Non-concurrency Analysis.

Stephen P. Masticola (masticol@cs.rutgers.edu)
Barbara G. Ryder (ryder@cs.rutgers.edu)

Department of Computer Science
Rutgers University, New Brunswick, NJ 08903

Abstract. Non-concurrency analysis is a set of techniques for statically identifying pairs (or sets) of statements in a concurrent program which can never happen together. This information aids programmers in debugging and manually optimizing programs, improves the precision of data flow analysis, enables optimized translation of rendezvous, facilitates dead code elimination and other automatic optimizations, and allows anomaly detection in explicitly parallel programs. We present a framework for non-concurrency analysis, capable of incorporating previous analysis algorithms [CS88, DS91] and improving upon them. We show general theoretical results which are useful in estimating non-concurrency, and examples of non-concurrency analysis frameworks for two synchronization primitives: the Ada rendezvous and binary semaphores. Both of these frameworks have a low-order polynomial bound on worst-case solution time. We provide experimental evidence that static non-concurrency analysis of Ada programs can be accomplished in a reasonable time, and is generally quite accurate. Our framework, and the set of refinement components we develop, also exhibits dramatic accuracy improvements over [DS91], when the latter is used as a stand-alone algorithm, as demonstrated by our experiments.

1 Introduction.

Parallel and concurrent programs are often explicitly divided into tasks by the programmer. The tasks may synchronize with each other at various points, to request services or information from each other, or to prevent races. If the synchronization structure is complex, it may be difficult to determine *a priori* which parts of the program may, or may not, execute in parallel. Taylor showed that determining this exactly is NP-hard [Tayl83b]; our objective, rather, is to obtain a close underestimate of non-concurrency, in a time polynomial in the number of program statements.

1.1 Applications.

Information about guaranteed non-concurrency has a variety of uses in compilation and debugging. Specifically, it is useful for program understanding tools, data flow analysis [RS90, LC91], rendezvous optimization, dead code elimination, and synchronization anomaly detection.

Debuggers and program understanding tools. While debugging concurrent programs, or tuning their performance, it may be valuable to know that some pair of code segments can never happen together. For example, if the code segments that access a shared resource are non-concurrent, the programmer can safely delete any locks that are protecting the resource.

We envision a debugger that includes a structured source code editor. On the user's command, the editor would highlight all code that might be concurrent with the code at the current cursor position. Alternately, the debugger could answer queries about whether two statements may be concurrent. Thus, non-concurrency analysis may become a valuable addition to program understanding tools.

Data flow analysis. If tasks pass data to each other during rendezvous, non-concurrency information can eliminate some infeasible def-use pairs. Statements which are non-concurrent cannot rendezvous with each other; no definitions can reach uses via rendezvous which can never occur. Similar examples exist for other classic data flow problems.

Optimizing rendezvous. Rendezvous can be optimized by choosing the proper implementation for a particular statement instance. In a simple form of rendezvous synchronization, signaling tasks call an accepting task; both wait until the call is completed. Each signaling task may call the accepting task from more than one call point; the accepting task may also accept the calls at any of several entry points.

Entries might be implemented in two alternate ways: by polled flags or by call queues. Polling will be more efficient when few call points can execute simultaneously with the entry; beyond some threshold, the time to poll will be greater than the overhead for queue management. Non-concurrency analysis might determine that this threshold has not been reached for a particular entry, even though the total number of call points is greater than the threshold. If so, we can gain speed by using polling.

Dead code elimination. Dead code can also be detected using non-concurrency information. If some statement in a task can never happen together with *any* other statement in the program (including the beginning and end of the program), then it can never happen at all. The code may thus be eliminated from the program. If the code is dead because some `accept` alternative of an entry can never rendezvous, then the entry may also be optimized by removing the alternative.

In some cases, the code may be dead because some predecessor in the same task *must* starve or deadlock; here, the dead code is only a symptom of a more serious anomaly.

Synchronization anomalies. More commonly, non-concurrency analysis has been applied to static detection of synchronization anomalies, including races and deadlocks.

A *race* occurs whenever two tasks may simultaneously access the same shared object (variable or resource), and if at least one of those accesses changes the state of the resource (e.g., advances a stream or writes a variable).[1] If we can prove that two accesses are non-concurrent, then there can be no races between them. Callahan and Subhlok [CS88] apply this methodology to post/wait synchronization in programs without loops.[2]

A *deadlock* occurs when several tasks mutually wait on each other to advance. The statements at which these tasks wait are the *head* statements of the deadlock. Given a proposed head statement s, we can often decide that all proposed deadlocks which might contain s must also contain statements that are non-concurrent with s, and thus do not represent actual deadlocks. This allows us to prune s as a possible member of any deadlock [MR90, MR91b].

[1] To guarantee that the program produces determinate results, we must usually also guarantee that conflicting statement instances are *totally ordered*, rather than just non-concurrent.

[2] Their later work [CKS90] incorporated loops, but the methodology they used there is only tangentially related to non-concurrency analysis.

1.2 Previous work.

Callahan and Subhlok [CS88] presented a method for polynomial-time static non-concurrency analysis in programs which use post/wait synchronization. Later, Duesterwald and Soffa [DS91] used similar methods in the rendezvous model. Our technique can use these as components to refine non-concurrency information; we adapted [DS91] as part of our empirical model. We show experimental evidence that for Ada programs, an algorithm based only on [DS91] would miss much non-concurrency information that a broader range of refinements can extract.

Bristow, et. al. [BDER79] presented some early ideas related to non-concurrency analysis; we describe the relation of their work to ours further in Sections 2 and 4.

2 Framework for non-concurrency analysis.

Figure 1 shows the general framework used in non-concurrency analysis. The framework must be specialized for a particular tasking model and set of synchronization primitives.

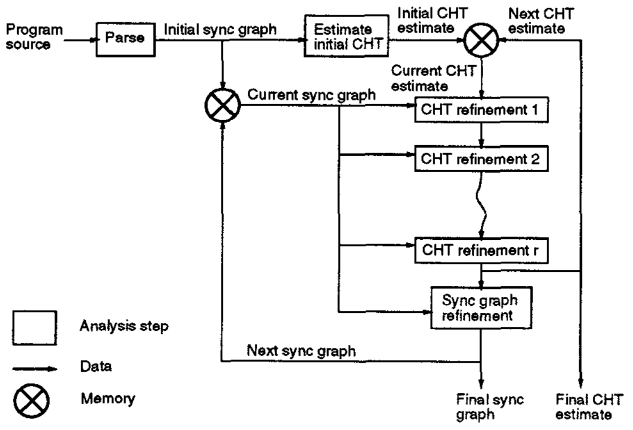

Figure 1: General framework of iterative non-concurrency analysis.

The program is parsed, and a *sync graph* representation of the program's synchronization behavior is generated. The sync graph represents only synchronizations and control flow between synchronization statements, and omits instruction timing and priorities, if they exist in the source language. Non-concurrency results are thus conservative with respect this information; knowledge of timing and priorities could add to non-concurrency information, but could never invalidate a non-concurrency relationship between statements.

Each node in the sync graph represents a synchronization statement in the program. Directed *control edges* are placed from node m to n if there is a control path from m to n that contains no tasking statements other than m and n. *Sync edges* represent synchronizations, and are placed between all nodes that may synchronize with each other. Sync edges may be directed, undirected, or multi-ended, depending on the synchronization model.

The non-concurrency relation is labeled CHT, for *can't happen together*. For each node n in the sync graph, $CHT(n)$ is a set of nodes which have no instances that can ever occur simultaneously with any instance of n, in any execution of the program. Note that $CHT(n)$ need not be the complete set of such nodes; the latter is denoted $CHT_{perf}(n)$. We obtain an initial estimate $CHT_0(n)$ of $CHT(n)$. If task instances are not created dynamically, we may use $CHT_0(n) = Task(n) - \{n\}$ as the initial estimate, where $Task(n)$ is the set of nodes in the task containing n.

We refine each estimate CHT_i of CHT to an estimate $CHT_{i+1} = r_i(CHT)$ by applying one of a set of conservative refinement analyses r_i. We require that all refinements be *monotone*, *inflationary*, and *conservative*, i.e.:

$$\begin{aligned}\forall r_i, n: \quad & [\forall m : CHT(m) \subseteq CHT'(m)] \\ & \Rightarrow [r_i(CHT)(n) \subseteq r_i(CHT')(n)] \\ \wedge \quad & CHT(n) \subseteq r_i(CHT)(n) \\ \wedge \quad & [\forall m : CHT(m) \subseteq CHT_{perf}(m)] \\ & \Rightarrow [r_i(CHT)(n) \subseteq CHT_{perf}(n)].\end{aligned}$$

As an example of a refinement, we may be able to decide, from the construction of the sync graph and CHT_i information currently available, that all instances of m happen before any instance of n, as is done in [CS88] and [DS91]. Thus, $m \in CHT_{i+1}(n)$ and $n \in CHT_{i+1}(m)$. Other forms of analysis are also possible; these must be specialized to the tasking model being analyzed.

While refining CHT, we also eliminate *extraneous* sync edges from the sync graph. A sync edge is extraneous if it cannot correspond to an executable synchronization. CHT information can be used to find extraneous sync edges; eliminating extraneous sync edges can, in turn, improve CHT information. Therefore, the sync graph and the CHT relation are refined in the same loop.

When no refinement can add any node to any of the CHT sets, we have reached a fixed point in the process of refinement. We then terminate refinement, returning the final CHT set and sync graph. As long as the refinements satisfy our requirements, they may be done in arbitrary order within the loop, and the same fixed point will be reached [Mast93]. The refinements need not be commutative for this guarantee to hold.

The refinement process terminates after performing at most $r|N|^2$ refinements, where N is the set of nodes in the sync graph and r is the total number of refinements available. This can be seen by a simple argument on the height of the lattice of CHT relations. Thus, if each refinement terminates in time polynomial in the size (nodes plus all edges) of the sync graph, then the entire CHT process also terminates in polynomial time.

Bristow, et. al. [BDER79] first mentioned the idea of iterative refinement of information similar to CHT, using a graph representation of programs similar to our sync graphs. Their work concerned only post/wait/clear synchronization, and they limited themselves to a single refinement. Our work thus generalizes their theory; in addition, we provide experimental evidence of the effectiveness of the technique.

2.1 Widening and pseudotransitivity of the CHT relation.

The CHT relation is not generally transitive; however, it admits two useful properties, which we call *widening* and *pseudotransitivity*. We will use these properties extensively in designing the refinements.

We can sometimes discover a set of nodes S for node n, such that whenever n is executing, some node(s) in S are executing as well. For example, n may be a node inside a remote procedure and S the set of nodes that call the remote procedure. Note that we do not require that a single node in S execute throughout the entire duration of n; the executing members of S may change during n's execution. We can then improve our $CHT(n)$ information using S as shown in Theorem 1:

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

In general, a small S yields the greatest chance for adding information to CHT. We can sometimes also expand CHT by finding S *within CHT itself.*

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

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

cannot happen together with n, where N is the set of all nodes in the sync graph and $CHT(n) \subseteq CHT_{perf}(n)$.

Pseudotransitivity is especially useful in cases where widening is being done to refine CHT, but some nodes have no set S. If we are manipulating relations as Boolean matrices, we may refine CHT by pseudotransitivity at the same time, by using $N - CHT(n) - \{n\}$ as a default value of S for node n. Doing this incurs little additional computation cost, may increase precision, and eliminates complicated handling of special cases. Furthermore, we can sharpen the accuracy of Corollary 1 by considering pseudotransitivity on a task-wise basis:

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

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

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

3 Rendezvous.

The Ada rendezvous model is quite complex. We will give here a simplified version of the complete model, with only the detail needed to illustrate *CHT* computation. For a more thorough treatment of the Ada rendezvous model, see [ANSI83]; for complete details of *CHT* detection in a larger subset of Ada, see [Mast92, Mast93].

In the Ada model, any task may signal other tasks, using a statement of the form `<acceptingTask>.<signalName>`. The signaler blocks until the accepting task has accepted, and completed processing of, the signal.

The accepting task accepts the signal at statements of the form, `accept <signal> [do <block>; end <signal>;]`. If no task has signaled, the accepting task blocks until one does; if multiple tasks have signaled, one is selected arbitrarily[3]. The optional `do` block is then executed; it may also contain rendezvous statements. While the `do` block executes, the signaling task remains suspended at the signaling statement. The `do` block is often used to implement a remote procedure call.

We exclude dynamic task creation and task access variables from our Ada subset model. Thus, the set of task instances, and the program statements which rendezvous with each other, are known precisely at compile time. In our work on Concurrent C [Mast93], we propose a non-concurrency analysis for the Concurrent C language [GR86], which includes dynamic task creation.

3.1 Ada sync hypergraphs.

Figure 2 shows a simple Ada program, and its *sync hypergraph* representation [MR91b]. Signaling statements are represented by a single *signaling node*; `accept` statements are represented by an *accept-in* node and an *accept-in* node, with the subgraph for their `do` block (if any) embedded between them.

```
int stop = 0;

task t1 is begin
  while !stop loop
    t2.s2;            -- 1
    accept s1 do      -- 2
      t3.call3;       -- 3
      t3.exit3;       -- 4
    end accept;       -- 5
    t2.s2;            -- 6
  end loop;
end t1;

task t2 is begin
  while !stop loop
    accept s2;        -- 7,8
    t1.s1;            -- 9
    accept s2;        -- 10,11
  end loop;
end t2;

task t3 is begin
  while !stop loop
    accept call3;   -- 12,13
    accept exit3;   -- 14,15
  end loop;
end t3;

begin
  for i in 1..100000 loop
    t3.call3;         -- 16
    t3.exit3;         -- 17
  end loop;
  stop := 1;
end;
```

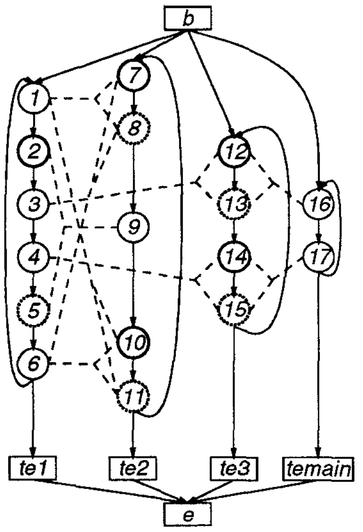

Figure 2: An example Ada program and its sync hypergraph.

Nodes b and e represent, respectively, the beginning and end of the program. In addition, a *task end* node is included for each task, to distinguish the end of each task from the end of the program. Doing so helps to eliminate special cases in certain analyses (Section 3.2.1), and enables the use of the task-wise pseudotransitivity of Corollary 2.

Control flow is represented conventionally by directed edges between nodes in the same task. Rendezvous are represented by three-ended *sync hyperedges*, connecting each signaling node with each accept-in/accept-out pair of the same signal type. A more detailed explanation of the sync hypergraph representation may be found in [Mast92, Mast93].

In our execution model, all program statements other than rendezvous are assumed to execute instantaneously. *CHT* remains conservative under this assumption [MR90].

3.2 Computation of *CHT*.

We exploit four refinements for *CHT* information. Each of these is described briefly in the following sections. In [Mast92, Mast93], we give more complete details of each refinement, and prove the correctness of each refinement given here, as well as polynomial worst-case time bounds.

3.2.1 Pinning analysis.

Pinning analysis is a general *CHT* refinement, based on widening the *CHT* relation using Theorem 1. Ex-

[3]This is a simplification of the true Ada model, which uses priority queues.

perimentation has shown that pinning analysis yields a wealth of *CHT* information.

Pinning analysis is best described by an example. Referring to Figure 2, node 2 cannot begin execution until its predecessor, node 1, has finished. Node 1 can rendezvous only with the pairs $(7, 8)$ and $(10, 11)$; thus, either node 8 or node 11 must complete immediately before node 2 starts. The successor of node 8 is node 9; the successors of node 11 are nodes 7 and and *te2*. Thus, one of these must start at the same time node 2 does. We call nodes 7, 9, and *te2* the *partners* of node 2.

Node 9 can rendezvous only with node 2; node *te2* cannot rendezvous with any other nodes. Node 7 can only rendezvous with nodes 1 and 6; neither of these may execute together with node 2, since they are in the same task as node 2. Thus, the partners of node 2 must continue executing at least as long as node 2 does, and node 2 is said to *pin* its partners. The set $\{7, 9, te2\}$ can thus be used as the set S of Theorem 1 to refine the *CHT* information for node 2.

Nodes may have the task end nodes of other tasks as partners. Including the task end nodes in the sync hypergraph helps to eliminate a troublesome special case in pinning analysis, which would occur if node e represented the end of each task, as well as the end of the program.

3.2.2 Remote procedure calls.

The nodes embedded inside a remote procedure, and the accept-out node for the procedure, may execute only while a signaling node that calls the remote procedure is executing. For instance, nodes 3, 4, and 5 of Figure 2 may execute only while node 9 is executing. Thus, the set $\{9\}$ can be used as the set S of Theorem 1 to refine the *CHT* information for nodes 3, 4, and 5.

3.2.3 Critical section analysis.

A *critical section structure* is a program construct which is often used to enforce mutual exclusion between tasks, without blocking the calling task as the remote procedure call does. In Figure 2, node pairs $(3, 4)$ and $(16, 17)$ delineate *call bodies* of a critical section structure, while node pair $(12, 15)$ delineates the *critical section body* of the structure[4].

Both call bodies and critical section bodies are single entry, single exit regions of the control flow subgraph, with additional restrictions detailed in [Mast92, Mast93]. The call bodies and critical section bodies exclude their entry nodes, but include their exit nodes. Knowledge about critical section structures can be used to derive *CHT* information in the following ways:

- No two call bodies of the same structure can execute simultaneously. Thus, every node in one call body for the structure may be placed in the *CHT* set of all nodes in other call bodies.
- The critical section body must execute concurrently with some call body. Thus, the set of nodes in all call bodies may be used as the set S of Theorem 1, for each node in any critical section body.
- Likewise, the call body must execute concurrently with some critical section body. Thus, the set of nodes in the critical section bodies form the set S of Theorem 1, for each node in any call body.

A more precise explanation of the behavior of critical section structures may be found in [Mast92, Mast93]. Empirical work indicates that this type of critical section structure is moderately common in practice (Table 1).

3.2.4 *B4* analysis.

B4 analysis attempts to discover pairs of nodes (m, n) such that all instances of m must occur before any instance of n. We denote this relation as $m \in B4(n)$. If $m \in B4(n)$, then $m \in CHT(n)$, and $n \in CHT(m)$[5].

Our approach in computing *B4* sets is, in principle, similar to that of [DS91]; however, there are differences in the set of Ada constructs addressed, the graph representation, and the propagation of *B4* information between tasks. We adapt the *B4* computation to the sync hypergraph model, as a *CHT* refinement, and also construct a lattice framework for iterative calculation of *B4*.

Computation of *B4* is done iteratively. We may place $m \in B4(n)$ if any of the following is true:

- There is a control path from m to n, but not from n to m.
- m is in the intersection of the *B4* sets of all control predecessors of n.
- m is in the intersection of the *B4* sets of all nodes that can rendezvous with the *completors* of n, i.e., the most immediate control ancestors of n that can complete a rendezvous. The completors of any node may include signaling and accept-out nodes, and the program begin node b. Figure 3 illustrates the concept of completors; the completors of x are s^5, a_o^3, and s^6.

These rules sketch the basics of *B4* computation.

[4]While the structure shown has only one critical section body, multiple critical section bodies are possible, with appropriate restrictions. See [Mast92, Mast93] for details.

[5][MR91b] detailed an extension of *B4* analysis to produce *CHT* information within loop bodies. This extension was later shown to be potentially unsafe, although no problems occurred in practice.

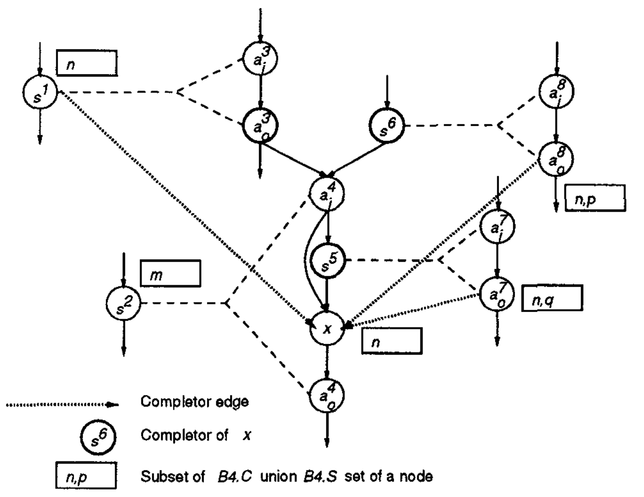

Figure 3: Completors of node x and completor edges to x.

***B4* lattice framework.** Lattice frameworks are a more complete encoding than the set of data flow equations used in [CS88] or [DS91], and can be used to deduce precision and convergence time properties. We present here the lattice framework we use for *B4* calculations; a similar framework is also used in a different concurrency analysis problem in Section 4.2.3. This provides a simple illustration of how data flow problems can be encoded as lattice problems, when information enters nodes through edges in different classes. Each edge class denotes a different source of information; therefore, a separate meet must be taken with respect to each edge class. To accomplish this, we represent information as a k-tuple, where k is the number of edge classes.

The lattice framework used for *B4* analysis is a quadruple $D = (G, L, F, M)$. $G = (N, E_C, E_{Completors})$, where N and E_C are as they were for the sync hypergraph, and $E_{Completors}$ is the set of pairs (m, n) such that m can rendezvous with a completor of n. $E_{Completors}$ is called the set of *completor (pseudo)edges.* In Figure 3, the completor edges to x are (s^1, x), (a_o^7, x), and (a_o^8, x).

L is a lattice of pairs of sets of nodes, (C, S); the meet operation is element-wise set intersection. Intuitively, if the framework has a pair (C, S) at node n, then C is the set of nodes in $B4(n)$ due to *control flow* into n, and S is the set of nodes in $B4(n)$ because of *synchronization* at the completors of n. Thus, $B4(n) = C \cup S$. For all $m \in B4(n)$, $m \in CHT_{perf}(n)$ and $n \in CHT_{perf}(m)$.

$F : L \to L$ contains the set of edge functions, and is the closure of the set of all possible edge functions under meet and function composition. $M : (E_C \cup E_{Completors}) \to F$ is a mapping from the edges of G to functions of F.

Let $CReach(m)$ denote the set of nodes q such that there is a control path from q to m, possibly of length zero. If $m \in CReach(n)$ and $n \notin CReach(m)$, then there is a control path from m to n, but not from n to m; thus, $m \in B4(n)$.

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

$$f_e((C, S)) = (C \cup S \cup K_e, N)$$

if e is a control edge, or

$$f_e((C, S)) = (N, C \cup S \cup K_e),$$

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.

We have shown in [Mast92, Mast93] that the functions of F are monotone and inflationary, and are therefore *1-semibounded* [MR91a]. However, the framework is not Kam-Ullman *rapid* [KU76] because the edge functions are not, in general, distributive.

3.2.5 Iterative refinement of *CHT* sets.

In the four refinements listed above, *CHT* sets are refined iteratively through two mechanisms. The most direct mechanism is seen in the refinements that use widening (i.e., all except *B4*). Since widening propagates *CHT* sets between nodes, improvements in *CHT* will also be propagated.

Eliminating extraneous sync hyperedges is also important for all four refinements. Suppose that a hyperedge includes signaling node s and accept-in node a_i, and that $s \in CHT(a_i)$. Then the hyperedge represents an unexecutable rendezvous. Such hyperedges introduce imprecision in each of the refinements to *CHT*, and can be safely eliminated from the refinements once we know that $s \in CHT(a_i)$.

4 Binary semaphores.

Binary semaphores, a specialization of integer-valued semaphores [Dijk68], are essentially Boolean flags. Since rendezvous can be implemented using a number of binary semaphores proportional to the number of rendezvous signal types, binary semaphores are at least as powerful a synchronization mechanism as rendezvous. (This is in contrast to post-wait-clear synchronization, in which the number of event flags must also increase with the number of signaling tasks.)

Binary semaphores are accessed only through the special commands p and v. A task executing the instruction $p(X)$ suspends execution until semaphore $X = 1$, then atomically decrements x to 0. If several tasks execute $p(X)$ simultaneously, only one can proceed until x is again set to 1. $v(X)$ sets the value of X to 1.

We assume that semaphores are declared to be either initially *set* (i.e, equal to 1) or *clear* (i.e., equal to 0). We will label initially set semaphores as S_i, and initially clear semaphores as C_i.

4.1 Binary semaphore sync graphs.

Figure 4 shows a program using binary semaphores, and its sync graph representation. $p(X)$ and $v(X)$ statements are represented by p and v nodes, respectively; task end nodes, and the program begin and end nodes b and e, are also included. Control paths between synchronization statements are represented by directed control edges; a directed sync edge is drawn from each $v(X)$ node to each $p(X)$ node for the same semaphore x. For such a set semaphore S, a sync edge is also drawn from the program begin node b to all $p(S)$ statements.

```
bool stop = false;
ClearSemaphore C1, C2;
SetSemaphore S;

task t1 is begin
  while !stop loop
    p(S);           -- 1
    v(C1)           -- 2
    p(C2);          -- 3
    v(S);           -- 4
  end loop;
end t1;

task t2 is begin
  while !stop loop
    p(S);           -- 5
    v(C1);          -- 6
    p(C2);          -- 7
    v(S)            -- 8
  end loop;
end t2;

task t3 is begin
  while !stop loop
    p(C1);          -- 9
    v(C2);          -- 10
  end loop;
end t3;

begin
  for i in 1..100000 loop
    null;
  end loop;
  stop := true;
end;
```

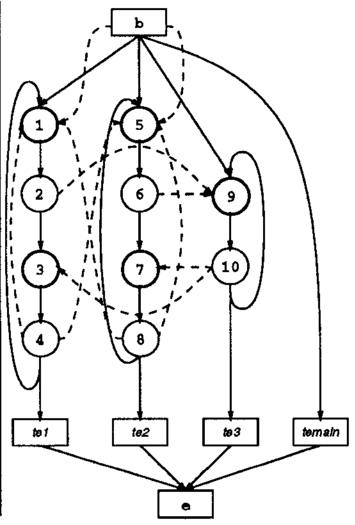

Figure 4: An example program which uses binary semaphore synchronization, and the sync graph of the program.

4.2 Computation of *CHT*.

We use three refinement analyses to estimate *CHT*: critical section and *B4* analysis, as with Ada, plus a new *mutex* analysis technique. Each of these is detailed in the sections below.

In the binary semaphore model, pinning does not occur, at least not at the abstraction level of semaphores, since a v node does not wait for a p node to clear the semaphore it sets. Therefore, we cannot perform pinning analysis. Also, since critical sections can be used to implement remote procedure calls in the binary semaphore model, we do not separate these two refinements.

4.2.1 Mutex analysis.

Set semaphores can easily be used to implement mutual exclusion. In Figure 4, the set semaphore S is used to implement mutual exclusion between node sets $\{2, 3, 4\}$ and $\{6, 7, 8\}$. If we can detect program structures that enforce mutual exclusion, we can use these structures as a source of *CHT* information.

Mutual exclusion is commonly enforced by embedding the exclusive code in a single-entry, single-exit region, with a $p(S)$ statement as the entry and a $v(S)$ statement as the exit, for some set semaphore S. As with critical section structures, some additional restrictions are also required, both on the node types and connectivity of the regions, and on the use of $p(S)$ and $v(S)$ statements throughout the program. If a set semaphore S obeys these restrictions, the p and v statements acting on it define a *mutex structure.*

The *mutex bodies* of a mutex structure are the regions bracketed by $p(S)$ and $v(S)$ statements, excluding the entry node. If a node n is in a mutex body, then all nodes in other mutex bodies of the same structure are in *CHT*(n).

4.2.2 Critical section analysis.

Critical section structures in a binary semaphore program are intended to function like those in a rendezvous program. One variant of critical section structures uses a pair of clear semaphores to signal calls to, and returns from, critical section bodies. In the program of Figure 4, semaphore $C1$ signals the call, and $C2$ signals the return. The call bodies of the critical section structure are delineated by entry/exit node pairs $(2, 3)$ and $(6, 7)$; the critical section body is delineated by node pair $(9, 10)$.

Note that call bodies $(2, 3)$ and $(6, 7)$ are enclosed in mutex structures. If they were not, then both call bodies could simultaneously call the critical section body. We require, as a condition of structural correctness, that all call bodies be in each others' *CHT* sets, since mutual exclusion of the call bodies could not otherwise be enforced. However, the entry nodes of the critical section bodies may be concurrent, since only one critical section body can be entered when a single semaphore is set.

Other variants of binary semaphore critical section structures exist; empirical work must determine which variants are the most common in practice. Then, frameworks can be devised for recognizing them and extracting *CHT* information. For at least some variants, *CHT* information is needed to determine whether a candidate critical section structure is recognizable. Thus, refining *CHT* information can reveal further critical section structures, which can in turn propagate *CHT* information.

4.2.3 *B4* analysis and the semaphore kill problem.

B4 sets in the binary semaphore model are generated and propagated much as they are in the rendezvous model. To iteratively refine *B4* information, we must eliminate extraneous sync edges, as we did in the rendezvous model.

There is an important difference between the rendezvous and binary semaphore models, which affects the elimination of extraneous sync edges. In the rendezvous model, sync edges may be eliminated solely on the basis of *CHT* information. In the binary semaphore model, however, the sync edges have different semantics; completion of the tail node represents an event that may enable the head node to complete execution, even when the head node executes later than the tail node.

To eliminate extraneous sync edges, we have to prove that the *definition* of the semaphore at the tail cannot reach the *use* of the semaphore at the head. This is called the *semaphore kill* problem; we say that $m = v(X) \in SemKill(n)$ if no definition of semaphore X by m can reach a use at n. This is true in at least two special cases:

- $n \in B4(m)$; or,
- Between any instance of m and any later instance of n, there must be another statement instance that sets or clears X. As in widening, the statement instances that modify X need not correspond to any single statement.

Bristow, et. al. [BDER79] showed a special case solution of this problem for post/wait/clear synchronization, but they limited themselves to problems in which the instances of a single statement modified X.

Our *SemKill* analysis for binary semaphores is similar to reaching-definitions analysis, in that we attach "generate" and "propagate" sets to control and synchronization edges [Mast93]. We use a lattice framework with multiple classes of edges, similar to the *B4* framework described in Section 3.2.4.

We briefly outline a set of conditions sufficient to determine that $m = v(X)$ is killed when n completes. Our *SemKill* lattice framework is based on these conditions.

- $n \in B4(m)$; or,
- $n \neq m$, and either $n = p(X)$ or $n = v(X)$; or,
- For all sync edges (n', n) such that n' is not killed at the top of n, either:
  - $m \in B4(n')$ and m is killed at n'; or,
  - $m = n'$;

  or,
- For all control predecessors n' of n, m is killed at the bottom of n' and $m \in CHT(n)$.

If m is killed at the top of n and $m \in CHT(n)$, then m is killed at the bottom of n.

5 Experimental work.

An experimental *CHT* analyzer was constructed for Ada, as part of a static deadlock analysis tool. The experiments were conducted on a Sun Sparc-2 processor with 225 megabytes of virtual memory. A total of 138 programs were successfully analyzed, using the polynomial-time analysis techniques of Section 3. We also were able to obtain CHT_{perf} sets for 127 of these programs by exhaustive state enumeration, in a manner similar to [Tayl83]. The remaining 11 programs had state spaces too large to enumerate within the available memory. Of the 127 programs for which we could determine CHT_{perf} experimentally, a total of 115 programs actually had pairs of rendezvous nodes in CHT_{perf}.

Execution time. Figure 5 shows the time taken by *CHT* analysis, as a function of the number of nodes in the sync graph. We see here that the actual time to estimate *CHT* is related to the size of the sync graph by a low-order polynomial (i.e., a polynomial whose degree in $|N|$ is ≤ 5, since $|N|^5$ is a factor of the theoretical worst-case limit); least-mean-square curve fittings also bear out this observation. For all programs tested, the *CHT* refinement process stabilized in five or fewer iterations.

Contribution of the refinement techniques. Table 1 shows the number of programs for which each refinement of *CHT* contributed to the total solution, and for which sync edges were pruned by *CHT*. From the table, we find that the great majority of programs have some *CHT* information; those that do not are quite simple in structure (i.e., one node per task). Pinning analysis disclosed at least some *CHT* information in 71.0% of the programs. The *B4* refinement was also of use in a sizable minority of the programs tested.

Figure 6 shows the relative contributions of each of the four techniques, as a percentage of the total number of ordered pairs (m, n) such that $m \in CHT(n)$. Pinning analysis and initialization are clearly the greatest contributors to the total *CHT* information, though the other techniques catch additional special cases that can seriously affect the accuracy of analyses which use *CHT* (e.g., deadlock detection). For instance, critical

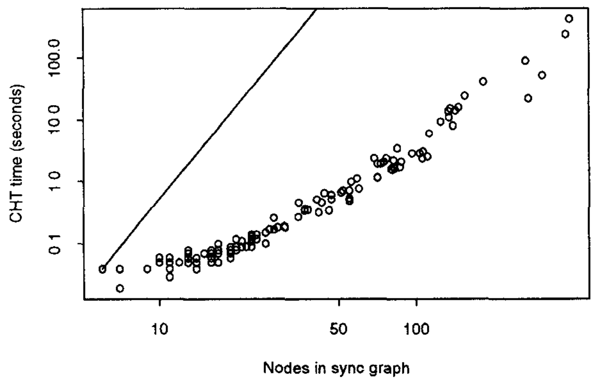

Figure 5: Time to estimate *CHT* (in user CPU seconds on a Sparc-2 processor), versus number of nodes in the sync hypergraph. The line indicates the $\mathcal{O}(|N|^5)$ factor of the theoretical time bound. The scale is log-log for clarity; a straight line on a log-log plot indicates a polynomial relation between x and y.

Refinement	Programs	Percent
Total programs	138	
Initial	126	91.3
Pinning	98	71.0
B4	24	17.4
Remote proc. calls	11	8.0
Critical sections	9	6.5
Pruned sync edges	25	18.1

Table 1: Number of programs with *CHT* contributions from each refinement, and for which sync edges were pruned. Note that a single program may have *CHT* contributions from more than one refinement.

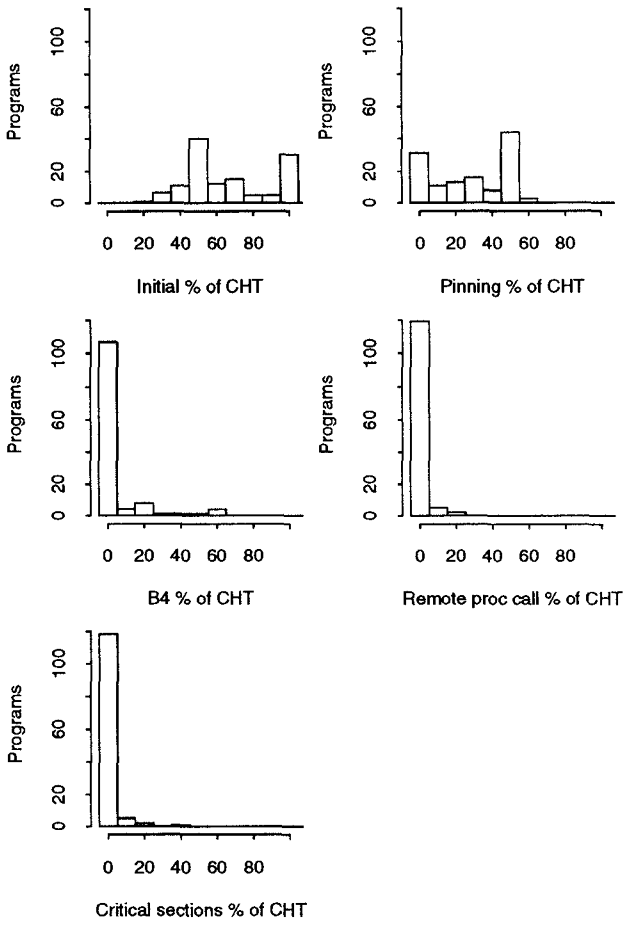

Figure 6: Histograms of the relative number of ordered *CHT* pairs contributed by initialization, pinning, *B4*, remote procedure call, and critical section analyses. The x axis is the percent of pairs contributed by the refinement shown. The y axis is the number of programs in which the specified percent of pairs was contributed by the refinement. The 12 programs with no *CHT* pairs are omitted.

section structures can cause incorrect reporting of potential deadlocks, unless they are recognized and their contribution to *CHT* is reported.

Accuracy of *CHT* estimation. Figure 7 (left) details the accuracy of the *CHT* estimate versus CHT_{perf}. For the vast majority of programs seen, the estimate was quite accurate. The plot shows that, for 92 of the 115 programs which had any pairs in CHT_{perf}, at least 95% of all pairs were found.

Comparison with [DS91]. Table 1 and Figures 6 and 7 (right) show that, while the *B4* analysis of [DS91] (and, by implication, [CS88]) are valuable as refinements, they clearly miss a large component of non-concurrency information in most programs. The other refinements we have presented are needed to obtain the majority of *CHT* information in the majority of programs.

Effectiveness of *CHT* in deadlock detection. *CHT* information has proven vital in static polynomial-time deadlock detection [MR91b, Mast92, Mast93]. To find deadlocks, we detect (but do not enumerate) cycles in the sync graph that might correspond to deadlocks. In such cycles, the nodes which represent deadlocked statements must be able to happen concurrently. We use *CHT* to prune away some spurious cycles, which helps to eliminate "false alarm" reports of deadlock.

To determine the effectiveness of *CHT* in preventing false alarms, we compared the false-alarm rate with, and without, *CHT* pruning. False alarms were found by comparing the results of our polynomial analysis against those of full concurrency state enumeration; we could thus detect false alarms in 127 candidate programs. We found that the use of *CHT* information prevented 34 false alarms, i.e., 87.2% of the 39 false

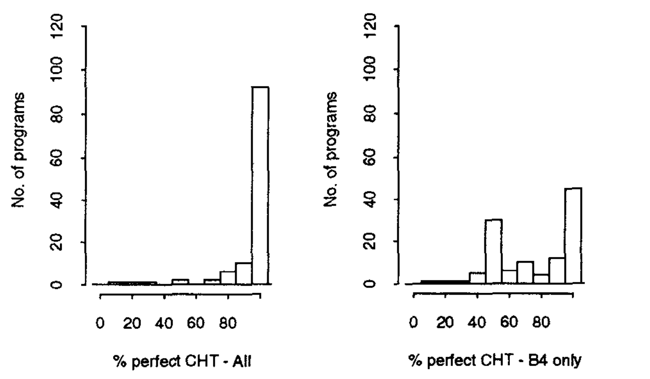

Figure 7: **Left:** Histogram of percent of pairs found by estimation, versus number of programs. All four of the *CHT* refinements of Section 3 were used. **Right:** Histogram of percent of pairs found by using only the *B4* refinement. This would be the result of the analysis of [DS91].

alarms that occurred when *CHT* was not used. Thus, non-concurrency information is crucial to the practical effectiveness of our deadlock detection system.

6 Conclusions.

Non-concurrency information about concurrent programs has a variety of uses in program analysis, optimization, and anomaly detection. It is possible to estimate non-concurrency information in time polynomial in the number of synchronization statements in the program. Experimental work has shown that these estimates are quite accurate, and that obtaining them takes little time in practice.

References

[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.

[BDER79] Bristow, G., Drey, C., Edwards, B., and Riddle, W. "Anomaly detection in concurrent programs." *Fourth IEEE International Conference on Software Engineering*, Munich, 1979, 265-273.

[CS88] Callahan, D. and Subhlok, J. "Static analysis of low-level synchronization." *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 1988, 100-111.

[CKS90] Callahan, D., Kennedy, K., and Subhlok, J. "Analysis of event synchronization in a parallel programming tool." *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990, 21-30

[Dijk68] Dijkstra, E. W. "Co-operating sequential processes." In *Programming Languages*, F. Genuys, ed., Academic Press, 1968, 43-112.

[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.

[GR86] Gehani, N. H. and Roome, W. D. "Concurrent C." *Software — Practice and Experience*, **16:9**, September 1986, 821-844.

[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." *ACM Symposium on Testing, Analysis and Verification (TAV4)*, Vancouver, October 1991, 21-35.

[Mast92] Masticola, S. P. "Detecting deadlocks in the Ada `accept...do` and `select` constructs." *LCSR-TR-190*, Laboratory for Computer Science Research, Rutgers University, 1992.

[Mast93] Masticola, S. P. "Static detection of deadlocks in polynomial time." Ph.D. thesis, Department of Computer Science, Rutgers University, 1993. In preparation.

[MR90] Masticola, S. P. and Ryder, B. G. "Static infinite wait anomaly detection in polynomial time." *1990 International Conference on Parallel Processing*, Chicago, August 1990, 2:78-87. ISBN 0-271-00728-1.

[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.

[RS90] Reif, J. H. and Smolka, S. A. "Data flow analysis of distributed communicating processes." *International Journal of Parallel Programming*, **19:1**, 1990, 1-30.

[Tayl83] Taylor, R. N. "A general-purpose algorithm for analyzing concurrent programs." *Communications of the ACM*, May 1983, 362-376.

[Tayl83b] Taylor, R. N. "Complexity of analyzing the synchronization structure of concurrent programs." *Acta Informatica*, **19**, 1983, 57-84.