g,j,l,m
l	b,c,d,e,g,j,k,m
m	b,c,d,f,g,j,k,l

Table 4: *CHT* Relation for the program of Figure 20.

so the wave cannot be deadlocked in this case, either. □

In connection with critical sections, we must be especially careful about the term "valid deadlock cycle," and the nature of the "waits-on" relationship. Consider the cycle (c, r, x, n', c) of Figure 21. The critical section exit node x could be said to wait on either r or r'. However, it more directly waits on r', since r' belongs to the call body which is executing concurrently with the critical section. Thus, if there is a valid deadlock cycle in this case, it must contain q as a head node, where q is a member of the call body containing the next $r \in R$ that can possibly execute. The cycle (c, r, x, n', c) is *irrelevant*, since it would exist in any program with two call bodies, regardless of whether the program deadlocked or not.

In Lemmas 53 through 56, we justify our assertion that deadlock cycles having only the bracket nodes of a single critical section structure as their head nodes are irrelevant.

Lemma 53 *If any valid deadlock cycle includes the exit node of a critical section body as a head node, there must also be an infinite wait anomaly which either includes a member of a call body for that critical section (other than the return node) as a head node of a deadlock or starved task, or in which a call body member (other than the return node) is transitively connected to a deadlock or starved task.*

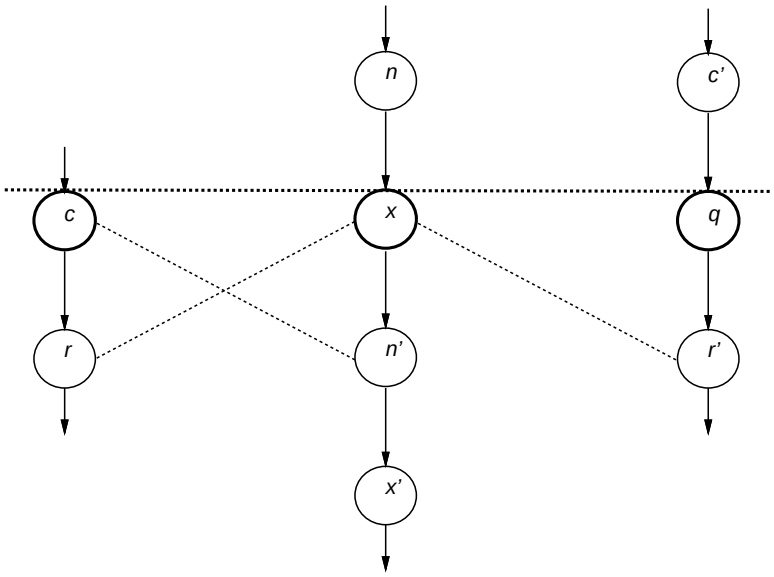

Figure 21: Spurious deadlock cycle which does not indicate a real "waits-on" relationship.

Proof: See Figure 22. The exit node x is a member of a valid deadlock cycle, by hypothesis. Therefore, it must be a member of an anomalous execution wave. x may only be present on the wave if a node q in a call body for x's critical section is also present. Furthermore, q cannot be the return node r (which could rendezvous with x). By Theorem 1 of [MR90], q must be a starved node, or a head node of a deadlock cycle, or transitively connected to a starved task or a deadlock. □

Lemma 54 *If any valid deadlock cycle includes the return node of a call body as a head node, then the cycle must also include a member of the critical section body (other than the exit node) as a head node.*

Proof: See Figure 23. The return node r is a member of a valid deadlock cycle, by hypothesis. Therefore, some member node q of a corresponding critical section body is executing along with r. $q \neq x$, for otherwise q and r could rendezvous. r can only rendezvous with nodes in $Task(q)$, by the definition of a critical section structure. Therefore, the deadlock cycle must include q as well as r as head nodes. □

Lemma 55 *If any valid deadlock cycle includes the entry node of a critical section body as a head node, then the deadlock cycle cannot include the call or return node of any call body for that critical section.*

Proof: If the deadlock cycle included the entry node and any call node, then the two could rendezvous. The deadlock cycle cannot include the entry node and any return node, since these

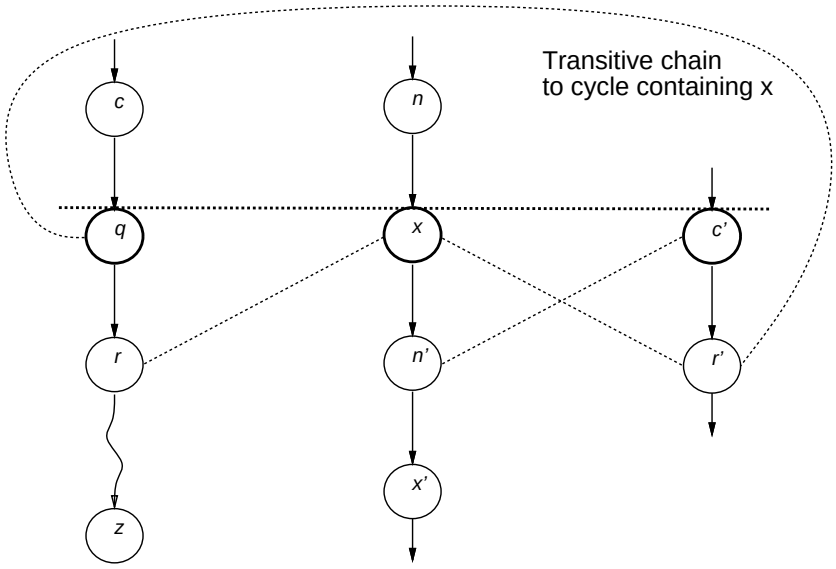

Figure 22: Deadlock cycle with a critical section exit node as head node.

can't happen together. □

Lemma 56 *If any valid deadlock cycle includes the call node of a call body as a head node, then the deadlock cycle cannot include the entry node of any critical section body for that critical section.*

Proof: If the deadlock cycle included the call node and any entry node, then the two could rendezvous. □

Theorem 5 (Bracket nodes and deadlock cycles.) *Any deadlock cycle which contains any bracket nodes of a critical section as head nodes must either contain some node other than the bracket node as a head node, or must be transitively coupled to another deadlock cycle or starved task.*

Proof: By Lemma 53, if the deadlock cycle contains an exit node x, it must contain, or be coupled to, a non-bracket head node q. If the cycle contains q, then the assertion is proved. If the cycle does not contain q, then either there is a path, along forward control edges and sync edges, from q to a node z which is reachable along forward control edges from a head node of the deadlock cycle and which exits q and enters z via sync edges, or there is no such path. (See Figure 24.) If there is such a path, then there is a deadlock cycle containing q. If there is no such path, then q is not waiting on any head node of the cycle, and there must thus be another anomaly with q as a head node or one on which q waits transitively. Thus, if the cycle contains an exit node x, then either there is a cycle containing a non-bracket node q or there is a separate anomaly.

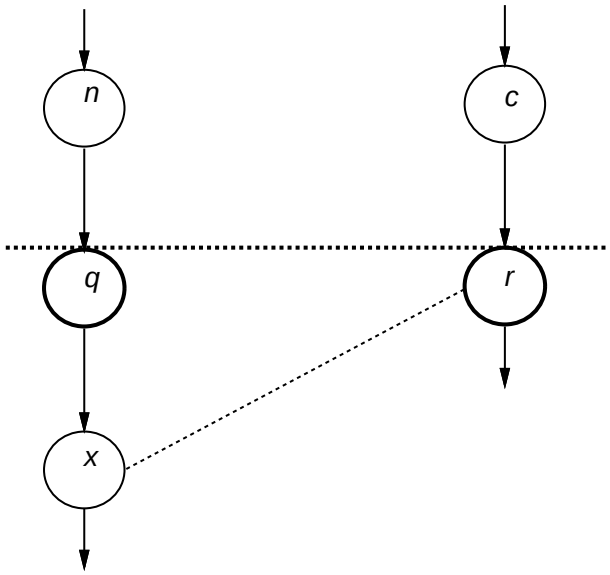

Figure 23: Deadlock cycle with a call body return node as head node.

By Lemma 54, if the deadlock cycle contains a return node, it must contain a non-bracket node.

By Lemma 55, if the deadlock cycle contains an entry node as a head node, it cannot contain any call or return nodes. Furthermore, no valid cycle can contain two entry nodes, or an entry and an exit node, since these are all in the same task, and thus can't happen together. So a deadlock cycle which contains an entry node must contain other nodes which are not in the bracket node set for the critical section.

By Lemma 56, if the deadlock cycle contains a call node, it cannot contain any entry nodes. If any call node c in a cycle is a head node, then an entry node n must be the tail node in the cycle which is connected to c via a sync edge. Thus, no single cycle contains two call nodes. (See Figure 25.) If the cycle contains c and no other members of the bracket set, then the assertion is trivially proved. If the cycle contains c and a return node r, it must contain a non-bracket node by Lemma 54. If the cycle contains c and an exit node x, then there must either be a cycle which contains a non-bracket node q or an infinite wait anomaly separate from the cycle. So a deadlock cycle which contains a call node must contain head nodes not in the bracket set, or must be transitively coupled to an anomaly containing head nodes not in the bracket set. □

Theorem 5 does not exclude the possibility of deadlock cycles containing call and exit head nodes, as the cycle (x, n', c', r', x) in Figure 22 shows. In fact, such cycles will exist in any program with more than one call body to a critical section, such that the call bodies can happen together with each other and with a critical section body. However, in such a case, the lack of a rendezvous for

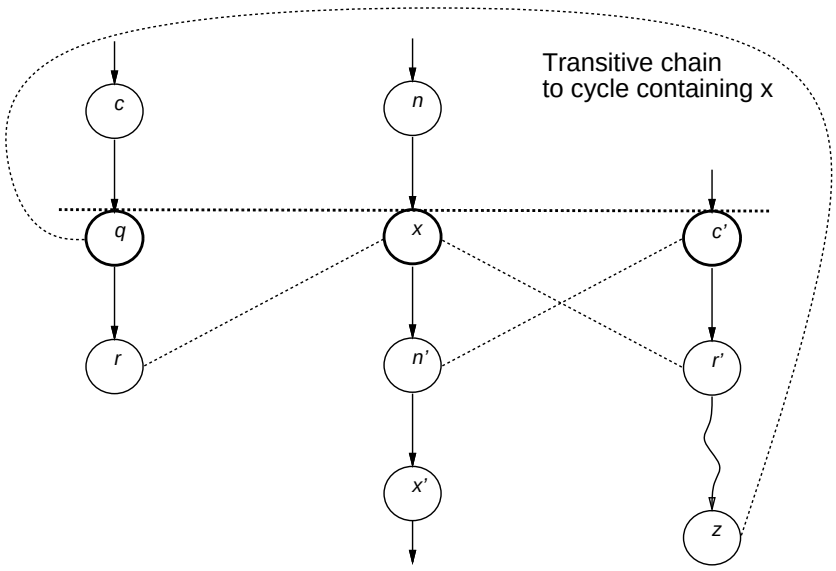

Figure 24: Cycles involving exit nodes.

c' is not the immediate reason why x cannot leave the execution wave; rather, the more immediate reason is that q cannot rendezvous and allow r to reach the wave. The cycle (x, n', c', r', x) is at most symptomatic of another problem, more directly involving q.

If the problem that causes q to have no rendezvous is that there is a deadlock cycle including q, then we can find the cycle by hypothesizing q or c' as a head node. If the problem is a deadlock cycle on which q waits transitively, then the deadlock can be found by tracing cycles which do not include the bracket nodes as head nodes.

We conclude that it is not necessary to propagate bracket nodes of a critical section to other bracket nodes of the same critical section in deadlock cycle tracing. Doing so will increase the false alarm rate, and will provide no additional deadlock information. Such tracing can easily be inhibited by preventing propagation of non-top *SigCnt* summaries for a bracket node of a critical section along sync edges to bracket nodes of the same critical section. With this step, it is possible to eliminate the spurious deadlock cycles containing nodes a and h in Figure 20, and in other programs in which critical section bodies and call bodies contain non-bracket nodes.

6.5.9 Signaling head node propagation and critical section structures.

As an optimization step in reachability computation, we avoid tracing the same cycles starting at both signaling and accept-in head nodes. Signaling nodes are thus ordinarily assumed as head nodes only to trace cycles whose head nodes are all signaling nodes. To accomplish this, signaling

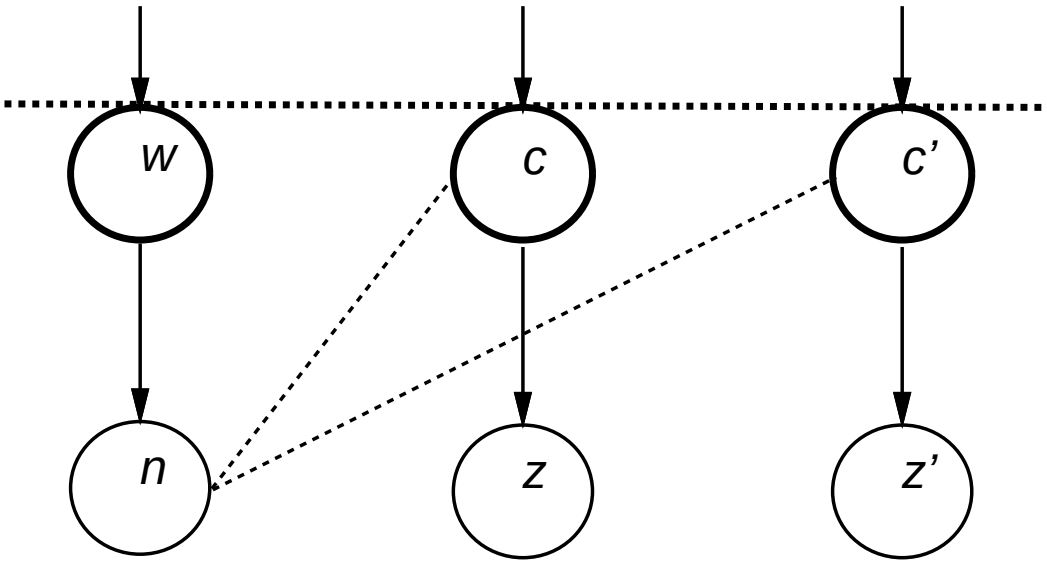

Figure 25: Cycles involving multiple call nodes.

node summaries are ordinarily propagated only to other signaling head nodes, in cases where the analysis would remain conservative.

If we eliminate propagation of bracket head node summaries to other bracket head nodes of the same critical section structure, we must allow signaling bracket node summaries to propagate to accept-in head nodes. For instance, Figure 26, a reduced example of the widely-known "dining philosophers" problem, does indeed deadlock, and all nodes in the program are bracket nodes. The deadlock cycle is (a, f, k, h, a). Nodes f and h cannot propagate to other head nodes of the cycle, since they are bracket nodes of the same critical section structure containing k and a respectively. Nodes a and k are signaling nodes; their summaries must propagate to accept-in head nodes if the cycle is to be detected.

Thus, we allow the summaries of signaling head nodes to propagate to accept-in head nodes only when the signaling nodes are bracket nodes of a critical section structure.

6.5.10 Critical sections which call other critical sections.

Here we show that neither *CHT* information, nor the propagation restriction of Section 6.5.8, can eliminate some spurious deadlock cycles that occur when calls to different critical sections are nested.

Cycle (a, b, c, d, h, g', a) of Figure 20 is a spurious deadlock cycle whose head nodes are all bracket nodes of the same critical section structure. If we do not propagate bracket head nodes of

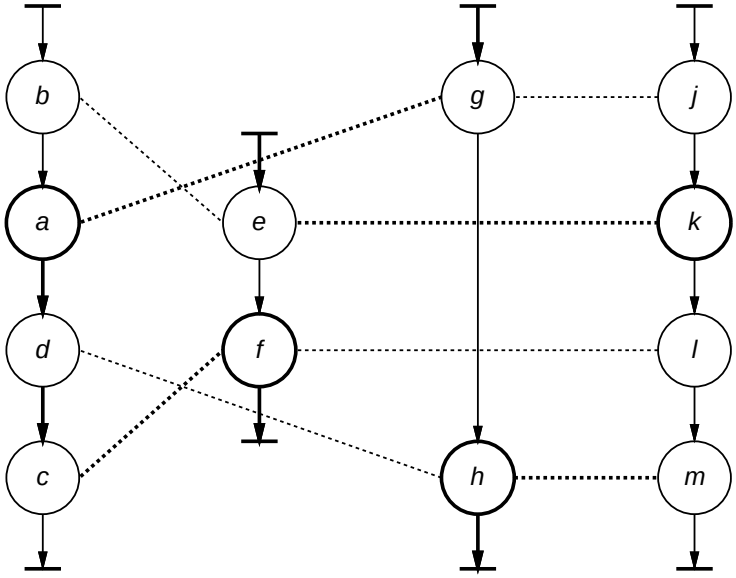

Figure 26: A simplified version of the "dining philosophers" problem with two philosopher tasks (beginning with nodes b and j) and two fork tasks (nodes e and g). The program contains a deadlock cycle (thick nodes and edges) which has only critical section bracket nodes as its head nodes.

a structure to other bracket head nodes of the same structure, we can avoid detecting this spurious deadlock.

However, other spurious cycles may involve the bracket nodes of more than one critical section structure. For instance, cycle $(a, b, c, f, e', k, l, m, h, g', a)$ (with head nodes a, f, h, and k) is spurious, because $k \in CHT(f)$. However, if a is the hypothesized head node for this cycle, the cycle cannot be eliminated, since none of the head nodes are in $CHT(a)$. Cycle $(a, b, e, f, l, m, h, g', a)$, with head nodes a, e, l, and h, is also spurious, because $l \in CHT(e)$; again, none of the head nodes are in $CHT(a)$, so the cycle is not eliminated.

A naive solution. We might try to eliminate these cycles by carrying, in the summaries for each hypothesized head node h, an (under)estimate of CHT for all head nodes of cycles including h. To maintain a conservative estimate, meet of CHT sets must be intersection.

However, this CHT summary information will not eliminate the spurious cycles. Going back to Figure 20 and Table 6.5.7, the CHT summary info for node a, propagating into node e along a sync edge, would be $CHT(a) \cup CHT(e) = \{b, c, d, f, l\}$. Similarly, propagating into f along a sync edge, we have $CHT(a) \cup CHT(f) = \{b, c, d, e, g, k, m\}$. Summaries propagate between e and f along

control edges, so the summary for a at both nodes is the meet (intersection) of the two CHT sets, i.e., $\{b, c, d\}$. This summarized CHT set does not eliminate either the cycle $(a, b, c, f, e', k, l, m, h, g', a)$ (with head nodes a, f, h, and k), or the cycle $(a, b, e, f, l, m, h, g', a)$ (with head nodes a, e, l, and h).

6.5.11 Critical section call graphs.

To resolve the problem of nested critical section calls, we can move up one level of abstraction and examine the *critical section call graph* of the program, following a construction developed by Saxena [Saxe77] for binary semaphores. The critical section call graph contains a node corresponding to each critical section, and an edge (x, y) if critical section y is called from within a call body of x at any point in the program (or if y is called from within critical section body x).

Figure 27 shows the critical section call graph for the program of Figure 20. If the critical section call graph is acyclic, then the set of critical section calls cannot deadlock.

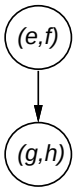

Figure 27: Critical section call graph for the program of Figure 20.

Critical section structures may, in general, have more complicated behavior than binary semaphores. For instance, the critical section bodies may contain synchronization, or may not happen together with all the call bodies. To avoid these complications, we do not construct a call graph of all critical sections. Instead, we limit ourselves to a subset of critical section structures which mimic semaphores. For this subset, a simplified analysis is possible.

6.5.12 Nested calls to simple critical sections.

In this section, we elaborate on a special case corresponding to a common use of critical sections: acquiring and releasing resources by essentially using critical section structures to mimic binary semaphores. We show how spurious deadlock cycles associated with nested critical section calls may be eliminated in this special case. Although the special case has not yet been seen in experimental data gathered from outside sources, it seems to us that it might occasionally be seen in practice; we would thus like to be prepared for this eventuality.

A critical section structure is *simple* if the bracket node pair (n, x) for each of its critical section bodies are the only rendezvous nodes in $Task(n)$. Thus, one of n or x (or the task end node of $Task(n)$) must be the entry for $Task(n)$ in all execution waves.

Simple critical section structures have several features that make deadlock cycle pruning easier than in general critical section structures. Return nodes of simple critical sections may not be participate in deadlock cycles, since an exit node of the same structure must be on the wave whenever the call body is active. Call nodes, on the other hand, may participate in deadlock cycles, but only when the cycle includes a node in a call body to the simple critical section.

Given two simple critical section structures S and S', structure S has a *nested call* to S' if any call node of S' is contained within any call body of S. We can elaborate this by transitive closure in the following recursive definition: S has a *transitively nested call* to S' if S has a nested call to S', or if S has a nested call to some structure S'' which has a transitively nested call to S'. Finally, a nested call from S to S' is an edge in a *strong component* of the critical section call graph if there is also a transitively nested call from S' to S.

Lemma 57 below shows how the nesting structure of simple critical sections may be

Lemma 57 *In a proposed anomalous execution wave E, if all head nodes in deadlock cycle $D \subseteq E$ are bracket nodes of simple critical sections, and no sync edge to any head node in D represents an edge in a strong component of the critical section call graph, then either or the deadlock D represents is only symptomatic of some other anomaly, or E is not a possible execution wave.*

Proof: Define the *dynamic nest level* of each bracket node $d \in D$ of a simple critical section structure as the maximum length of any transitive call chain from any bracket node (of a simple critical section structure) in D, to the structure containing d (or zero, if there is no such call chain for d). By the assumption that no sync edge to any head node in D represents an edge in a strong component of the critical section call graph, all such call chains are of finite length. Thus, there is at least one node (and therefore at least one structure) whose dynamic nest level is the greatest in D.

Suppose some call node c of a structure S is at the maximum dynamic nest level, as in Figure 28(a). If c is on the wave, then some exit node x of S must also be on the wave; x must also be in the cycle, since otherwise the cycle would not be connected through c. x can only be on the wave when a node z in some other call body $CallB(c', r')$ for S is in E. If $z \notin D$, then D would not be a valid deadlock cycle if z could enter the wave; D is thus only symptomatic of the anomaly which keeps z from entering the wave. If $z \in D$, then z must be a call node of a simple critical

section structure, and that call node is nested inside of a call body of S. Thus, c cannot be at the maximum dynamic nest level.

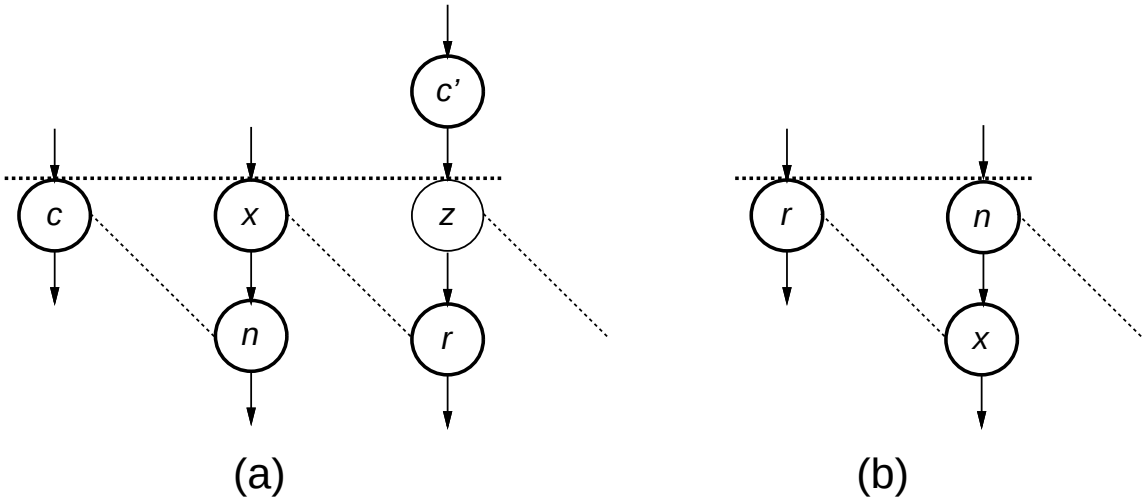

Figure 28: Simple critical section structure examples from Lemma 57.

If some exit node $x \in D$ of structure S is at the maximum dynamic nest level, then some call node c of S must also be in D. The situation of Figure 28(a) still holds in this case.

If some return node r of S is at the maximum dynamic nest level, as in Figure 28(b), then the entry node n of some critical section body of S must also be on the cycle. This is not possible, since $r \in CHT(n)$. Thus, E is not a possible execution wave. Finally, if an entry node n of S is at the maximum dynamic nest level, the situation of Figure 28(b) still applies.

Thus, either D represents a deadlock that is only symptomatic of another anomaly, or E is not a valid execution wave. □.

From Lemma 28, we conclude that we may safely avoid propagating summary information for assumed head nodes out of their tasks, along sync edges corresponding to acyclic regions of the call graph of simple critical sections. This will eliminate the spurious deadlock cycles involving calls between properly nested simple critical sections.

6.5.13 Algorithm for constructing the simple critical section call graph.

We present here an algorithm for identifying simple critical sections and building the simple critical section call graph. Because experimental evidence has not yet suggested that nested simple critical sections are common, we have not integrated it into the rest of deadlock detection; we present it here only to demonstrate that the general polynomial time bound of the algorithm would be

maintained if it were added.

Algorithm 7 *Construct the simple critical section call graph; identify bracket nodes in the sync graph corresponding to calls in acyclic regions of the simple critical section call graph.*

Input:

- The CHT form sync hypergraph, $G = (N_{CHT}, E_{C_{CHT}}, E_{S_{CHT}})$.
- The set of critical section structures, CRITSTRUCTS.

Output:

- The array $Acyclic$, such that $n \in Acyclic(m)$ if m and n are bracket nodes of simple critical section structures S and S' respectively, such that n is contained in the call body whose call node is m, and the edge (S, S') is in an acyclic region of the simple critical section call graph.

Procedure:

1. Let $SimpleCSS$ be the set of critical section structures in CRITSTRUCTS whose critical section bodies contain all nodes in their respective tasks.
2. Let E_{call} be the set of ordered pairs (S, S'), $S, S' \in SimpleCSS$ such that some call body in S contains a call node of S'.
3. Construct the simple critical section call graph $CallG = (SimpleCSS, E_{call})$.
4. Find the strong components of $CallG$.
5. Remove from E_{call} all edges between nodes in any strong component of $CallG$.
6. For each pair $(S, S') \in E_{call}$, add the bracket nodes of S' to $Acyclic(S)$.

Lemma 58 (Time bounds for the simple critical section algorithm.) *Algorithm 7 requires $\mathcal{O}(|Sigs|^4 |N_{CHT}|^2)$ time to execute, in the worst case.*

Proof: There are at most $\mathcal{O}(|Sigs|^2)$ critical section structures in the program; each of these structures may contain at most $\mathcal{O}(|N_{CHT}|)$ critical section bodies. For each of these bodies, we can look up the task containing the body in $\mathcal{O}(|Tasks|)$ time to check the number of nodes in the the task. Step 1 thus requires $\mathcal{O}(|Sigs|^2 |N_{CHT}||Tasks|)$ time in the worst case (or $\mathcal{O}(|Sigs|^2 |N_{CHT}|^2)$ time, since $|Tasks| < |N_{CHT}|$).

Step 2 can be done by checking the set of signal types of nodes in the call bodies of each structure against the set of signal types of call nodes of the other structures. This may be accomplished in $\mathcal{O}(|Sigs|^2 \log(|Sigs|))$ time, by looking up the signal type for the call nodes each critical section structure in a height-balanced tree of signal types of nodes in the call bodies. Attached to each node of the height-balanced tree is a set of up to $\mathcal{O}(|Sigs|^2)$ critical section structures whose call bodies contain nodes of the given signal type. Expanding this set to a set of edges in E_{call} takes $\mathcal{O}(|Sigs|^2)$ time per structure, for a total of $\mathcal{O}(|Sigs|^4)$, which is also an upper bound on the size of E_{call}. Step 3 can then be done in constant time.

The strong components of $CallG$ may be found in time proportional to the number of nodes and edges of $CallG$ [AHU74]; step 4 thus takes $\mathcal{O}(|Sigs|^4)$ in the worst case. Step 5 can then be done via a simple scan of E_{call}, in $\mathcal{O}(|Sigs|^4)$ time.

Finally, for each pair (S, S') remaining in E_{call}, we add the bracket nodes of S' to the set $Acyclic(s)$ for each bracket node s of S. This step takes at most $\mathcal{O}(|Sigs|^4 |N_{CHT}|^2)$ time.

Thus, the total time to construct the $Acyclic$ array is $\mathcal{O}(|Sigs|^4 |N_{CHT}|^2)$ time. □

6.6 A more general definition of critical sections.

The critical section structures of Section 6.5 require that the critical section and call bodies be delineated as single-entry, single-exit regions. In this section, we show how this restriction can be weakened, and show how a multiple-entry, multiple-exit region can enforce CHT between a pair of nodes. We emphasize that the analysis of this section has not yet been integrated into the deadlock algorithm; we show only some examples in which they might be used in a future algorithm.

6.6.1 Two-task mutual exclusion.

To insure that $m \in CHT(n)$, we must be able to show the following:

- No instances of m and n can start at the same time.
- Whenever any new instance of m starts, no instance of n is executing.
- Whenever any new instance of n starts, no instance of m is executing.

To demonstrate this, it is sufficient to show:

- For each instance m_j, there is a sequence of instances $(x(1) = d_i, ..., x(k) = n_j)$, where d_i is a descendant of m_i within $L_i(m)$. We will call this sequence an *ordering chain* from m_i to n_j.

We further require that ordering chains have the following properties:

- Each $x(h+1)$ is either a proper control successor of $x(h)$, or does rendezvous with $x(h)$;
- n_j is a proper control successor of $x(k-1)$;
- When the chain enters a task, it leaves the task in the same loop body in which it entered.

  More formally, for each subchain

  $$(x(h+1), x(h+2), ..., x(h+g))$$

  such that the subchain from $x(h+1)$ to $x(h+g)$ contains all elements of the chain that are in a particular task, instance $x(h+1)$ is in the same iteration as $x(h+g)$.

  The subchain from $x(h+1)$ to $x(h+g)$ contains all elements of the chain that are in a particular task if:

  * $x(h+1) = d_i$ and $x(h+g) = n_j$, and each node $x(h+e), 2 \leq e \leq g$ is the control successor of $x(h+e-1)$; or,
  * $x(h+1) = d_i$ and $x(h+g)$ does rendezvous with $x(h+g+1)$, and each node $x(h+e), 2 \leq e \leq g$ is the control successor of $x(h+e-1)$; or,
  * $x(h+g) = n_j$, and $x(h)$ does rendezvous with $x(h+1)$, and each node $x(h+e), 2 \leq e \leq g$ is the control successor of $x(h+e-1)$; or,
  * $x(h)$ does rendezvous with $x(h+1)$, and $x(h+g)$ does rendezvous with $x(h+g+1)$, and each node $x(h+e), 2 \leq e \leq g$ in the subsequence is the control successor of $x(h+e-1)$.
- If $x(g)$ does rendezvous with $x(g+1)$ and $x(h \neq g)$ does rendezvous with $x(h+1)$, then $Task(g) \neq Task(h+1)$.

The existence of an ordering chain establishes that, for instance n_j, there exists some instance m_i (or its descendant within iteration i of its loop body) such that m_i is ordered before n_j.

- No new instance $m_{i'}, i' > i$ can start execution until n_j finishes.
- When the next instance $m_{i'}, i' > i$ starts, no instance of n can be executing.
- When the next instance $n_{j'}, j' > j$ starts, no instance of m can be executing.

Theorem 6 gives specific conditions on the sync hypergraph, for which we can prove $m \in CHT(n)$.

Theorem 6 *Let m and n be nodes in the sync hypergraph. Let X be a subset of proper ancestors of n within $L(n)$ which can rendezvous only with descendants of m within $L(m)$, and let W be the set of nodes with which X may rendezvous.*

Let Z be a set of proper descendants of m within $L(m)$ which can only rendezvous with descendants of n within $L(n)$, and let Y be the set of nodes with which the members of Z may rendezvous. If the following conditions hold:

- *X cuts $(\{LS(n)\}, \{n\})$;*
- *Z cuts $(\{m\}, \{LE(m)\})$;*
- *Z cuts $(W, \{LE(m)\})$.*

Then no instance of m can happen together with any instance of n.

Proof: See Figure 29. Without loss of generality, m and n are in different tasks. Since X cuts $(\{LS(n)\}, \{n\})$, (and, presumably, there is a control path from $LS(n)$ to n), X is non-empty. Since Z cuts $(\{m\}, \{LE(m)\})$, Z is non-empty.

Before each instance n_j of n, a rendezvous must occur between instance i of some node in W and instance j of some node in X. Suppose that n_j executes, and that $x_j \in X_j$ has a rendezvous with instance $w_i \in W_i$. If m_i executes, then either $m_i = w_i$ or m_i precedes w_i. Thus, if m_i and n_j both execute, then m_i precedes n_j.

Next we show that when the next instance $n_{j'}, j' > j$ executes, then any instance of m which started after n_j must be off the wave when $n_{j'}$ starts. Suppose that instances $m_{i+1}...m_{i'}$ start between n_j and $n_{j'}$. In order for $n_{j'}$ to start, some node instance in $X_{j'}$ must complete a rendezvous with an instance of a node in W. If this instance is in $W_{i'}$, then $m_{i'}$ must complete before $n_{j'}$ starts. If this node is in $W_{i''}, i < i'' < i'$, then $m_{i'}$ cannot start execution until some instance in $Z_{i''}$ has rendezvoused with an instance in $Y_{j'}$, i.e., not until $n_{j'}$ is off the wave. This contradicts the assumption that $m_{i'}$ starts before $n_{j'}$ does.

Finally, we show that when the next instance $m_{i'}, i' > i$ executes, then any instance of n which starts after m_i must be off the wave when $m_{i'}$ starts. Preceding any instance of n, a node in X must rendezvous with a node in W. Therefore, any instance $n_{j'}, j' > j$ which starts between the end of m_i and the start of $m_{i'}$ must do so following a rendezvous between a node instance in X_j and a node instance in $W_i \cup W_{i+1} ... \cup W_{i'-1}$. But, in order for a new instance $m_{i'}$ to start after a

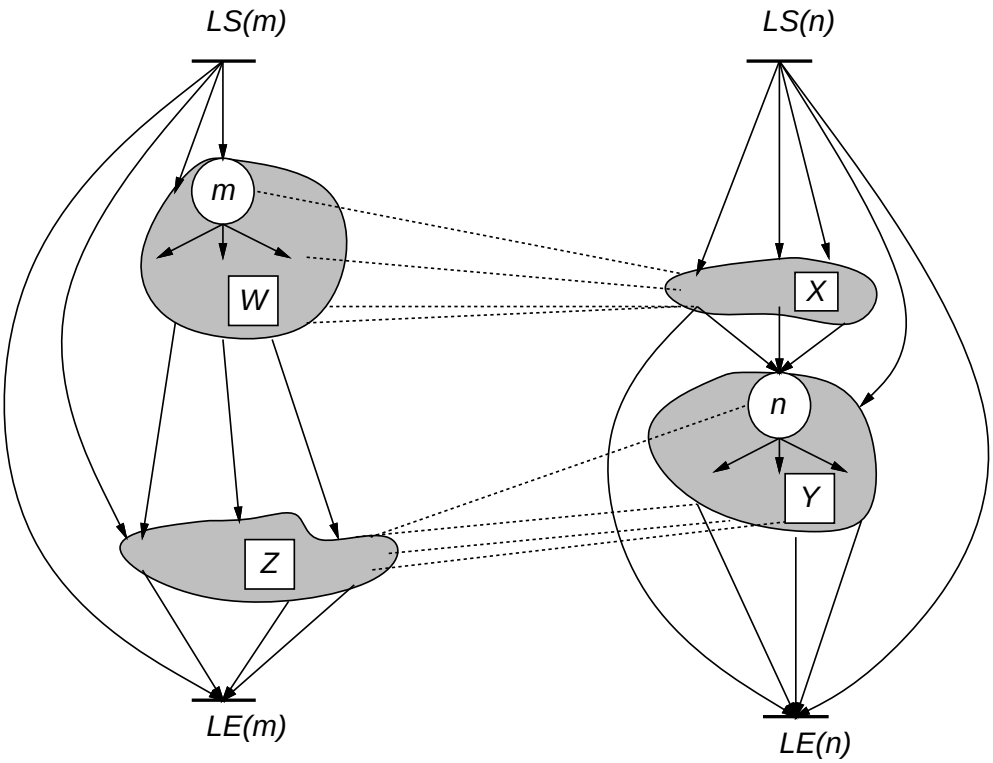

Figure 29: *CHT* example from Theorem 6.

node in W is executed, control in $Task(m)$ must leave W and pass through Z. In order for control to pass through Z, a node in Z must rendezvous with one in Y. Y includes only descendants of n. Therefore, the last instance of n which started before $m_{i'}$ started must have completed by the time $m_{i'}$ starts.

Therefore, no instance of m can happen together with any instance of n. □

Note that the sets X and Z (and therefore W and Y) satisfying the preconditions of Theorem 6 need not be unique. In particular, there may be X and Z sets that satisfy some of the preconditions of Theorem 6, but not others. For instance, the maximal X (the set of all proper ancestors of n which can rendezvous only with descendants of m) may induce some members in W that prevent Z from cutting $(W, LE(m))$. (See Figure 30.)

6.6.2 An example of *CHT* with three tasks.

In this section, we present an example in which *CHT* is enforced by multiple-entry, multiple-exit regions in three tasks. We present this mainly to show the problems involved in generalizing the results of Section 6.6.1 to mutual exclusion in an arbitrary number of tasks.

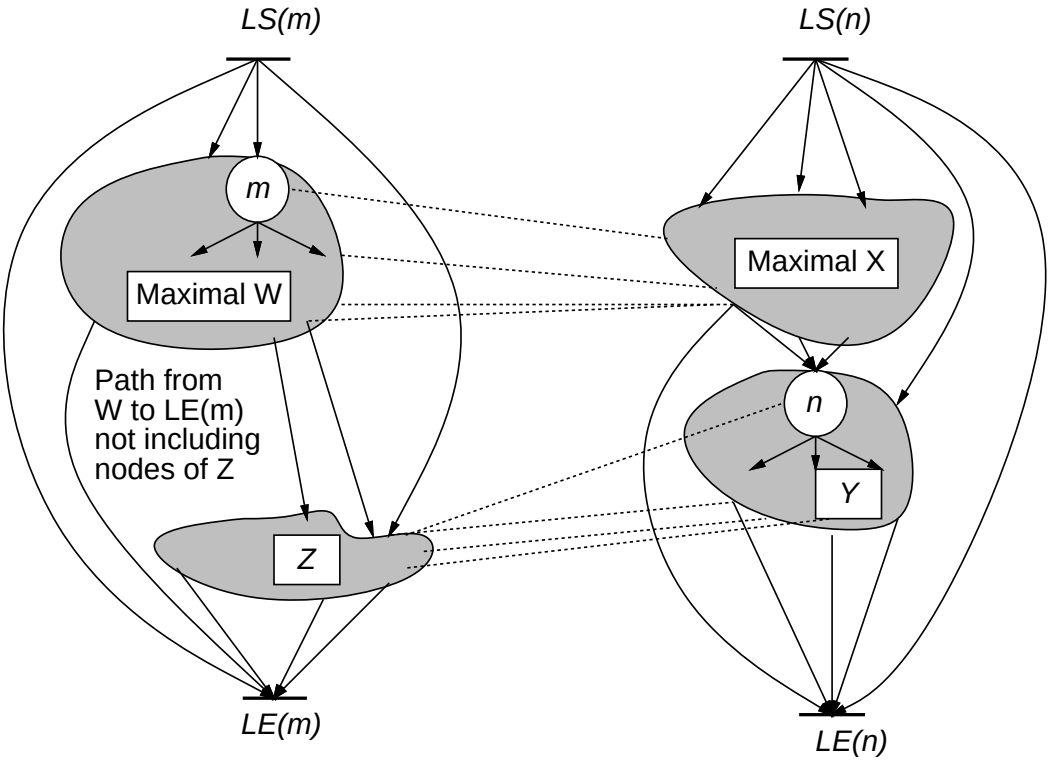

Figure 30: Maximal X set that induces a W that fails to satisfy the preconditions of Theorem 6. Since X is too large, Z no longer cuts $(W, \{LE(m)\})$.

Figure 31 shows an example in which the constraints of Theorem 6 are not obeyed, yet $m \in CHT(m)$. The following are true of the regions shown in Figure 31:

- Z postdominates m and W;
- X' dominates W';
- X' must rendezvous with W;
- X dominates n and Y;
- X must rendezvous with W or W';
- Y must rendezvous with Z';
- Y' must rendezvous with Z;
- Z' postdominates W' and dominates Y'.

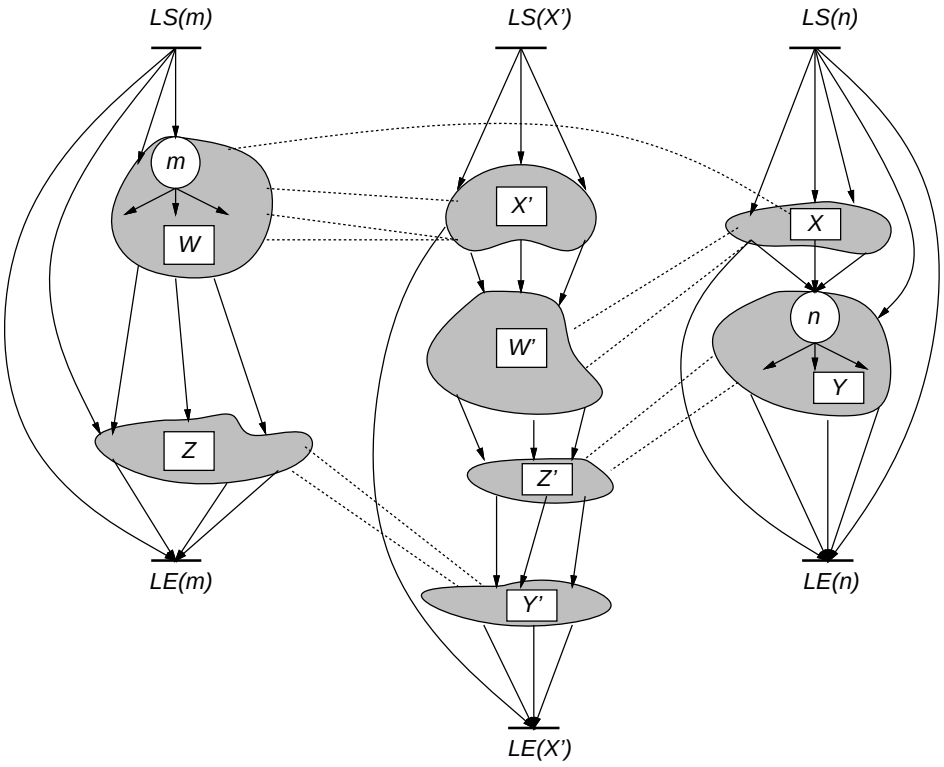

Figure 31: A program in which $m \in CHT(m)$ is enforced by three tasks.

Since X dominates n, some instance of a node in X must complete before any instance n_j can start. Nodes in X can rendezvous only with nodes in W and W'. So some node in W or W' must complete before any n_j can start.

Since X' dominates W', some instance in X' must complete before any instance in W' can start. Since X' can only rendezvous with W, and the nodes in W are descendants of m, some descendant of m must complete before any node in X' can complete, and thus before any node in W' can start. Therefore, some instance in W must complete before any instance of n can start. There is thus an ordering chain from a descendant $d_i \in W_i$ of $m_i \in W_i$ to each instance of n_j, even if m_i does not execute.

Now we will show that no instance $m_{i'}, i' > i$ can start while n_j is still executing. After n_j starts and while it is still executing, control in $Task(a)$ must be in W or Z, or between the two. Control in $Task(X')$ must be in some ancestor of Z'. In order for control to pass to m again, there must be a rendezvous between Z and Y'. But Y' cannot be reached until control passes through Z', and this will only happen if there is a rendezvous between Y and Z'. When this last rendezvous occurs, n_j will be taken off the wave. Therefore, instance $m_{i'}$ must start after n_j finishes.

Next, we will show that no instance of m can start while any instance of n is executing. Suppose that $m_{i'}$ is the next instance of m to start after m_i finishes. Suppose also that, in the period from the end of m_i to the start of $m_{i'}$, instance $n_{j'}, j' > j$ is the last instance of n to start. For $n_{j'}$ to start, control must have entered W or W', so that a rendezvous with some member of $X_{j'}$ could occur. Following this, for $m_{i'}$ to start, control must have passed through Z, implying that control had reached Y' at the same time. For control to reach Y', a rendezvous between Z' and Y must have occurred, implying that $n_{j'}$ finished. So $n_{j'}$ cannot still be executing when $m_{i'}$ starts.

Last, we will show that the next instance of n cannot start while any instance of m is executing. Suppose the next instance of n is $n_{j'}, j' > j$, and $m_{i'}, i' > i$ is the last instance of m to start before $n_{j'}$ starts. In order for $n_{j'}$ to start, a rendezvous must have occurred between X and either W or W'. If this rendezvous was between X and W, then $m_{i'}$ cannot be executing when $n_{j'}$ starts. If the rendezvous was between X and W', then no instance of m could have been executing when the rendezvous occurred, and no further instance of m could start until Y had been entered. Thus, $m_{i'}$ could not have been executing when $n_{j'}$ started.

6.7 Remote procedure call analysis.

If some set of nodes S is contained within an `accept-do` construct of signal type t, then the nodes of S must happen together with a signaling node of type t. This leads to the following:

Lemma 59 *If some set of nodes S is contained within an* `accept-do` *construct of signal type t, and Q is the set of signaling nodes that can call S, and $CHT_{perf} \supseteq CHT$, then, for each $s \in S$, $CHT_{perf}(s) \supseteq \bigcap_{q \in Q} CHT(q)$.*

Proof: The set $\bigcap_{q \in Q} CHT(q)$ is a set of nodes that can't happen together with any node in Q. Since only the nodes of q can call the `accept-do` construct in which S is contained, any node that can't happen together with all members of Q can't happen together with any member of S. □

The following algorithm may be used to augment CHT information using the embedded `accept-do` construct. Algorithm 8 does not call *Widen*; instead, it performs a less expensive set of vector operations to achieve the same widening effect on the CHT sets of nodes embedded in remote procedure calls.

Algorithm 8 *Remote procedure call analysis.*

Inputs:

- The CHT sets for each node.

- For each `accept-do` construct A containing embedded synchronization:
  - The set of nodes $N(A)$ contained within the construct A.
  - The accept-in node $AIN(A)$ of the construct.
  - The signaling nodes $S(A)$ which call A.

Output: The CHT set for each node, augmented by remote procedure call information. Denoted CHT'.

Procedure:

1. For each `accept-do` construct A do:
   (a) $ADDED \Leftarrow N_{CHT}$.
   (b) For each $s \in S(A)$ such that $s \notin CHT(AIN(a))$ do:
      i. $ADDED \Leftarrow ADDED \cap CHT(s)$.
   (c) For each $n \in N(A)$ do:
      i. $CHT'(n) \Leftarrow CHT(n) \cup ADDED$.

Lemma 60 (Correctness of Algorithm 8.) *If $CHT \subseteq CHT_{perf}$ and the other parameters of Algorithm 8 are correctly specified, then $CHT \subseteq CHT' \subseteq CHT_{perf}$.*

Proof: From Theorem 1 and Lemma 59. □

Lemma 61 (Time bounds for the remote procedure call algorithm.) *The time to execute Algorithm 8 is $\mathcal{O}(|N_{CHT}|^3)$ bit operations in the worst case.*

Proof: A program may have $\mathcal{O}(|N_{CHT}|)$ `accept-do` constructs, each called by $\mathcal{O}(|N_{CHT}|)$ signaling nodes. Thus, the first inner loop (statement 1b) may have to execute $\mathcal{O}(|N_{CHT}|^2)$ times. Similarly, in pathological cases, an `accept-do` construct may contain $\mathcal{O}(|N_{CHT}|)$ nodes, so the second inner loop (statement 1c) may have to execute $\mathcal{O}(|N_{CHT}|^2)$ times. Assuming sets of nodes are represented as bit vectors of length $|N_{CHT}|$, the total time for Algorithm 8 is $\mathcal{O}(|N_{CHT}|^3)$ bit operations in the worst case. □

In practice, we have found that `accept-do` statements with embedded rendezvous statements are quite rare, so the pathological worst-case time of Lemma 61 should not occur frequently in practice.

6.8 Iterative *CHT* analysis.

We may use any combination of *B4* analysis, pinning analysis, critical section analysis, and remote procedure call analysis, to derive a *CHT* relation for a program. Furthermore, we may use any of these techniques to iteratively refine *CHT* information, in the following ways.

First, pseudotransitivity allows *CHT* information to propagate from node to node in pinning, critical section, or remote procedure call analysis. Thus, *CHT* information found in one analysis phase may be used to derive further *CHT* information in another.

Second, if any signaling and accept-in nodes that can't happen together are connected by a sync hyperedge, the rendezvous corresponding to that hyperedge can never occur. Therefore, the hyperedge in question may be eliminated safely from the sync hypergraph. With the hyperedge eliminated, it is possible that further analysis may be able to uncover more *CHT* information.

We base the following algorithm on this form of iterative analysis. Note that all four component *CHT* algorithms intrinsically eliminate sync edges between nodes that can't happen together.

Algorithm 9 *Find the CHT sets of each node in a program.*

Input: The sync hypergraph of the program.

Output: A *CHT* set for each node in the program.

Procedure: The functions *Pinning*, *RemoteProc*, *CriticalSect*, and *B4Analysis* correspond to the obvious refinements; parameters which are constant within a single problem are omitted here for clarity.

1. For each node n, $CHT(n) \Leftarrow Task(n) - \{n\}$.
2. $CHT_{old} \Leftarrow \emptyset$.
3. While $CHT_{old} \neq CHT$ do:
   (a) $CHT_{old} \Leftarrow CHT$.
   (b) $CHT \Leftarrow Pinning(CHT)$.
   (c) $CHT \Leftarrow RemoteProc(CHT)$.
   (d) $CHT \Leftarrow CriticalSect(CHT)$.
   (e) $CHT \Leftarrow B4Analysis(CHT)$.

Analysis	Time	Lemma
Pinning	$\mathcal{O}(\|Tasks\|\|N_{CHT}\|^3)$	33
Remote procedure	$\mathcal{O}(\|N_{CHT}\|^3)$	61
Critical section setup	$\mathcal{O}(\|N_{CHT}\|^3)$	51
Critical section update	$\mathcal{O}(\|N_{CHT}\|^3)$ per bit	52
B4 (worklist)	$\mathcal{O}(\|N_{CHT}\|^3 \log \|N_{CHT}\|)$	22
B4 (Tarjan)	$\mathcal{O}(\|N_{CHT}\|^3)$	22

Table 5: Worst-case time for each form of *CHT* analysis.

Lemma 62 (Correctness of *CHT* analysis.) *Algorithm 9 terminates with* $CHT_{perf} \supseteq CHT$.

Proof: By induction on the number of refinements applied. Let CHT_i be the *CHT* set following the i'th refinement.

Basis: $CHT_0(n)$ is $Task(n) - \{n\}$ for each node n. Obviously, $CHT_{perf}(n) \supseteq CHT_0(n)$ for all n.

Induction step: Suppose $CHT_{perf} \supseteq CHT_{i-1}$, and we apply the i'th refinement step to CHT_{i-1} to get CHT_i.

If the i'th refinement is *B4* analysis, then Lemma 21 shows that $CHT_{perf} \supseteq CHT_i \supseteq CHT_{i-1}$.

If the i'th refinement is pinning analysis, then $CHT_{perf} \supseteq CHT_i \supseteq CHT_{i-1}$ by Lemma 32.

If the i'th refinement is critical section analysis, then $CHT_{perf} \supseteq CHT_i \supseteq CHT_{i-1}$ by Lemma 49.

If the i'th refinement is remote procedure call analysis, then $CHT_{perf} \supseteq CHT_i \supseteq CHT_{i-1}$ by Lemma 60.

Thus, $CHT_{perf} \supseteq CHT_i$. Also, since $CHT_{i-1} \subseteq CHT_i$ for $i > 0$, $CHT_{i-1} = CHT_i$ for $i > 4|CHT_{perf}|$ (where we take $|CHT_{perf}| = \sum_{n \in N} |CHT_{perf}(n)|$). Thus, at most $4|CHT_{perf}|$ refinements can be done before the algorithm terminates. □

6.9 Total time for *CHT* analysis.

Table 5 shows the worst-case time for each technique used in *CHT* analysis.

Lemma 63 (Time bounds for the *CHT* algorithm.) *The total time for Algorithm 9 is* $\mathcal{O}(|Tasks||N_{CHT}|^5 \log(|N_{CHT}|))$ *when B4 is solved using the worklist algorithm, and* $\mathcal{O}(|Tasks||N_{CHT}|^5)$ *when B4 is solved using Tarjan's algorithm.*

Proof: From Table 5, each iteration (or bit change) of CHT will require

$$\mathcal{O}(|Tasks||N_{CHT}|^3 \log(|N_{CHT}|))$$

time using the worklist algorithm. At most $|N_{CHT}|^2$ iterations must be done, if one bit of CHT information is added per iteration. The total time for Algorithm 9 is thus

$$\mathcal{O}(|Tasks||N_{CHT}|^5 \log(|N_{CHT}|))$$

when $B4$ is solved using the worklist algorithm.

Similarly, each iteration (or bit change) of CHT will require $\mathcal{O}(|Tasks||N_{CHT}|^3)$ time using Tarjan's algorithm, for a total of $\mathcal{O}(|Tasks||N_{CHT}|^5)$ time. □

6.10 Additional uses of the CHT relation.

In addition to static detection of deadlocks, the CHT relation has several other notable uses.

The first, and simplest, additional use of CHT is in the detection of unexecutable statements. If $CHT(n) = N_{CHT}$, then we know that n cannot execute. This may indicate either an infinite wait anomaly involving the control ancestors of n, or the situation that n is unreachable in control flow.

The second additional use of CHT is in data flow analysis involving communications between tasks [LC91]. For example, consider reaching definitions analysis. If two tasks do not share memory, then definitions can propagate directly between them only through the parameters of a rendezvous. The rendezvous cannot occur if the corresponding signaling and accept-in nodes can't happen together. If we know this, we can eliminate the corresponding sync hyperedge, thus improving the accuracy of the solution.

Finally, we may be able to use CHT information for optimization of rendezvous code. Given that an entry call can't happen together with an entry of the same signal type, we can eliminate from the code for the entry any specialized code or data structures (semaphores, case statements, etc.) corresponding to the call. This may permit both speed and space optimization, depending on the implementation of the rendezvous.

7 Signaling count lattice framework.

We define a signaling count lattice framework in which the functions correspond to edges in the HCLG [MR91a]. A *signaling node count summary* $SigCnt$ is either:

- A vector of nonnegative integers of length $|Sigs|$, such that the y'th element of the vector is not more than some constant k_y, or;
- The distinguished top element $SigCnt_\top$.

For a node n, $SigCnt$ represents a lower bound on the number of signaling head nodes of each signal type encountered on valid paths from a particular head node h to n. k_y above is the total number of signaling nodes of a given signal type y in the sync graph. $SigCnt_i \sqsubseteq SigCnt_j$ if either:

- $SigCnt_j = SigCnt_\top$, or;
- $SigCnt_i \neq SigCnt_\top$ and $SigCnt_j \neq SigCnt_\top$ and $SigCnt_i[y] \leq SigCnt_j[y]$ for all $y \in Sigs$.

The meet operation is thus an element-wise minimum. The lattice subspace L_{SigCnt} is:

$$L_{SigCnt} \subset \mathcal{N}^{|Sigs|} \cup \{SigCnt_\top\}$$

$$SigCnt_\top \in L_{SigCnt}$$

where $\mathcal{N}$ represents the set of nonnegative integers. For convenience, let $SigCnt_\perp$ denote a vector of all zero elements of length $|Sigs|$.

Lemma 64 *The height of the L_{SigCnt} lattice is at most $|N| + 2$.*

Proof: The $\sqsubseteq$ relation is defined for L_{SigCnt} as element-wise $\leq$. The maximum value other than $SigCnt_\top$ that any element of L_{SigCnt} may have is a vector $(k_1, k_2, ..., k_{|Sigs|})$, where $\sum_{y \in Sigs} k_y \leq |N|$ (since each k_y is the number of signaling nodes of type y in G.) Thus, $SigCnt_\perp$ can be increased a maximum of $|N|$ times before reaching the maximum possible value (other than $SigCnt_\top$). □

7.1 Function class for $SigCnt$ framework.

The function class $F_{SigCnt} = \{f : L_{SigCnt} \rightarrow L_{SigCnt}\}$ is the transitive closure of the the set of possible edge functions under meet and function composition. A particular edge e will be represented by one of the following function types:

- The *identity function* $f_\iota(x) = x$.

- The *top function* $f_\top(x) = SigCnt_\top$.
- The *bottom function* $f_\bot(x) = SigCnt_\bot$.
- A *conditional identity* function $f_e(x)$ such that, if $x[y] \leq V_e[y]$ for some constant vector of integers V_e and for $y \in Y_e$ where Y_e is a constant set of signal types, then $f_e(x) = x$. Otherwise $f_e(x) = SigCnt_\top$.
- A *conditional increment* function $f_e(x)$ such that, if $x[y] \leq V_e[y]$ for some constant vector of integers V_e and for $y \in Y_e$ where Y_e is a constant set of signal types, then:
  - $f_e(x)[y] = x[y], y \in Sigs, y \neq y_e$, where y_e is a constant signal type;
  - $f_e(x)[y_e] = \min(x[y_e] + 1, k_{y_e})$.

  Otherwise $f_e(x) = SigCnt_\top$.

Note that the identity and top functions are special cases of the conditional identity function.

Lemma 65 *The set of edge functions is monotonic. Furthermore, with the exception of $f_\bot$, the set of edge functions is nondecreasing.*

Proof: Clearly, f_ι and $f_\top$ are monotonic and nondecreasing, and $f_\bot$ is monotonic.

Since a conditional identity function $f_e(x)$ has a value of either $SigCnt_\top$ or x, it is nondecreasing. To show that f_e is monotonic, assume some $a, b \in L_{SigCnt}$ such that $a \neq SigCnt_\top, b \neq SigCnt_\top$, and $a \sqsubseteq b$. If neither $f_e(a)$ nor $f_e(b) = SigCnt_\top$, then $f_e(a) = a$ and $f_e(b) = b$; thus $f_e(a) \sqsubseteq f_e(b)$. If $f_e(b) = SigCnt_\top$, then $f_e(a) \sqsubseteq f_e(b)$.

Now suppose that $f_e(a) = SigCnt_\top$ and $f_e(b) \neq SigCnt_\top$. This would imply that, for some $y \in Sigs$, $a[y] > V_e[y] \geq b[y]$. By the assumption that $a \sqsubseteq b$, though, $a[y] \leq b[y]$, so this case cannot occur. Therefore, the conditional identity functions must be monotonic.

A very similar argument will show that the conditional increment functions are also monotonic and nondecreasing. □

Let F^-_{SigCnt} be the transitive closure of the set of edge functions excluding $f_\bot$ under meet and function composition.

Lemma 66 *The functions of F^-_{SigCnt} are monotonic, nondecreasing, and 1-semibounded.*

Proof: By Lemma 6, the set of all monotonic nondecreasing functions on L_{SigCnt} is closed under composition and meet. Since F^-_{SigCnt} is a closure of a set of monotonic and nondecreasing functions on L_{SigCnt}, the functions of F_{SigCnt} must also be monotonic and nondecreasing. Lemma 8 shows that F^-_{SigCnt} is 1-semibounded. □

Lemma 67 *F^-_{SigCnt} contains functions that are not distributive.*

Proof: Consider a conditional identity function f_e such that $Y_e = \{y_1, y_2\}$, and two summaries $a, b \in L_{SigCnt}$ such that:

$$\begin{aligned} a[y_1] &< V_e[y_1] \\ a[y_2] &> V_e[y_2] \\ b[y_1] &> V_e[y_1] \\ b[y_2] &< V_e[y_2]. \end{aligned}$$

Thus, $f_e(a) = f_e(b) = SigCnt_\top$. But $(a \sqcap b)[y_1] < V_e[y_1]$ and $(a \sqcap b)[y_2] < V_e[y_2]$, so $f_e(a \sqcap b) = a \sqcap b \sqsubset SigCnt_\top$.

A similar argument holds for the conditional increment functions. □

Lemma 67 implies that the time complexity results of Lemma 11 cannot be used in the *SigCnt* problem.

Lemma 68 *The meet operation in L_{SigCnt} requires $\mathcal{O}(|Sigs|)$ time to compute.*

Proof: From the definition of the meet operation and the L_{SigCnt} lattice. □

Lemma 69 *An edge function application requires $\mathcal{O}(|Sigs|)$ time to compute.*

Proof: The conditional identity and conditional increment functions require a scan of some elements of their argument against corresponding elements of a constant vector. This takes $\mathcal{O}(|Sigs|)$ time. Creating the result also requires $\mathcal{O}(|Sigs|)$ time for initialization. □

Lemma 70 *A worklist iterative algorithm using a lattice framework $D = (G, L_{SigCnt}, F^-_{SigCnt}, M)$ will converge from an initialization of all elements at $SigCnt_\top$ in time $\mathcal{O}(|Sigs||N|^3 \log(|N|))$ in the worst case.*

Proof: From Lemmas 9 and 64, the worklist algorithm will require no more than $(|N| + 2)|N|^2\lceil \log_2(|N|)\rceil$ meet operations, $(|N| + 2)|N|^2$ edge function evaluations, and $\mathcal{O}(|N|^2 \log(|N|))$ time to maintain the worklist in the worst case. From Lemmas 69 and 68, this will result in a worst case time of $\mathcal{O}(|Sigs||N|^3 \log(|N|))$. □

7.2 Signal counts and 1-semiboundedness.

The only edge functions above which are nonincreasing are the $f_\perp$ functions, and these are mapped only to control edges leaving the HCLG "out" node h' corresponding to the hypothesized head node h. With a simple transformation peculiar to each h, it is possible to eliminate the need for the $f_\perp$ functions, and thereby to have a 1-semibounded framework. However, we can also show that the presence of the $f_\perp$ functions does not affect the speed of convergence.

To eliminate the need for $f_\perp$ edges, we simply split h' into two nodes, h_s and h_d. (See Figure 32.) h_s acts as the source of a $SigCnt_\perp$ signal. All control edges formerly leaving h' would leave h_s after the split, and are mapped to identity edge functions. (Note that no HCLG edges corresponding to sync hyperedges leave h', so none leave h_s either.) h_s has no edges entering it.

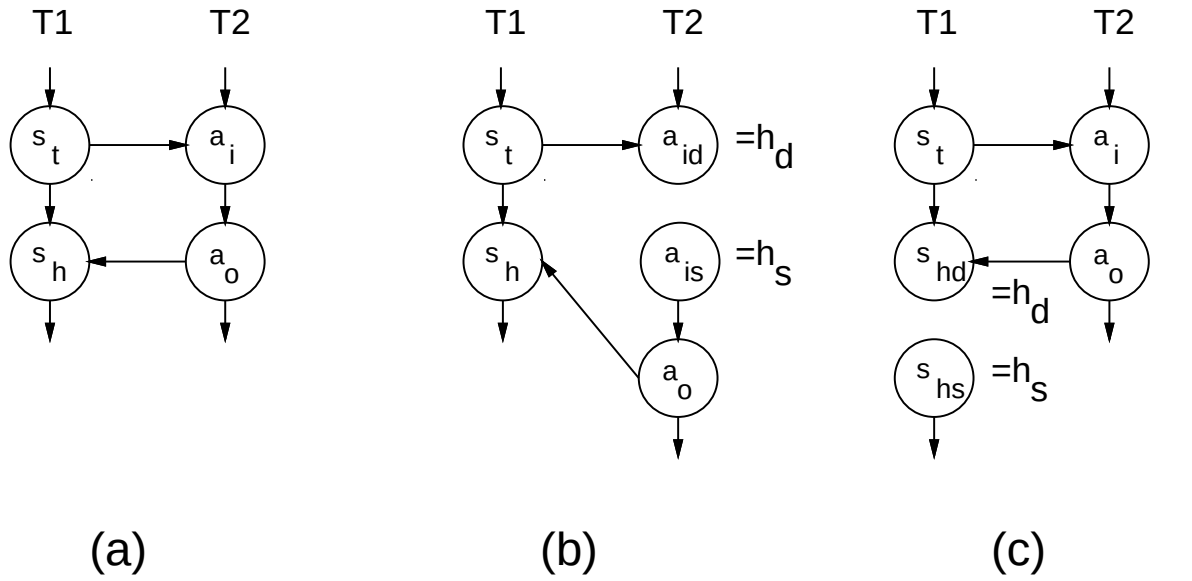

Figure 32: Splitting a hypothesized head node. (a) Original HCLG segment. (b) Node a_i split. (c) Node s_h split.

Node h_d acts as the destination node of propagation; a possible deadlock cycle involving h happens if h_d has a value other than $SigCnt_\top$ after convergence. All edges which formerly entered h' would enter h_d after the node split, and retain their original function mappings. h_d has no edges leaving it.

Because the edge functions in the HCLG are all nondecreasing after the node split, Lemma 8 asserts that the resulting function space is 1-semibounded. We would solve the reachability problem for h by simply setting the value of h_s initially to $SigCnt_\perp$, and all other node values to $SigCnt_\top$, and iterating to convergence. However, we may effectively obtain the same convergence time without transforming the HCLG by simply not evaluating the edge functions for edges out of h' after the first evaluation, since their values will never change. Thus, through a slight trick, the $SigCnt$ problem can be seen to converge in the same time as a 1-semibounded problem.

Lemma 71 *The total time for solution of a SigCnt lattice problem using Tarjan's path expression algorithm is* $\mathcal{O}(|N|^2|Sigs|)$ *in the worst case.*

Proof: Direct from Lemmas 10, 66, 68 and 69. □

A similar node-splitting operation is used by Zadeck [Zade84] to prove that a class of lattice frameworks $\mathcal{O}(|N| + |E|)$, assuming that the time to perform a meet operation is constant. Zadeck's node functions (and therefore his edge functions) are either constant valued or non*increasing* and distributive; nodes with constant valued outputs are split, as in our model. In addition to the time bound, Zadeck's model yields an exact solution (the meet over all paths equals the maximum fixed point.) Since our edge functions are not distributive, we do not have these convergence time and exactness results.

7.3 Mapping function in *SigCnt* framework.

To represent an edge $e = (m, n)$ in HCLG G in this framework for a hypothesized head node h, select $f_e \in F_{SigCnt}$ such that:

- If e represents a control edge (m', n') in the sync hypergraph:
  - If $m' \neq h$, then $f_e = f_\iota$.

    This allows unimpeded propagation along control edges within tasks, unless the edge is used for initialization.
  - If $m' = h$, then $f_e = f_\perp$.

    This starts propagation along a control edge leaving h.
- If e is an edge (s_e, s_d) in the HCLG for some signaling node s in the sync hypergraph, then $f_e = f_\iota$.

  Propagation is unimpeded along "pseudo-control" edges added between split copies of signaling nodes.
- If e is an HCLG edge (s_e, a_i) representing mode (b) propagation:
  - If h is a signaling node, and is not the bracket node of a critical section structure, then $f_e = f_\top$.

    Signaling node propagation of non-bracket head nodes along sync hyperedges is only to other signaling nodes. This is done for efficiency reasons; if the deadlock contains an accept node, then the information found by propagating signaling nodes around the deadlock cycle is redundant. Section 6.5.9 explains why this condition must be relaxed for bracket nodes of critical section structures.

- If a_i cannot be a head node, or is in the task of h and $a_i \neq h$, then $f_e = f_\top$.
- If a_i is not co-executable with h or is in $CHT(h)$, then then $f_e = f_\top$.
- If $s \in Task(h)$ and a_i and h are bracket nodes of the same critical section structure, then then $f_e = f_\top$.

  This step prevents h from propagating from its own task to bracket nodes of the same critical section structure, eliminating cycles which contain only the bracket nodes of the same critical section structure as head nodes.
- If none of the above is true, then f_e is a conditional identity function such that:
  * Y_e is the set containing the signal types of a_i and h, and all of their unguarded sisters;
  * For each $y \in Y_e$, $V_e[y]$ is the minimum scope depth of all unguarded sisters of a_i or h of type y, with respect to signal type y. a_i and h are also included in this minimum if either is of signal type y.

Here, propagation is allowed only if we cannot guarantee that a_i or h can rendezvous with some signaling head node on the deadlock path.

- If e is an HCLG edge (a_o, s_d) representing mode (a) or (c) propagation, and s_d represents node s in the sync hypergraph:
  - If s cannot be a head node, or is in the task of h and $s \neq h$, then $f_e = f_\top$.
  - If s is not co-executable with h or is in $CHT(h)$, then then $f_e = f_\top$.
  - If $a_o \in Task(h)$ and s and h are bracket nodes of the same critical section structure, then then $f_e = f_\top$.
  - If none of the above is true, then f_e is a conditional increment function such that:
    * Y_e is the set containing the signal types of h and all of its unguarded sisters;
    * y_e is the signal type of node s;
    * For each $y \in Y_e, y \neq y_e$, $V_e[y]$ is the minimum scope depth of all unguarded sisters of h of type y, with respect to signal type y. The scope depth of h with respect to signal type y is also included in this minimum if h is of type y.
    * $V_e[y_e]$ is the minimum scope depth of all unguarded sisters of h of type y_e, with respect to signal type y_e, *minus one*. The scope depth of h is also included in this

minimum if h is of signal type y_e. If h is not of type y_e and has no unguarded sisters of type y_e, then $V_e[y_e]$ may be set arbitrarily.

Here, we allow propagation only if we cannot guarantee that h can rendezvous with s or some other signaling node along the deadlock path. If propagation is allowed, we increase the count of signaling head nodes of s's type by one, since s is added to the path as a head node.

7.4 Time for the total reachability algorithm.

To solve the *RHead* problem, one *SigCnt* problem must be solved for each potential head node. Thus, the worst-case total time for solution of the entire reachability problem is $\mathcal{O}(|Sigs||N_H||N|^3 \log(|N|))$ using the worklist algorithm, and $\mathcal{O}(|Sigs||N_H||N|^3)$ using Tarjan's path expression algorithm.

It is not likely that we could economize by re-using the same path expression for different assumed head nodes h in Tarjan's algorithm. We would wish to make h the origin node of the program, and use the node splitting approach of Section 7.2. This would require constructing a separate path expression for each h.

8 Signaling node count/accept scope depth lattice framework.

We have found by experiment that it is highly beneficial to maintain, for each hypothesized head node h, an upper bound with respect to each signal type y on the scope depth of all accept-in head nodes of type y on the paths from h to other program points. This helps us to use constraint 2 to eliminate some paths which could include a rendezvous between head nodes other than h.

We extend the *SigCnt* lattice framework to maintain upper bounds on scope depth for accept-in head nodes (and their unguarded sisters) for deadlock paths, as well as lower bounds on the number of signaling nodes of each signal type. If the lower bound on the number of signaling nodes of type y is at least as great as the upper bound on scope depth for signals of type y, then the deadlock path must include a pair of head nodes that can rendezvous. Such a path violates constraint 2, and is thus not part of a deadlock cycle.

A *signaling node/accept scope depth count summary SACnt* is either:

- A pair of vectors (S, A), where S is a signaling node vector as defined in Section 7, and A is a vector of integers of length $|Sigs|$, such that the y'th element of the vector lies between 0 and some constant j_y, or;
- The distinguished top element $SACnt_\top$.

For a node n, $SACnt$ represents both a lower bound on the number of signaling head nodes of each signal type, and an upper bound on the scope depth of any accept head node of each signal type, encountered on valid paths from a particular head node h to n. j_y above is the greater of:

- The total number of signaling nodes of type y in the sync graph (i.e., k_y), plus one, or;
- The maximum scope depth of any accept node of type y in the sync hypergraph with respect to signal type y, plus one.

No deadlock path can include either j_y signaling nodes or an accept node of type y with a scope depth of j_y with respect to signal type y. We can thus use $A[y] = j_y$ as a summary for paths which contain no accept head nodes of type y, i.e., in the bottom element of L_{SACnt}.

$SACnt_i \sqsubseteq SACnt_j$ if either:

- $SACnt_j = SACnt_\top$, or;
- $SACnt_i = (S_i, A_i) \neq SACnt_\top$ and $SACnt_j = (S_j, A_j) \neq SACnt_\top$ and $S_i[y] \leq S_j[y]$ and $A_i[y] \geq A_j[y]$ for all $y \in Sigs$.

The meet operation is thus an element-wise minimum over the S vectors and an element-wise maximum over the A vectors. The lattice subspace L_{SACnt} is:

$$L_{SACnt} \subset \mathcal{N}^{2|Sigs|} \cup SACnt_{\top}$$

$$SACnt_{\top} \in L_{SACnt}$$

where $\mathcal{N}$ represents the set of nonnegative integers. For convenience, let $SACnt_{\bot}$ denote a pair whose S element is a vector of all zero elements of length $|Sigs|$, and whose A element is a vector of length $|Sigs|$ such that $A[y] = j_y$ for $y \in Sigs$.

Lemma 72

$$\sum_{y \in Sigs} j_y \leq |N| + |Sigs|.$$

Proof: For each $y \in Sigs$, j_y is one more than either the number of signaling nodes of type y or the maximum scope depth of any accept node of type y with respect to y, whichever is greater. If the former is greater, then j_y cannot be greater than one plus the number of signaling nodes of type y in the sync hypergraph.

Suppose the latter is greater and that node a is an accept-in (or accept-out) node with signal type y and scope depth $j_y - 1$. Then a must be nested in $j_y - 1$ `accept y do...` constructs to achieve this scope depth. The value of j_y is maximized when a is nested inside *all* other `accept y do...` constructs in the program. For each such construct, there is one accept-in node (and one accept-out node) of type y in the sync hypergraph. Then j_y cannot be greater than the number of accept-in nodes of type y in the sync hypergraph.

Therefore, j_y must be at most one more than the number of nodes of type y in the sync hypergraph, and

$$\sum_{y \in Sigs} j_y \leq |N| + |Sigs|.\Box$$

Lemma 73 *The height of the L_{SACnt} lattice is at most $2|N| + |Sigs| + 2$.*

Proof: From Lemma 64, the elements of S can be increased a maximum of $|N|$ times before reaching the maximum value other than $SACnt_{\top}$. From Lemma 72, the elements of A can be decreased at most $|N| + |Sigs|$ times before reaching their minimum value other than $SACnt_{\top}$. Therefore, $SACnt_{\bot}$ can be increased at most $2|N| + |Sigs| + 2$ times before reaching $SigCnt_{\top}$. $\Box$

8.1 Function class for *SACnt* framework.

The function class $F_{SACnt} = \{f : L_{SACnt} \rightarrow L_{SACnt}\}$ is, as before, the transitive closure of the set of possible *edge functions* under meet and function composition. A particular edge e will be represented by one of the following function types:

- The *identity function* $f_\iota(x) = x$.
- The *top function* $f_\top(x) = SACnt_\top$.
- A *constant function* $f_e(x) = k_e$ where $k_e \in L_{SACnt}$ is a constant. Note that $f_\bot$ and $f_\top$ are constant functions.
- An *accept head node function* $f_e((S_x, A_x))$ such that, if:
  - $S_x[y] < A_x[y]$ for $y \in Y$;
  - $S_x[y] \leq V_e[y]$ for some constant vector of integers V_e and for $y \in Y_e$ where Y_e is a constant set of signal types; and,
  - $A_x[y] = j_y$ for $y \in Z_e$ where Z_e is a constant set of signal types;

  then $f_e(x) = (S_x, A')$ where:
  - $A'[y] = A_x[y]$ for $y \in Sigs, y \notin Z_e$, and;
  - $A'[y] = A_e[y]$ for $y \in Z_e$, where A_e is a constant vector such that $0 \leq A_e[y] < j_y$ for $y \in Sigs$.

  Otherwise $f_e(x) = SACnt_\top$.
- A *signaling head node function* $f_e((S_x, A_x))$ such that, if:
  - $S_x[y] < A_x[y]$ for $y \in Y$; and,
  - $\min(k_{y_e}, S_x[y_e] + 1) < A_x[y_e]$ where y_e is a constant signal type and k_{y_e} is a constant integer;

  then $f_e((S_x, A_x)) = (S', A_x)$ where:
  - $S'[y] = S_x[y]$ where $y \in Sigs, y \neq y_e$;
  - $S'[y_e] = \min(S_x[y_e] + 1, k_{y_e})$.

Otherwise $f_e(x) = SACnt_\top$.

Lemma 74 *The set of edge functions is monotonic. Furthermore, with the exception of the constant functions, the set of edge functions is nondecreasing.*

Proof: The identity and top functions are clearly both monotonic and nondecreasing. The constant functions are, likewise, monotonic.

In the case of the accept head node functions $f_e((S_x, A_x)) = (S_x, A')$, the only elements of A' which are different from those of A_x are those for signal types y where $A_x[y] = j_y$, the maximum permissible value. Thus, $A'[y] \leq A_x[y]$ for $y \in Sigs$, and the accept head node functions are nondecreasing (since the lattice ordering $\sqsubseteq$ on A is element-wise $\geq$). A similar argument holds for the signaling head node functions.

To show monotonicity for the accept head node functions, suppose that $(S_x, A_x) \sqsubseteq (S_z, A_z)$. Thus, for each signal type y,

$$S_x[y] \leq S_z[y]$$

and

$$A_x[y] \geq A_z[y].$$

Suppose that

$$f_e((S_x, A_x)) = (S_x, A'_x)$$

and

$$f_e((S_z, A_z)) = (S_z, A'_z).$$

Thus, neither $f_e((S_x, A_x))$ nor $f_e((S_z, A_z))$ equals $SACnt_\top$. This can only happen if

$$\forall y \in Z_e : A_x[y] = A_z[y] = j_y.$$

Therefore,

$$\forall y \in Z_e : A'_x[y] = A'_z[y] = j_e[y],$$

and

$$\forall y \in Sigs \overset{\text{such that}}{} y \notin Z_e : A'_x[y] = A_x[y] \geq A_z[y] = A'_z[y].$$

In this case,

$$f_e((S_x, A_x)) \sqsubseteq f_e((S_z, A_z)).$$

In the case where $f_e((S_z, A_z)) = SACnt_\top$, monotonicity follows automatically. Suppose, though, that we have $(S_x, A_x) \sqsubseteq (S_z, A_z)$, $f_e((S_x, A_x)) = SACnt_\top$, and $f_e((S_z, A_z)) = (S_z, A'_z) \neq SACnt_\top$. Since

$$\forall y \in Z_e : A_x[y] \geq A_z[y] = j_y$$

this must be because there is some $y \notin Z_e$ such that $S_x[y] > V_e[y]$ for some $y \in Y_e$. But

$$S_x[y] \leq S_z[y],$$

so

$$S_z[y] > V_e[y]$$

also, in which case

$$f_e((S_z, A_z)) = SACnt_\top.$$

Thus, the last case cannot exist and f_e is monotonic.

The monotonicity arguments for signaling head node functions are nearly identical to those for accept head node functions, except for the case where $(S_x, A_x) \sqsubseteq (S_z, A_z)$, $f_e((S_x, A_x)) = SACnt_\top$, and $f_e((S_z, A_z)) = (S_z, A'_z)$. Since

$$\forall y \in Z_e : A_x[y] \geq A_z[y],$$

$f_e((S_x, A_x)) = SACnt_\top$ either because

$$\exists y \in Y_e : y \notin Z_e \wedge S_x[y] > A_x[y],$$

or because

$$\min(k_{y_e}, S_x[y_e] + 1) > A_x[y_e].$$

But, for all $y \in Sigs$,

$$S_x[y] \leq S_z[y]$$

and

$$A_x[y] \geq A_z[y],$$

so we must have the same situation in (S_z, A_z), which would imply that $f_e((S_z, A_z)) = SACnt_\top$. So the situation described cannot exist, and the signaling head node functions are monotonic. □

Lemma 75 *F_{SACnt} includes functions which are not distributive.*

Proof: Consider an accept head node function f_e such that $Z_e = \{z_1, z_2\}$, and two summaries $a = (S, A_a), b = (S, A_b) \in L_{SACnt}$ such that, for all $y \in Sigs$, $S[y] = 0$, and:

$$\begin{aligned} A_a[z_1] &< j_{z_1} \\ A_a[z_2] &= j_{z_2} \\ A_b[z_1] &= j_{z_1} \\ A_b[z_2] &< j_{z_2}. \end{aligned}$$

Thus,

$$f_e(a) = f_e(b) = SACnt_\top = f_e(a) \sqcap f_e(b),$$

while

$$f_e(a \sqcap b) \sqsubseteq SACnt_\top.$$

Thus, f_e is not distributive. □

As before, we cannot use the time complexity results of Lemma 11 for the *SACnt* problem because the members of the function space are not distributive.

Lemma 76 *The meet operation in L_{SACnt} requires $\mathcal{O}(|Sigs|)$ time to compute.*

Proof: From the definition of the meet operation and the L_{SACnt} lattice. □

Lemma 77 *An edge function application requires $\mathcal{O}(|Sigs|)$ time to compute.*

Proof: The accept head node and signaling head node edge functions require a scan of some elements of their argument against corresponding elements of a constant vector. This takes $\mathcal{O}(|Sigs|)$ time. Creating the result also requires $\mathcal{O}(|Sigs|)$ time for initialization. □

Let F^-_{SACnt} be the transitive closure of the set of edge functions (excluding the constant functions) under meet and function composition.

Lemma 78 *The functions of F^-_{SACnt} are monotonic, nondecreasing, and 1-semibounded.*

Proof: Identical to that of Lemma 66. □

As in the case of the *SigCnt* problems, the constant edge functions occur only on control edges leaving a hypothesized head node. We can therefore split each hypothesized head node h into h_d and h_s, initializing h_s to the proper constant value and all other nodes to $SACnt_\top$ before iteration. The F_{SACnt} problem will thus converge in the same time as a 1-semibounded problem which uses only the F^-_{SACnt} functions on a graph that has at most one more node and $|N|$ more edges than the HCLG.

Lemma 79 *A worklist iterative algorithm using a lattice framework* $D = (G, L_{SACnt}, F^{-}_{SACnt}, M)$ *will converge from an initialization of all elements at* $SACnt_{\top}$ *in time* $\mathcal{O}(|Sigs||N|^3 \log(|N|) + |Sigs|^2|N|^2 \log(|N|))$ *in the worst case.*

Proof: From Lemmas 9 and 73, the worklist algorithm will require no more than $(2|N| + |Sigs| + 2)|N|^2 \lceil \log_2(|N|) \rceil$ meet operations, $(2|N| + |Sigs| + 2)|N|^2$ edge function evaluations, and $\mathcal{O}(|N|^2 \log(|N|))$ time to maintain the worklist in the worst case. From Lemmas 68 and 69, this will result in a worst case time of $\mathcal{O}(|Sigs||N|^3 \log(|N|) + |Sigs|^2|N|^2 \log(|N|))$. □

Lemma 80 *The total time for solution of* $D = (G, L_{SACnt}, F^{-}_{SACnt}, M)$ *using Tarjan's path expression algorithm is* $\mathcal{O}(|N|^2|Sigs|)$ *in the worst case.*

Proof: Direct from Lemmas 10, 76, 77 and 78. □

8.2 Mapping function in *SACnt* framework.

To represent an edge $e = (m, n)$ in HCLG G in this framework for a hypothesized head node h, select $f_e \in F_{SACnt}$ such that:

- If e represents a control edge (m', n') in the sync hypergraph:
  - If $m' \neq h$, then $f_e = f_\iota$.

    This allows unrestricted propagation along control edges whose tail is not an assumed head node.
  - If $m' = h$, and h is a signaling node of type y_h, then $f_e = (S_e, A_e)$ where:
    - $S_e[y_h] = 1$ and $S_e[y] = 0$ for $y \in Sigs, y \neq y_h$.
    - $A_e[y] = j_y$ for $y \in Sigs$.

    This starts propagation of *SACnt* records for h, with the number of signaling nodes of type y_h initialized to 1, all other signaling nodes to 0, and the maximum scope depth of encountered accept head nodes effectively infinite. (Actually, since h propagates only to other signaling head nodes in this case, we need not be too concerned about the initial value, as long as the initial value does not *a priori* indicate that a rendezvous may happen.)
  - If $m' = h$, and h is an accept-in node of type y_e, then let S_h be $\{h\}$ unioned with the set of unguarded sisters of h. Let Y_e be the set of signal types of the elements of S_h, and let d_y be the minimum scope depth of any element of S_h which has type y (with respect to y). The edge function f_e is a constant function returning (S_e, A_e), where:

* $S_e[y] = 0$ for $y \in Sigs$.
* $A_e[y] = d_y$ for $y \in Y_e$, $A_e[y] = j_y$ for $y \in Sigs, y \notin Y_e$.

This starts propagation of $SACnt$ records for h, with the number of signaling nodes of all types initialized to 0, the maximum scope depth of h and its unguarded sisters initialized to the minimum necessary to guarantee that some sister can rendezvous, and the maximum scope depth of all other accept head nodes effectively infinite.

- If e is an edge (s_e, s_d) in the HCLG for some signaling node s in the sync hypergraph, then $f_e = f_\iota$.

  This allows unrestricted propagation from engage (s_e) to disengage (s_d) signaling nodes in the HCLG.

- If e is an HCLG edge (s_e, a_i) representing mode (b) propagation, then the edge function f_e is a function such that:
  - If a_i cannot be a head node, or is in the task of h and $a_i \neq h$, then $f_e = f_\top$.
  - If a_i is not co-executable with h or is in $CHT(h)$, then then $f_e = f_\top$.
  - If h is a signaling node, and is not the bracket node of a critical section structure, then $f_e = f_\top$.

    As in the $SigCnt$ lattice, hypothesized signaling nodes propagate only to other signaling head nodes along sync hyperedges.
  - If $s \in Task(h)$ and a_i and h are bracket nodes of the same critical section structure, then then $f_e = f_\top$.
  - If none of the above is true, then f_e is an accept head node function such that:
    * Y_e is the set containing the signal types of a_i and all of its unguarded sisters;
    * For each $y \in Y_e$, $V_e[y]$ is the minimum scope depth of all unguarded sisters of a_i of type y, with respect to signal type y. a_i is also included in this minimum if it is of signal type y.

    This allows mode (b) propagation only along paths where a_i (or one of its sisters) is not guaranteed to rendezvous with a signaling head node already in the path, and disallows propagation along some paths which enter a_i's task more than once (i.e., those which include more than one accept-in head node or unguarded sister with the same signal type).

- If e is an HCLG edge (a_o, s_d) representing mode (a) or (c) propagation, and s_d represents node s in the sync hypergraph:
  - If s cannot be a head node, or is in the task of h and $s \neq h$, then $f_e = f_\top$.
  - If s is not co-executable with h or is in $CHT(h)$, then $f_e = f_\top$.
  - If $a_o \in Task(h)$ and s and h are bracket nodes of the same critical section structure, then then $f_e = f_\top$.
  - If none of the above is true, then f_e is a signaling head node function such that y_e is the signal type of node s.

    This assignment allows summaries to propagate to a signaling head node only when we cannot guarantee that some head nodes in the cycle will be able to rendezvous.

9 Total time for the algorithm.

Tables 6 and 7 summarize the timing analysis for the deadlock detection algorithm using the worklist and Tarjan's algorithms respectively, with both *SigCnt* and *SACnt* lattice frameworks. Note that:

- $|N| = \mathcal{O}(|N_{CHT}|)$;
- $|N_H| = \mathcal{O}(|N|)$;
- $|E_C| = \mathcal{O}(|N|^2)$;
- $|E_S| = \mathcal{O}(|N|^2)$.

Operation	*SigCnt* algorithm	*SACnt* algorithm
Parse	$\mathcal{O}(\|N_{CHT}\| + \|E_C\| + \|E_S\|)$	$\mathcal{O}(\|N_{CHT}\| + \|E_C\| + \|E_S\|)$
CHT sets	$\mathcal{O}(\|Tasks\|\|N_{CHT}\|^5)$	$\mathcal{O}(\|Tasks\|\|N_{CHT}\|^5)$
Loop unroll	$\mathcal{O}(\|N_{CHT}\| + \|E_C\| + \|E_S\|)$	$\mathcal{O}(\|N_{CHT}\| + \|E_C\| + \|E_S\|)$
Reachability	$\mathcal{O}(\|Sigs\|\|N_H\|\|N\|^3 \log(\|N\|))$	$\mathcal{O}(\|Sigs\|\|N_H\|\|N\|^3 \log(\|N\|)$ $+\|Sigs\|^2\|N_H\|\|N\|^2 \log(\|N\|))$
Total	$\mathcal{O}(\|Tasks\|\|N\|^5$ $+\|Sigs\|\|N_H\|\|N\|^3 \log(\|N\|))$	$\mathcal{O}(\|Tasks\|\|N\|^5$ $+\|Sigs\|\|N_H\|\|N\|^3 \log(\|N\|)$ $+\|Sigs\|^2\|N_H\|\|N\|^2 \log(\|N\|))$

Table 6: Summary of worst-case timing analysis for static deadlock detection using the worklist algorithm.

Operation	*SigCnt* algorithm	*SACnt* algorithm
Parse	$\mathcal{O}(\|N_{CHT}\| + \|E_C\| + \|E_S\|)$	$\mathcal{O}(\|N_{CHT}\| + \|E_C\| + \|E_S\|)$
CHT sets	$\mathcal{O}(\|Tasks\|\|N_{CHT}\|^5)$	$\mathcal{O}(\|Tasks\|\|N_{CHT}\|^5)$
Loop unroll	$\mathcal{O}(\|N_{CHT}\| + \|E_C\| + \|E_S\|)$	$\mathcal{O}(\|N_{CHT}\| + \|E_C\| + \|E_S\|)$
Reachability	$\mathcal{O}(\|Sigs\|\|N_H\|\|N\|^2)$	$\mathcal{O}(\|Sigs\|\|N_H\|\|N\|^2)$
Total	$\mathcal{O}(\|Tasks\|\|N\|^5)$	$\mathcal{O}(\|Tasks\|\|N\|^5)$

Table 7: Summary of worst-case timing analysis for static deadlock detection using Tarjan's path expression algorithm.

A Notation.

The following notation is used in this report.

Signal types. A signal type y is a pair (t_y, n_y) where t_y is the task accepting the signal and n_y is the name of the signal.

Execution waves. In noting execution waves (or parts of waves), $E[t]$ refers to the node of task t which is in execution wave E.

Instances and iterations. If n is a node, n_i represents the instance of a occurring in the i'th iteration of the loop most immediately containing a. Likewise, $L_i(a)$ denotes the instances of all statements in the i'th iteration of that loop.

Reachability, *B4*, and *CHT* form sync hypergraphs. Unless otherwise specified, "sync hypergraph" refers to the reachability form.

Arrays and sets. Commonly, we will refer to a map from objects to sets of objects as an "array." For instance, the *CHT* array is a map from each node to a set of nodes. Such a map can be easily implemented as a Boolean array, hence the name.

Symbols.

- $AInSameType(n)$ is the set of accept-in nodes of the same type as n, if n is a signaling node. If n is not a signaling node, $AInSameType(n)$ is empty.
- $Accept?(E, y)$ is a predicate which yields true iff the set (or execution wave) E contains an accept-in node of signal type y.
- $CallB(c, r)$ is the call body defined by the node pair (c, r), i.e., the set of nodes in the control flow interval (c, r), minus $\{c\}$.
- $CHT(n)$ is a set of nodes that never execute at the same time as n.
- $Completors(x)$ is the set of the most immediate control ancestors of x which must complete a rendezvous before x executes.

- *CompleteRend*(x) is the set of nodes which may complete a rendezvous with x. If x is a signaling node, *CompleteRend*(x) is the set of accept-out nodes of the same type; if x is an accept-out node, *CompleteRend*(x) is the set of signaling nodes of the same type. Otherwise, *CompleteRend*(x) is empty.

- *CPreds*(x) is the set of control predecessors of node x.

- *CReach*(x) is the set of nodes n such that there is a control path from n to x in the *B4* form sync hypergraph.

- $CSB(n, x)$ is the critical section body defined by the node pair (n, x), i.e., the set of nodes in the control flow interval (n, x), minus $\{n\}$.

- $DOM(n)$ is the set of control dominators of node n.

- E_C is the set of control edges in the sync hypergraph.

- $E_{Completors}$ is the set of completor edges derived from the *CHT* form sync hypergraph.

- E_S is the set of sync hyperedges in the sync hypergraph.

- $E_{S_{CHT}}$ is the set of sync hyperedges in the *CHT* form sync hypergraph.

- I is the identity array.

- *InClause*(n) is, for each accept-in node n, the set of nodes contained within the do clause of n (including n's accept-out node). If n is not an accept-in node, then *InClause*(n) is empty.

- $L(x)$ is the loop body most immediately containing node x in the *CHT* form sync hypergraph.

- $LE(x)$ is the end node of the loop body most immediately containing node x in the *CHT* form sync hypergraph.

- $LS(x)$ is the begin node of the loop body most immediately containing node x in the *CHT* form sync hypergraph.

- *Loops* is the set of loop bodies in the program. $|Loops| < |N_{CHT}|$.

- N is the set of nodes in the reach form sync hypergraph.

- N_{CHT} is the set of nodes in the *CHT*-form sync hypergraph.

- N_H is the set of potential head nodes in the sync hypergraph. $N_H \subset N$.
- $NodeType(n)$ is the node type of n (i.e., signaling, accept-in, etc.)
- ONE is an array of all ones.
- $Partners(n)$ is a set of nodes, one of which must appear on the execution wave whenever n starts.
- $Pins(n)$ contains all nodes if n pins its partners, $\emptyset$ if not.
- $Partners_{Pinned}(n)$ contains the set of partners n pins, or $\emptyset$ if n does not pin its partners.
- $POSTDOM(n)$ is the set of control postdominators of node n.
- $RHead(n)$ is the set of hypothesized head nodes h such that there may be a valid deadlock path from h to n.
- $SACnt_h(n)$ is a lower-bound count of the number of signaling head nodes of each signal type, and an upper-bound count on the scope depth of accept-in head nodes of each signal type, on all deadlock paths from h to n.
- $SigCnt_h(n)$ is a lower-bound count of the number of signaling head nodes of each signal type, on all deadlock paths from h to n.
- $Sigs$ is the set of signal names (of the form `task.entry`) in the program.
- $Scope(n, y)$ is the scope depth of node n with respect to signal type y.
- $Scope_{set}(E, y)$ is the sum, for all nodes n in the set (or execution wave) E, of the scope depth of n with respect to signal type y.
- $SigN_{set}(E, y)$ is the number of signaling nodes of signal type y in the set (or execution wave) E.
- $Sisters(n)$ is the reflexive set of sisters of n, i.e., accept-in nodes in the same conditional entry construct, if n is an accept-in node. Otherwise, $Sisters(n)$ is empty.
- $Starters(n)$ is the set of most immediate control descendants of n which can start a rendezvous.

- *StartRendSig*(n) is the set of signaling nodes that can start a rendezvous with n If n is an accept-in node, *StartRendSig*(n) is the set of signaling nodes of the same signal type as n. If n is not an accept-in node, then *StartRendSig*(n) is empty.

- *StrictInt*($G, \mathcal{P}$) is the set of strict intervals on graph G as defined by the partition $\mathcal{P}$ of the nodes of G. See Section 6.4 for a more complete definition.

- *SigType*(n) is the signal type of n.

- *TakeOff*(n) is the set of nodes which can take n off the execution wave by starting or completing a rendezvous.

- *Task*(n) is the task to which node n belongs.

- *TaskFrontNodes*(t) is the set of nodes in task t which may be placed on the execution wave at the start of the program.

- *UngSis?*(E, y) is a predicate which yields true iff the set (or execution wave) E contains an accept-in node of type y, or which has an unguarded sister of type y.

- *Waits*(E, x, y) is a relation such that, in execution wave E, node x cannot complete execution, but x could complete execution if y were on E, and y is a control successor of a node now on E. (I.e., "x waits on y in E.")

- *Widen*(CHT, S, T) is an array which is a superset of the CHT array. The refinement of CHT is made under the assumption that, for any node n, $S(n)$ is a set of nodes, such that some member of $S(n)$ must happen together with n (if $S(n)$ is not empty).

  T is a set of nodes such that, for each node n in the program, n must execute together with members of T. Typically, T will be either the nodes of a single task or the total set of nodes in the program.

References

[AHU74] Aho, A. V., Hopcroft, J. E., and Ullman, J. D.

The design and analysis of computer algorithms.

Addison-Wesley, Reading, Massachussetts, 1974. ISBN 0-201-00029-6.

[ANSI83] American National Standards Institute.

"ANSI/MIL-STD 1815A (1983) reference manual for the Ada programming language."

United States Government Printing Office, Washington DC, 1983.

[CS88] Callahan, D. and Subhlok, J.

"Static analysis of low-level synchronization."

SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, 1988, 100-111.

[DS91] Duesterwald, E. and Soffa, M. L.

"Concurrency analysis in the presence of procedures using a data-flow framework."

ACM Symposium on Testing, Analysis and Verification (TAV4), Vancouver, October 1991, 36-48.

[Dues91] Duesterwald, E.

"Static concurrency analysis in the presence of procedures."

Technical Report 91-6, Department of Computer Science, University of Pittsburgh, March 1991.

[Floy62] Floyd, R.

"Algorithm 97: shortest path."

Communications of the ACM, **5:6**, June 1962, 345.

[Hec77] Hecht, M. S.

"Flow analysis of computer programs."

Elsevier North-Holland, New York, 1977. ISBN 0-444-00210-3.

[KU76] Kam, J. B. and Ullman, J. D.

"Global data flow analysis and iterative algorithms."

Journal of the ACM, **23:1**, January 1976, 158-171.

[LC91] Long, D. L. and Clarke, L. A.

"Data flow analysis of concurrent systems that use the rendezvous model of synchronization."

In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, October

1991, 21-35.

[LT79] Lengauer, T. and Tarjan, R. E.

"A fast algorithm for finding dominators in a flowgraph."

ACM Trans. Prog. Lang. and Syst., **1:1**, July 1979, 121-141.

[MR90] Masticola, S. P. and Ryder, B. G.

"Static infinite wait anomaly detection in polynomial time."

LCSR-TR-141, Laboratory for Computer Science Research, Rutgers University, 1990.

[MR91a] Marlowe, T. J. and Ryder, B. G.

"Properties of data flow frameworks: a unified model."

Acta Informatica **28:2**, 1991, 121-164.

[MR91b] Masticola, S. P. and Ryder, B. G.

"A model of Ada programs for static deadlock detection in polynomial time."

1991 ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, May 1991, 91-102.

Also *ACM SIGPLAN Notices* **26:12**, December 1991, 97-107.

[Saxe77] Saxena, A.

"Static detection of deadlocks."

CU-CS-122-77, Dept. of Computer Science, University of Colorado at Boulder, 1977.

[Tarj81a] Tarjan, R. E.

"A unified approach to path problems."

Journal of the Association for Computing Machinery, **28:3**, July 1981, 577-593.

[Tarj81b] Tarjan, R. E.

"Fast algorithms for solving path problems."

Journal of the Association for Computing Machinery, **28:3**, July 1981, 594-614.

[Zade84] Zadeck, F. K.

"Incremental data flow analysis in a structured program editor."

ACM SIGPLAN '84 Symposium on Compiler Construction.

In *ACM SIGPLAN Notices*, **19:6**, June 1984, 132-143.