Static Infinite Wait Anomaly Detection in Polynomial Time

Stephen P. Masticola

Barbara G. Ryder

Department of Computer Science

Rutgers University[*]

Abstract.

Infinite wait anomalies associated with a barrier rendezvous model (e.g., Ada) can be divided into two classes: *stalls* and *deadlocks*. Although precise static deadlock detection is NP-hard, we present two polynomial time algorithms which operate on a statically derivable program representation, the *sync graph*, to certify a useful class of programs free of deadlocks. We identify three conditions local to any deadlocked tasks, and a fourth global condition on all tasks, which must occur in the sync graph of any program which can deadlock. Again, exact checking of the local conditions is NP-hard; the algorithms check them using conservative approximations. Certifying stall freedom is intractable for programs with conditional branching, including loops. We give program transforms which may help alleviate this difficulty.

Keywords: synchronization anomalies, Ada, deadlocks, static analysis, parallel programming

1 Introduction.

Infinite wait synchronization anomalies associated with a barrier rendezvous model (e.g., Ada) can be divided into two classes: *stalls* and *deadlocks*. We have designed two polynomial time algorithms to statically certify that parallel programs are free of deadlocks under reasonable assumptions about program execution. Our first algorithm is efficient, but predictably imprecise; our second algorithm, a refinement of the first, achieves greater precision at greater cost, while retaining polynomial execution time.

Both algorithms work on the *sync graph*, a static representation of the control flow within tasks and their synchronization interactions. Four conditions on the sync graph are necessary for deadlock to occur; of these conditions, three are local to the deadlocking tasks and a fourth global condition involves all tasks in the program. The three local conditions form the

[*]Contact authors at Rutgers University Department of Computer Science, New Brunswick, NJ 08903. Telephone: 201-932-2001. Email: masticol@cs.rutgers.edu, ryder@cs.rutgers.edu.

foundation of our analysis techniques. A program is "certified" to be free of deadlocks by checking that these necessary conditions do not occur.

The sync graph can be checked for the absence of the first local condition in polynomial time. Checking the absence of the first condition in combination with either of the other two local conditions is shown to be NP-hard; this justifies our use of approximate techniques. Initial results on stall detection and avoidance also are presented.

As is common in static analysis, we assume that all control flow paths in a program are executable and that the control flow graph of a procedure is *reducible*, i.e., each loop has only one entry point[Hech77]. Both deadlock detection algorithms are *safe* in that if an anomaly is possible, they will report this possibility; however, sometimes they will report anomalous situations when none can actually occur. Using a barrier rendezvous model like that of Ada without `select` statements, we assume that program execution always will reach the end of each task if rendezvous statements do not cause any task to wait. For more convenient analysis, we further require that all tasks terminate, although some parallel programs (e.g., real-time systems) have tasks which, by design, do not terminate.

Intuitively, in a multiple-task program, an *infinite wait* is any failure of the program to terminate successfully due to the failure of one or more tasks to complete a needed rendezvous. The symptom of an infinite wait or synchronization anomaly is that some required part of a task t cannot execute, because t is waiting for a rendezvous which cannot possibly occur. In a *stall* (Figure 2(a)), some task waits on a rendezvous, but no other task can make that rendezvous at any future point in its execution. In a *deadlock* (Figure 2(b)), some set of tasks wait to rendezvous with each other, but no pair of tasks can achieve a rendezvous.

2 Infinite wait anomalies.

Our parallel programming model first is described in terms of tasks and their allowed interactions. Second, we define the sync graph, explaining how it embodies all possible program executions using execution waves. Finally, we define infinite wait anomalies in terms of our representation.

Tasks. Our model of synchronization is a subset of the full Ada rendezvous model of intertask communication. Basically, we have `accept` and entry call (i.e., `send`) statements but no `selects`. Tasks are statically created, i.e., tasks do not create other tasks at run time. All tasks are activated at the start of the program and all end at program termination. For the present, our model assumes that all rendezvous occur in the main procedure of the task; we hope to extend this model to an interprocedural one in later work.

Tasks send messages to each other. Each task is uniquely identified by a task identifier. Messages are directed to a specific task, and are of a specific message type. A receiving task/message pair is called a *signal*; signals may be denoted by (t, m), where t denotes the task receiving the signal and m is the message type of the signal. The number of message types is finite and statically discernible. In a `send`, the receiving task is explicitly identified, whereas in an `accept`, the sending task is not specified.

Upon executing a `send`, the task which sends a message (i.e., the *signaling task*) suspends execution until the task to which the message is directed (i.e., the *accepting task*) has executed an `accept` command of the same message type. Once the accepting task has executed its `accept`, the two tasks can communicate and the sending task can continue execution.

Upon executing an `accept`, the accepting task suspends execution until another signaling task has sent a message of the

type specified by the `accept`. A task may send or accept the same signal repeatedly. Any number of tasks can signal an accepting task.

A *rendezvous point* is any statement in a program which sends or accepts a signal. We assume that all rendezvous points in a task are reachable on some execution path from the task entry. We also assume that control flow in any task is independent of any other task. We use the notation (t, m, s) to specify a particular rendezvous point, where (t, m) denotes the signal type and s is defined to be *plus* $(+)$ if the point is a signaling rendezvous point and *minus* $(-)$ if the point is an accepting rendezvous point. $\overline{s}$ denotes a sign complementary to s. For example, in Figure 1 the rendezvous point coded as `t2.sig1` in task `t1` would be denoted as $(t2, sig1, +)$ whereas the rendezvous point coded as `accept sig2` would be denoted as $(t1, sig2, -)$.

Sync graphs. The *sync graph* is an abstract representation of a program which includes sufficient control flow and synchronization information to detect infinite wait anomalies. Figure 1 shows a program and its sync graph.

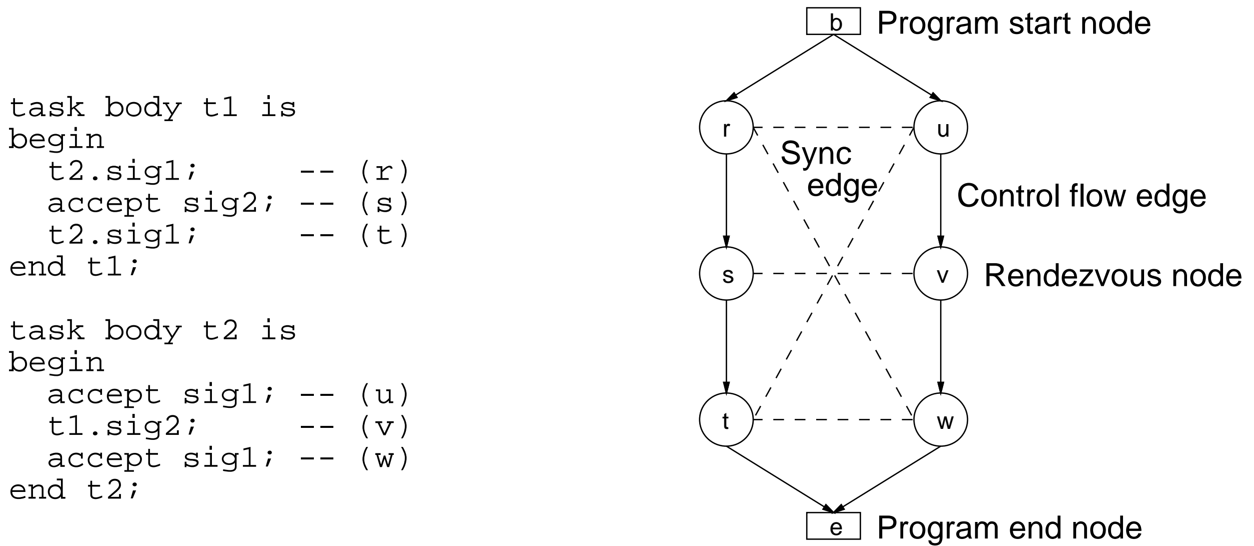

Figure 1: A program and its sync graph. In all figures, nodes of the same task are arranged vertically.

The sync graph of a program P is a quadruple, $SG_P = (T, N, E_C, E_S)$ where

- T is the set of tasks in the program.
- N is the set of nodes in the sync graph, and includes one node r corresponding to each rendezvous statement in the program.[1] N also includes two distinguished nodes b and e, which denote respectively the beginning and end points of the program.
- E_C is the set of control flow edges in the sync graph which represent realizable control flow between rendezvous points in the program. A directed edge $e_c = (r, s)$ is present in E_C if there is a control flow path between r and s in the program which includes no other rendezvous points.
- E_S is the set of synchronization edges (or sync edges) in the sync graph which represent possible synchronizations in the program. An undirected sync edge $e_s = \{r, s\}$ exists between every pair of complementary rendezvous points of the same signal type in N; that is, if $r = (t, m, +)$ is a rendezvous point then there will be a sync edge between r and

[1] In the remainder of this paper we use *rendezvous point* and *node* interchangeably.

every rendezvous point $s = (t, m, -)$ in the sync graph.

Model of program execution. Consider a program P without cycles in the control flow subgraph (N, E_C) (i.e., a loop-free program). The program begins execution at node b, the fork point for all tasks. For each task u which contains a rendezvous point, the first such node $r = (t, m, s)$ becomes *potentially executable.* (If more than one node in u can be reached directly from b, exactly one of these, chosen nondeterministically, becomes executable. This choice models a conditional branch to the node.) Any pair of potentially executable nodes $r = (t, m, -), s = (t, m, +)$ which can rendezvous with each other may do so nondeterministically, at which time their control flow successors become potentially executable.

Thus, simulation of a program execution may be seen as the advance of an *execution wave* of potentially executable nodes along control flow edges. At any point during the execution simulation, the execution wave contains one node belonging to each task, including possibly b or e.

Under the assumption that the control flow subgraph has no cycles, each node in the sync graph will be in one of the following states:

NOT-SEEN. The node has not yet been reached in control flow.

READY. The node has been reached in control flow, and *can rendezvous* with another node which has also been reached but not yet executed.

WAITING. The node has been reached in control flow, but *cannot rendezvous* with any other node which has been reached but not yet executed.

EXECUTED. The node already has achieved rendezvous, and execution of the program is past it. Such nodes cannot be re-executed.

All nodes on the execution wave are READY or WAITING. Any node which has not yet been a member of the execution wave is NOT-SEEN, and any node which has been removed from the execution wave is EXECUTED.

We represent an execution wave by a vector W, which has one entry per task; for task u, $W[u]$ is the node r which is the chosen potentially executable node in task u (i.e., next rendezvous to be executed). The initial execution wave, W_{INIT}, corresponds to the state of the program when it starts execution, and contains, for each task u, some node r which is a control flow successor of b.

Initially, all nodes are NOT-SEEN except b which is READY. (If there is a control flow edge (b, e) in task u, then $W_{INIT}[u]$ may be e. In this case the initial state of e is READY.) The execution wave advances (simulating the program's execution) when some pair of READY nodes on the wave connected by a sync edge make a rendezvous and thus become EXECUTED. This changes the state of each of their control flow successors w from NOT-SEEN to either READY or WAITING, depending on the state of the synchronization neighbors of w.

More formally, let W be an execution wave and W', an execution wave directly derived from W by the rendezvous of nodes r and s, where r is in task q and s is in task z. Then for each task u, $W[u]$ and $W'[u]$ are related as follows:

- For $u \neq q$ and $u \neq z$: $W[u] = W'[u]$. The state of node $W[u]$ is the same on both waves.
- For $u = q$: $W[u] = r$ and $W'[u] = t$ where t is a control flow successor of r. When W' is the current execution wave, node r is EXECUTED and node t is READY if there is any task j such that $\{W'[j], t\} \in E_S$, or WAITING otherwise.
- For $u = z$: $W[u] = s$ and $W'[u] = y$ where y is a control flow successor of s. When W' is the current execution wave, node s is EXECUTED and node y is READY if there is any task j such that $\{W'[j], y\} \in E_S$, or WAITING otherwise.

There may be several different execution waves derivable from a particular execution wave W, because there may be more than one pair of READY nodes on the execution wave. The set of *feasible execution waves* achievable from an initial wave W_{INIT} describes all the synchronization states realizable by a program. Define $NextWaves(W)$ to be the set of execution waves derivable from wave W. Assuming V is a set of execution waves, let the function $NextWavesSet(V)$ define the set of execution waves derivable from some execution wave in V using the the $NextWaves$ function; that is, $W \in NextWavesSet(V)$ iff there exists a wave $Z \in V$ such that $W \in NextWaves(Z)$.

Then we wish to obtain:

$$
\begin{aligned}
NextWavesSet^*(W_{INIT}) &\stackrel{def}{\equiv} \\
&\{W_{INIT}\} \\
\cup\ &NextWavesSet(\{W_{INIT}\}) \\
\cup\ &NextWavesSet(NextWavesSet(\{W_{INIT}\})) \\
\cup\ &NextWavesSet^3(\{W_{INIT}\}) \ldots
\end{aligned}
$$

or the transitive reflexive closure of $NextWavesSet$ applied to W_{INIT}. Thus $NextWavesSet^*(W_{INIT})$ calculates the set of *feasible execution waves* for a program. If an infinite wait anomaly can occur, it must be associated with some feasible execution wave.

Although our model of parallel program execution assumes that only one rendezvous may execute at any time, it is clear that composing applications of $NextWavesSet(W_{INIT})$ can produce an execution wave where all feasible disjoint rendezvous have occurred. Thus, our representation generates waves which would have resulted if multiple rendezvous were possible within a single update of the execution wave.

The model further assumes that there is no delay between the execution of a rendezvous and its control flow successor. In an actual program, such delays always exist, but do not matter for the purposes of modeling possible sequences of rendezvous.

Infinite wait anomalies. An execution wave W is *anomalous* if it contains at least one rendezvous point[2] and if no node in W can rendezvous with any other node in W. Formally,

$$
\begin{aligned}
Anomalous(W) &\stackrel{def}{\equiv} \exists t : (W[t] \neq b \bigwedge W[t] \neq e) \\
&\bigwedge \forall x : \forall y : ((x = W[u] \bigwedge y = W[v]) \Rightarrow \{x, y\} \notin E_S)
\end{aligned}
$$

A program P has an *infinite wait anomaly* if one of its feasible execution waves is anomalous. Any node on an anomalous wave is WAITING; a node directly reachable from it along an synchronization edge is either EXECUTED or NOT-SEEN.

Execution wave W has a *stall anomaly* if W is anomalous and includes a node $r = (t, m, s)$ such that there is no node $z = (t, m, \overline{s})$ reachable on a control flow path from any node on W. There is no node which at some later time can execute and rendezvous with r in task u; thus, task u can never proceed. We call r a *stall node*, since its failure to rendezvous prevents u from completing execution. Figure 2(a) illustrates a stall anomaly with r as the stall node.

Execution wave W has a *deadlock anomaly* if W is anomalous, and there is a set of nodes $D \subset W$ such that for any node $r \in D$ there is a node $s \in D$ which has a control flow descendent that is a sync edge neighbor of r. Thus, there is no distinguished stall node in D. Every task u with a node in D can execute further only if a node in D not in task u proceeds first. Figure 2(b) shows the sync graph of a program with a deadlock anomaly.

Deadlocks and stalls may exist within the same execution wave. Tasks which can rendezvous only with stalled or deadlocked tasks may also be prevented from executing by the anomalies, even though they are not directly contributing to

---

[2] All nodes in W may be either b or e; in either case, W is not anomalous.

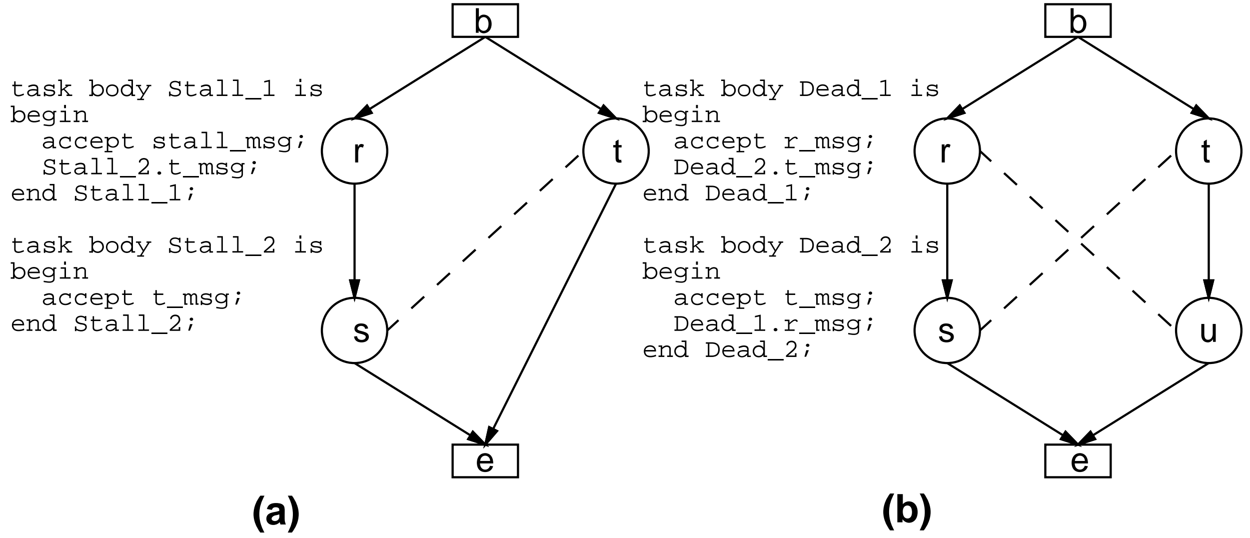

Figure 2: Sync graphs of a program with a stall anomaly (a) and with a deadlock anomaly (b).

the anomaly.

A node r in an anomalous execution wave W is *coupled* to node s in W if there is a path from s in task u to r, forward through at least one control flow edge in u and containing exactly one sync edge. r may thus rendezvous with a node which executes after s in u.

Node r is *transitively coupled* to node $t \in W$ if r is coupled to t, or if r is coupled to some node $s \in W$ which is transitively coupled to t. Transitive coupling thus forms chains of tasks. It is easy to see that nodes whose transitive coupling chains terminate in anomalies are themselves prevented from executing further by the anomalies. The following proof uses transitive coupling to show that deadlocks and stalls completely cover the possible infinite wait anomalies in a program as exhibited by its execution waves.

Theorem 1 *All nodes on an anomalous execution wave must participate in stalls or deadlocks, or be transitively coupled to some stalled or deadlocked task.*

Proof: All nodes in an anomalous execution wave W must be stall nodes, or participants in a deadlock, or neither. Let r be a node of the third kind, and assume that r is not transitively coupled to any stalled or deadlocked node.

If r can rendezvous with any node on the execution wave, then W is not anomalous. If r has no rendezvous with any NOT-SEEN node which is reachable by a control flow path from some node in W, then r is a stall node, violating our assumption. Thus, r must be able to rendezvous with some NOT-SEEN node, and is coupled to the representative s in W of the task containing that node. If s is transitively coupled to a stall or deadlock, then the assumption about r will be violated.

We may thus partition W into two nonempty sets of nodes: one which contains those nodes which are reflexively transitively coupled to stalls or deadlocks, and one (call it R) which is not. Nodes r and s (and, by generalization, all nodes in R) are transitively coupled to some other node in R. If this transitive coupling forms a cycle, then we have a deadlock in R. By the partition, R cannot contain any deadlocks, so the transitive coupling may not form a cycle. So we can form a topological ordering on the nodes in R using the transitive coupling relation. In this order, there must be a last node, which is not transitively coupled to any other node. But this is impossible, since all nodes in R are coupled to other nodes in R. Thus, R must be empty. □

3 Deadlock conditions.

A program consisting solely of straight-line code can deadlock only if it has a feasible execution wave such that some subset of nodes in the wave are coupled in a cycle. This result is not in itself novel [CD73]; our approach is to refine an approximation to feasible execution waves and use that as a basis for cycle detection. We require that the refined algorithm execute in low-order polynomial time.

Let D be the set of *head nodes*, or nodes of a deadlock cycle in an anomalous execution wave. The following conditions are necessarily true for the nodes in D.

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 edge.
   (b) The path traverses at least one control flow edge in task u.
   (c) The path leaves u via a sync edge and does not re-enter u.
2. No two head nodes of the deadlock cycle can rendezvous with each other. (If they could, the execution wave could advance.)
3. All head nodes may execute at the same time. This implies:
   (a) No two nodes in D are *sequenceable*; that is, no pair of nodes in D are such that one must always finish executing before the other starts[3]. (If the nodes in D were sequenceable, then they could not simultaneously be in the execution wave.)
   (b) All nodes in D are *co-executable* in the sense of Callahan and Subhlok [CS88]; that is, all the nodes in D may be executed in the same run of the program.
4. When the nodes of D are executing simultaneously, this must not imply that a node which can rendezvous with a member of D is also executing with them. Alternately stated, the deadlock is not always broken by some task outside the deadlock.

Note that the first three constraints apply only to the tasks participating in the deadlock, while the fourth may apply to all tasks in the program. Thus, we say that the first three constraints are *local* with respect to the deadlock, and the fourth is *global*. A simple depth-first search can find cycles which satisfy the first constraint in a slightly transformed version of the sync graph. Unfortunately, cycles satisfying the first constraint occur frequently in programs which cannot deadlock; we therefore need to refine the detection algorithm by applying the second two constraints.

Constraint 4 can be used to eliminate spurious cycles in some simple cases, as in Figure 3. Nodes r and t are head nodes in a deadlock cycle r, s, t, u which is valid according to the first three constraints. However, whenever t is ready to execute, w must also be ready, since w can only rendezvous with t or some node (here v) which must execute after t. Thus, w can always rendezvous with t and break the deadlock. Methods of applying constraint 4 more generally are under investigation.

Detecting cycles under the second and third constraints, in conjunction with the first, is NP-hard, given only the sync graph as input. However, a safe approximation to all three local constraints results in a polynomial time algorithm consisting

[3] We can generalize this constraint to apply to all *disjoint subsets* of nodes in D. There are no nonempty subsets D_1 and D_2 of D such that some node in D_1 must finish before any node in D_2 can start.

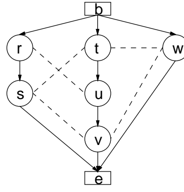

Figure 3: A deadlock cycle which violates the fourth constraint.

of a specialized search for strongly connected components. Assuming that we are given co-executability information about the program through other static analysis, we can include this in a similar manner.

3.1 Naive algorithm: Sync graph cycles.

A depth-first traversal of the sync graph, starting at node b and including both control and sync edges, will find a cycle if one exists. (We are assuming acyclic programs for the moment, and thus need not worry about control flow edge cycles.) Such cycles will not necessarily reflect constraint 1b. In particular, it is possible to traverse sync edges into and out of nodes in SG_P without traversing any control flow edges in the task thus entered. This will lead to spurious cycles being detected. Figure 4(a) shows a sync graph with spurious cycles in sync edges connecting nodes `r`, `s`, `t`, and `u`.

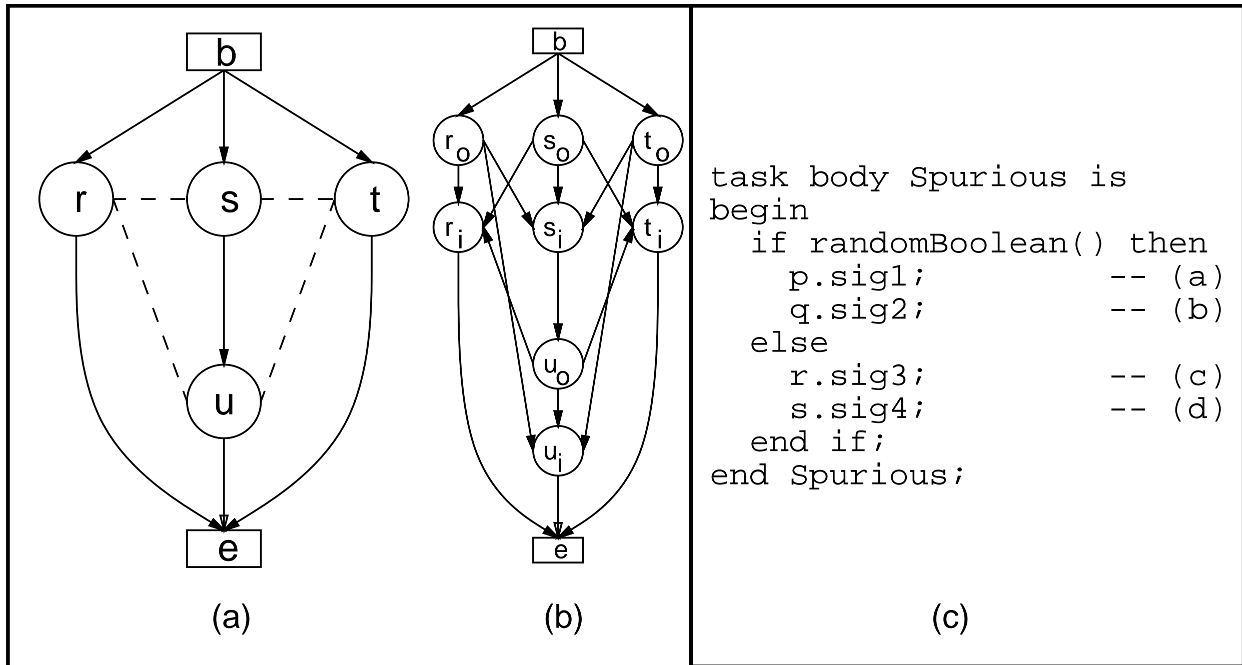

Figure 4: *Left:* Sync graph with a spurious cycle (a) and its cycle location graph (b). *Right:* Task in a spurious deadlock cycle (c).

We avoid detecting these spurious cycles by transforming the sync graph to a more suitable *cycle location graph*, or CLG. The effect of the transformation is to split the nodes and sync edges of the sync graph so that any (transformed) node entered via a sync edge can only be exited via a (transformed) control flow edge. Thus, constraint 1b is enforced in a depth-first search of the CLG.

The CLG is a directed graph of the form $C_P = (N_{CLG}, E_{CLG})$. N_{CLG} represents the nodes of the CLG, and E_{CLG} represents its set of directed edges. In the CLG, no distinction is made between sync and control flow edges, since we use it only to detect cycles.

The cycle location graph is constructed from the sync graph $SG_P = (T, N, E_C, E_S)$ as follows. Nodes whose names have an i subscript are constructed with incoming sync edges only; those with an o subscript have only outgoing sync edges.

1. Create distinguished nodes b and e in N_{CLG}.
2. For each node r other than b or e in N, create a pair of nodes r_i and r_o in N_{CLG}.
3. Create a directed edge (r_o, r_i) in E_{CLG} for each pair of nodes created in step 2.
4. For each control flow edge (b, r) in E_C, create an edge (b, r_o) in E_{CLG}. For each control flow edge (r, e) in E_C, create an edge (r_i, e) in E_{CLG}.
5. For each control flow edge $(r, s), r \neq b, s \neq e$ in E_C, create an edge (r_i, s_o) in E_{CLG}.
6. For each undirected sync edge $\{r, s\}$ in E_S, create directed edges (r_o, s_i) and (s_o, r_i).

Figure 4(b) shows the CLG formed by transforming the sync graph of 4(a). Note that there are no cycles in the CLG, which is sufficient to indicate that the program will not deadlock.

3.1.1 Straight-line code.

Consider a program whose tasks contain only "straight-line" code, without any conditionally executed nodes. The program has no control flow cycles. The CLG effectively breaks those paths which enter and exit some task without traversing some control flow edge in the task. Any cycle found in the CLG of such a program will thus satisfy constraints 1a and 1b.

Constraint 1c does not matter in straight-line programs. If some cycle in the sync graph enters a task more than once while obeying 1a and 1b, then the graph also has a cycle which enters the task only once.

3.1.2 Programs with conditional execution.

Consider applying cycle detection to programs with conditional execution, but without control flow loops. We assume that any control flow path may be taken independently by each task.

Specifically, consider a node s which is executed conditionally. There is a control flow edge $(r, s) \in E_C$ for each node r which may execute immediately before s without intervening nodes between r and s. Similarly, there is an edge $(s, t) \in E_C$ for each node t which may execute immediately after s without intervening nodes. It is easy to see that any control flow path through the node in a loop-free program corresponds to a path through the edges in E_C of the sync graph. Thus, if the program deadlocks, then there is a deadlock cycle in the sync graph, and we can detect the deadlock cycle as before.

However, there may be a spurious deadlock cycle which does not correspond to any realizable execution of the program. For instance, consider the task of Figure 4(c). In the sync graph, it is possible for both control flow edges (a, b) and (c, d) to be included in a cycle which cannot exist without including both of these edges[4]. Such cycles tend to increase the number of false deadlocks reported by static analysis. Spurious cycles can be at least partially suppressed by the methods of Section 4.2, at an increased cost in computation time.

[4]Note that such a cycle would actually violate *two* constraints: 1c (a deadlock cycle may not enter the same task twice) and 3b (head nodes in the cycle must be co-executable).

When we allow conditional paths in the program, constraint 1c (which requires that cycles enter each task only once) is no longer enforced automatically by the sync graph construct. For instance, it is possible for a spurious deadlock cycle to enter a task at some node r, leave, and re-enter at a node s such that there is no control flow path between r and s. Unlike the case in section 3.1.1, we cannot say that there is any cycle which enters the task once. However, here r and s are not co-executable, so constraint 3b is violated. By generalization, any cycle which enters the same task twice either corresponds to one which enters the task once or one which violates 3b.

3.1.3 Anomalies and source transforms.

The CLG can be directly applied to programs without loops, but control flow loops cause problems. However, it is possible to transform programs with reducible control flow graphs into programs without loops, while retaining enough of the program's synchronization behavior to detect deadlocks using the CLG. Some definitions are needed before we discuss this transform in detail.

Anomaly preserving transforms. Consider applying an *anomaly preserving* transform T to a program P such that, whenever an anomaly is present in P, some instance of the anomaly must also be present in $T(P)$. If T does not add anomalies to correct programs, we can also say that it is *precise*, in the sense that it will not cause "false alarms". Specifically, if we can find a transform which removes control flow loops from a program while preserving deadlock anomalies, then we can simply detect deadlocks in the transformed program.

Linearized execution. Consider a specific execution E of a program P. We can form a corresponding *linearized* version P_E of P, which contains no conditional branches, but which executes nodes in the same order (within each task) as E. If there is a sync anomaly in P, then the same anomaly must exist in some P_E. Thus, if a transform T is precise for all linearized executions of program P, then T is precise for P.

We can form P_E by unrolling loops in P as many times as they are actually executed and pruning conditional paths which are not taken in E. Alternatively, P can be partially linearized by unrolling loops without pruning branches.

3.1.4 Programs with loops.

We cannot directly apply the CLG method if the program has control flow loops. For example, if we retain all control flow edges between nodes, then the back edges of the loops themselves will be traversed by a depth first search (DFS) of the sync graph. We cannot merely prohibit traversing control flow back edges, since a deadlock cycle may enter a loop body in one iteration and exit in the next. To use the CLG, we must transform the program in such a way that control-edge cycles are not formed and all deadlock cycles are preserved by the transform.

Clearly, any execution of a program P is semantically equivalent to some execution of a program P_E such that P_E has all of its loops unrolled in the same manner in which they were executed in P (with the iteration counts of nested loops preserved). A deadlock cycle will exist in P if and only if the deadlock cycle also exists in P_E for some execution E.

Unrolling the loops in P is impractical for large iteration counts, and is impossible when iteration counts depend on input data. Fortunately, an alternative is possible which is computationally attractive and preserves deadlock cycles, and results in a sync graph without control flow loops.

Lemma 1 *Consider the transform $T(P)$ which unrolls each loop in P twice (recursively, from innermost to outermost nest levels). The sync graph of program $T(P)$ will contain all deadlock cycles present in any linearized execution of P. Furthermore,*

$T(P)$ can only contain deadlock cycles which are contained in some linearized form of P where loops are unrolled more than twice. Thus, T is anomaly preserving and precise.

Proof: Recall that deadlock cycles enter and exit the control flow paths of tasks at exactly one point. Assume that, for some execution E of program P, we have produced a partially linearized program P_E in which loops have been unrolled exactly as they were executed. Now consider the paths which deadlock cycles can take through P_E which traverse some unrolled part of the loop. Call the node where the cycle enters the task r_{in}, and the exit node for the task r_{out}. We will show that, for all orderings of r_{in} and r_{out} with respect to the unrolled loop body in P_E, there exists in $T(P)$ a control flow path from a node of the same type as r_{in} to a node of the same type as r_{out}.

- r_{in} *before the loop,* r_{out} *after the loop.* Only control flow edges are traversed in P_E. A control flow path exists between r_{in} and r_{out} in $T(P)$.
- r_{in} *before the loop,* r_{out} *inside the loop.* Nodes which rendezvous on signals corresponding to r_{in} and r_{out} are present in $T(P)$ as well, and there is a control flow path between them.
- r_{in} *inside the loop,* r_{out} *after the loop.* The same as the previous case.
- r_{in} *and* r_{out} *inside the loop.* Nodes which rendezvous on signals corresponding to r_{in} and r_{out} are present in $T(P)$ as well. Because the loop was unrolled twice in $T(P)$, a control flow path must also exist between at least one node corresponding to r_{in} and one corresponding to r_{out}.

Thus, if there is an execution path between any two nodes of type r_{in} and r_{out} in P_E in any task, then such a path also exists in the same task in $T(P)$, and vice versa. So $T(P)$ is a safe approximation to P_E for any execution. □

Recursively unrolling loops once per nest level will result in a control flow graph with worst-case size $\mathcal{O}(statements \times 2^{nest\ levels})$, if almost all statements in the program are contained in the innermost loop. Loops are not frequently nested to great depth in actual programs, so the size of $T(P)$ should not grow explosively with the size of P except in pathological cases [Knut71].

4 Refined algorithm.

Unfortunately, if we attempt to detect deadlocks using constraint 1 alone, we will incorrectly diagnose too many deadlocks in programs which do not actually deadlock. For instance, if we construct a CLG for the program of Figure 1, we will detect a deadlock cycle involving sync edges to nodes r, t, u, and w, and another cycle involving r, s, v, and w.

Therefore, we need to consider the feasibility of execution waves in finding deadlock cycles in the CLG. For example, nodes r and u in Figure 1 cannot be members of a deadlocked execution wave. Since they can rendezvous with each other, any execution wave containing both of them cannot, by definition, be anomalous.

The cycle involving r, s, v, and w is also spurious, since in this cycle r and v must be on the execution wave simultaneously. This is also impossible, since s can rendezvous only with v, and s must follow r. Therefore, we know that v must execute after r.

4.1 Deadlock detection complexity.

Taylor [Tayl83b] showed that exact static detection of possible deadlocks in straight-line Ada programs without `select` statements is NP-complete. Here, our intention is to develop a conservative polynomial-time algorithm which can prove that

some useful subset of programs is free of deadlock.

We examine the complexity of algorithms which respect constraints 2 and 3a: that valid deadlock cycles have no pairs of head nodes which can rendezvous and have no sequenceable pairs of head nodes.

Unsequenceable head nodes. Consider detecting deadlocks in a loop-free program when the partial ordering governing node execution is available. It is possible to derive a subset of the node orderings directly from the sync graph; we might use a data flow framework based on the following two rules, similar to the $SCPreserved(k)$ lattice of Callahan and Subhlok [CS88].

1. If r dominates s in the control flow graph of their task, then r must precede s.
2. If there is a node r such that for all sync edges $\{r, s\}$, s precedes some node t, then r must precede t.

Even when we have exact ordering information, the problem of deadlock detection is intractable. Theorem 2 in Appendix A demonstrates that a program can be created corresponding to an instance of the 3-satisfiability problem. This program will have a deadlock cycle with unorderable head nodes iff the given expression is satisfiable. Thus, detecting deadlock cycles satisfying both constraints 1 and 3a is NP-hard.

Head nodes that rendezvous with each other. Constraint 2 specifies that no pair of nodes in a deadlocked execution wave D can be connected by a sync edge; if they were, they could rendezvous and the execution wave could advance. Theorem 3 in Appendix A shows that detecting deadlock cycles that satisfy constraints 1 and 2 is NP-complete.

4.1.1 Cycles with rendezvousing head nodes.

We make the following observation about rendezvousing head nodes, which will be useful in eliminating some spurious cycles.

Lemma 2 *Consider a program whose sync graph has a cycle which is valid under constraint 1, but which is spurious under constraint 2, i.e., a pair of its head nodes are connected by a sync edge. At least one of the tasks in this cycle is entered and exited through accept nodes of the same signal type.*

Proof: Consider two tasks in such a cycle. One is a signaling node s and the other must be an accepting node a. (See Figure 5(a).) Node a, and all other accept nodes of the same signal type (t, m), must be part of the same task t. Since s is a head node, it must rendezvous with the tail node of some task in the cycle. The tail node must also be an accept node of type (t, m), which can only be a part of task t. Call this node a'. Thus, the head and tail nodes of task t are a and a' respectively, and both are accept nodes of the same type. Since the cycle is valid under constraint 1, there is a control flow path from a to a' and $a \neq a'$. □

If we can eliminate partial paths which enter and leave a task via accept nodes of the same signal type, we can eliminate spurious deadlock cycles where head nodes can rendezvous. To do this, we might require that any cycle which enters a task through an accept node of a given type leave the task through another type of node.

4.2 Conservative approximation algorithm.

Even if we cannot eliminate all spurious deadlock cycles using a tractable algorithm, we can use the constraint to eliminate some of them through simple testing. The algorithm presented here finds strongly-connected components in the CLG once for each node r which might be the head node of a deadlock cycle. We hypothesize for each r selected that r is a head node and eliminate edges in the CLG which would be included only in spurious deadlock cycles.

If, for all hypothesized head nodes r, no strong components containing r are found by this search, then the program can be declared deadlock-free. If a strong component containing r is found, we conservatively declare that it is a possible deadlock, even though it might not correspond to a feasible execution wave. Thus the algorithm remains both conservative and tractable.

Algorithm: Deadlock cycle detection with partial elimination of spurious cycles.

Inputs:

- A sync graph $SG_P = (T, N, E_S, E_C)$ and its derived CLG $C_P = (N_{CLG}, E_{CLG})$ corresponding to a program P. (Mappings between each sync graph node r and its derived CLG nodes r_i and r_o are assumed to be available.)
- A set POSS-HEADS $\subset N$, containing all possible head nodes. A rendezvous node is a possible head node if it is connected to at least one sync edge and is the tail of at least one control edge leading to another rendezvous node in the sync graph.
- A vector[5] SEQUENCEABLE[ ], containing for each node r those nodes which are sequenceable with r.
- A vector NOT-COEXEC[ ], containing for each node r those nodes which are not co-executable with r.
- A vector COACCEPT[ ], containing for each accept node r those accept nodes of the same signal type as r, and containing the empty set for each signaling node.

Output: `TRUE` if P is proven deadlock-free, `FALSE` otherwise.

For each node h in POSS-HEADS **do begin**

    Remove all marks from all nodes in the CLG;

    For each node k in SEQUENCEABLE[h] **do**

        Mark k_i as `NO-SYNC`;

    For each node k in COACCEPT[h] **do begin**

        Mark k_i as `NO-SYNC`;

        Mark k_o as `NO-SYNC`;

        end;

    For each node k in NOT-COEXEC[h] **do begin**

        Mark k_i as `DO-NOT-ENTER`;

        Mark k_o as `DO-NOT-ENTER`;

        end;

    Depth-first search the CLG starting at h_i,
     without traversing any edges into nodes
     marked `DO-NOT-ENTER`, and without
     traversing any sync edges into or out of
     nodes marked `NO-SYNC`;

    If a strong component including h_i was found

        return `FALSE`;

    end;

[5] The vectors described here each contain $|N|$ sets of sync graph nodes.

return `TRUE`;

One iteration of the main loop is required per node in the CLG. Each iteration requires $\mathcal{O}(|N_{CLG}| + |E_{CLG}|)$ to extract the strongly-connected components, and at most $\mathcal{O}(|N_{CLG}|)$ to mark the nodes. Total time for the algorithm is thus $\mathcal{O}(|N_{CLG}| \times (|N_{CLG}| + |E_{CLG}|))$.

Some extensions can be made to the above algorithm to eliminate more spurious cycles, with additional execution time penalties. These extensions include:

- Hypothesize *pairs* of head nodes r and s to test in the main loop. Eliminate sync edge traversals as before, but for both nodes. Return `FALSE` if the DFS returns a strong component containing r and s[6].
- Test possible *tail nodes* as well as head nodes. For a candidate head node h, a corresponding candidate tail node t would be chosen such that t is reachable by control flow edges from h and $t \notin$ COACCEPT$[h] \bigcup$NOT-COEXEC$[h]$. All nodes $r \in$ NOT-COEXEC$[h] \bigcup$NOT-COEXEC$[t]$ have their corresponding CLG nodes r_i and r_o marked `DO-NOT-ENTER`. Nodes r in SEQUENCEABLE$[h]$ have r_i marked `NO-SYNC`. There is no need to mark nodes in COACCEPT$[h]$, since we are hypothesizing that we exit the task through t. Again, we detect possible deadlock only upon finding a strong component containing both h and t.

  Note that hypothesizing tail nodes helps to eliminate spurious cycles only in cases where a hypothesized tail node is not co-executable with some other node in the cycle. (Tail nodes may be ordered with each other or with head nodes on a valid deadlock cycle; thus, information about tail node ordering does not help to eliminate spurious cycles.) When we hypothesized only the head nodes, the COACCEPT vector eliminated the case in which the wrong choice of a tail node might result in a spurious cycle.
- Combine the above two strategies, by hypothesizing multiple head-tail node pairs.
- For some specific number of tasks k, hypothesize k head-tail node pairs. If there is a deadlock, then either the deadlock cycle must join fewer than k tasks, or some set of k hypothesized pairs must be contained in a strong component. Cycles involving fewer than k tasks may be eliminated by searching the graph for them exhaustively.

These strategies form a spectrum of tradeoffs of accuracy versus execution time for deadlock cycle detection.

5 Stallability analysis.

In a program without conditional branches or loops, stallability can be detected statically in time $\mathcal{O}(|N|)$.

Lemma 3 *A program without conditional branches or loops is stall-free iff the numbers of signal and accept nodes are identical for all signal types.*

Proof: Consider a stall node r of type (t, m, s) in such a program. r is WAITING, and there are no nodes which are WAITING, READY, or NOT-SEEN which may rendezvous with r. Each node of type (t, m, s) which has rendezvoused before the execution wave was stalled has done so with a node of type $(t, m, \overline{s})$. Therefore, in the case of a stall, the number of nodes of type (t, m, s) in the program is greater than the number of nodes of type $(t, m, \overline{s})$. If the program executes to completion, the number of nodes of each type must be equal. □

[6] Note that deadlock cycles may involve as few as two head nodes, so it is not safe to extend this generate-and-test approach beyond two selected nodes without considering cycles of two nodes only.

Unfortunately, this implies a second lemma in the case where conditional branches exist in the program.

Lemma 4 *A program with conditional branches is stall-free iff, for all feasible linearized executions of the program, the numbers of signal and accept nodes are identical for all signal types.*

Proof: Lemma 3 must hold for any linearized version of the program. □

Proving programs stall-free thus requires detailed knowledge of their feasible executions. Enumerating feasible executions is combinatorially explosive in programs without loops, and subsumes the Turing halting problem if the program contains loops. Further, in view of the stringent requirements of Lemma 4, any safe approximation of stall detection would likely generate an unacceptably high number of false alarms. Despite this difficulty, programs can sometimes be transformed to reduce the complexity of stall analysis.

5.1 Automatic stall detection methods.

A programmer makes use of special knowledge about feasible executions to avoid stalls. Two common inference patterns allow moving rendezvous out of conditional branches.

In the first pattern, we might know that node r is always executed on one side of the branch and node r' of the same type is always executed on the other side of the branch. Thus, both nodes may effectively be combined into one node r'' which is unconditionally executed. The transformation should maintain relative node ordering, so that data dependencies between accept nodes and conditionals are preserved; conditionals are "split" to maintain these relations, and eliminated if all nodes are moved out of the conditional. (See Figure 5 (b) and (c).)

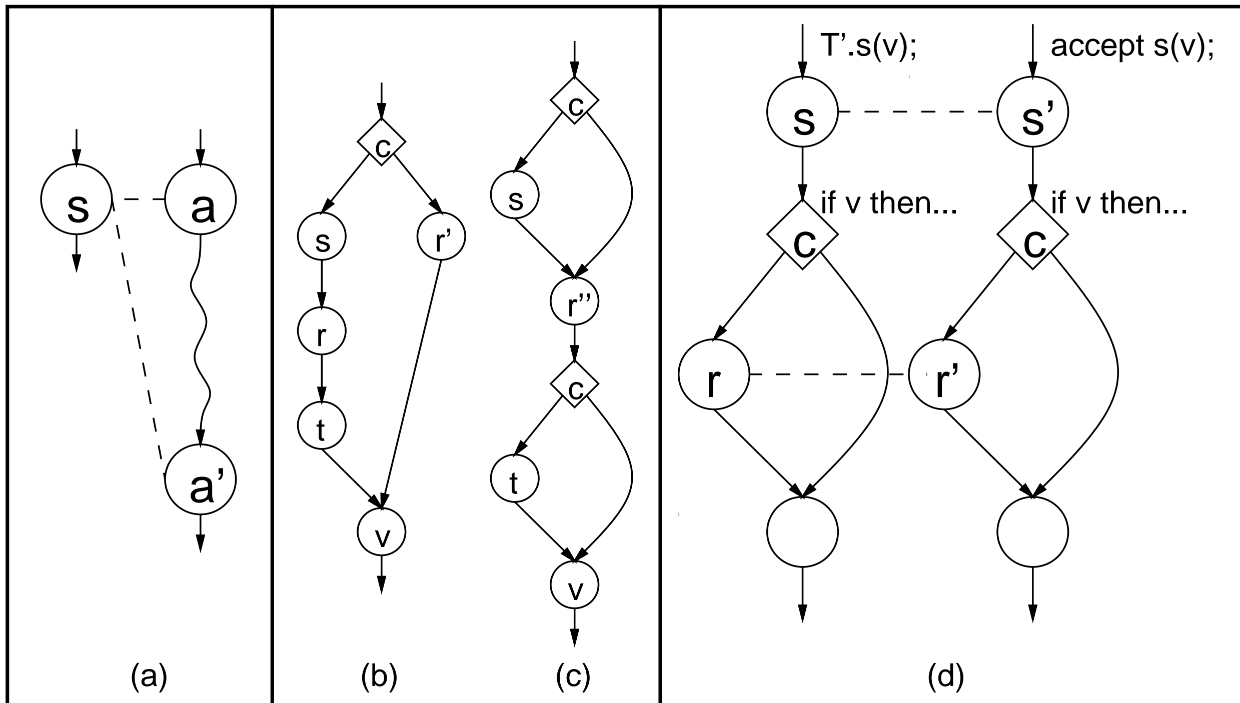

Figure 5: *Left:* Rendezvousing head nodes in a deadlock cycle (a). *Center:* Rendezvous which are executed in both the `then` and `else` parts of a conditional (b) and the transformation which merges them (c). *Right:* Rendezvous whose execution depends on the same condition (d).

In the second pattern, we know that node r in task T is executed iff a complementary node r' is executed in task T'. Thus, r and r' can be factored out of the count of nodes. If we do not use the `select` construct, then the co-dependence of r and r' can only come about through some communication between tasks. A simple example is shown in Figure 5(d). Here, a boolean variable `v` is passed from task T to T' by the rendezvous of s with s'[7]. Nodes r and r' are both conditionally executed when `v` is true. Since the same definition of v reaches both conditionals, when both conditionals are executed r will execute iff r' does. Thus, r and r' can be replaced by nodes outside their respective conditionals if such a dependence exists.

The second pattern is generally harder to establish than the first, since we must establish that different expressions in different tasks always evaluate equivalently. The general problem thus involves both finding reaching definitions and unifying expressions; we wish to avoid doing the latter because of its intractability. Two alternatives suggest themselves.

The first alternative is to allow the programmer to certify that two branches are co-dependent. While simple to implement, it places an additional burden on the programmer, and is unsafe if the programmer makes a mistake in the certification. The second alternative is to "encapsulate" conditional branch expressions as a single Boolean variable. Conditional branches containing rendezvous which cannot be removed by the first transformation would be required to use the encapsulated expressions as arguments. Encapsulated expressions could be communicated between tasks, but their values could never be changed. Thus, the problem of expression unification is eliminated.

6 Related work.

Much of the research on static analysis of rendezvousing tasks has focused on detection of race conditions [EP88]. Of particular relevance to this research is the work of Callahan and Subhlok [CS88], who present an $\mathcal{O}(|statements|^3)$ lattice algorithm for conservative approximation of race anomalies under a slightly different model of rendezvous than the one used here.

Avrunin et. al. [ADWR86] propose a representation of possible rendezvous sequences based upon "constrained expressions", formalized as regular grammars with additional nonregular constraints. Programs are proven deadlock-free by proving that they do not generate rendezvous sequences which lead to deadlock. Dillon [Dill90] presents another interesting technique based on semiautomatic proofs of correctness. Her model represents programs as forests of "symbolic execution trees"; freedom from deadlock is proven by proving a set of assertions that deadlocked execution waves are infeasible. Neither paper claims tractability or full automation of the proof algorithm.

Taylor [Tayl83a] represents the possible *concurrency states* of a program as a graph. A concurrency state represents the rendezvous status of all tasks within a program. Thus, the number of concurrency states is greater than the product of the numbers of rendezvous nodes in each task. Later work by Young and Taylor [YT88] apply information gathered during symbolic execution to rule out infeasible concurrency states. Long and Clarke [LC89] propose a similar *task interaction concurrency graph* representation, and cite empirical evidence of a linear reduction in the number of states with respect to the concurrency state graph. McDowell [McDo89] builds a *reduced concurrency history graph* structure by aggregating related concurrency history states into "clans"; the resulting graph may still exceed polynomial size in the worst case.

Work specifically concerning static deadlock detection in parallel programs dates to [Saxe77] for semaphore-based communication and [Apt83] for CSP. More recently, Murata et.al. [MSS89] proposed a method for Ada deadlock detection based

[7] For clarity in the example, the expression argument of the signal statement is a variable with the same name as the variable parameter of the accept statement. This need not generally be the case.

on a Petri-net representation of possible rendezvous. Although the authors of [MSS89] do not provide complexity analysis, the execution time of their "inconsistency" deadlock detection algorithm is clearly proportional to the size of the powerset of rendezvous statements in the program.

7 Conclusions.

We have shown here that all infinite-wait anomalies can be divided into *deadlocks* and *stalls*. We have defined the sync graph and its feasible execution waves that embody all possible program executions for programs with Ada-like synchronizations. We have identified four constraints on the sync graph which must apply to some feasible execution wave for a program to exhibit deadlock. Deadlocks satisfying only constraint 1 can be found in time linear in the size of the program and exponential in loop nest depth by our first detection algorithm which conservatively identifies potential deadlock in programs with cyclic control flow. Finding whether there are deadlocks satisfying constraints 1 and 2, or 1 and 3a, is NP-hard; proofs presented in [MR90] are based on 3-satisfiability. Our second deadlock detection algorithm is a refinement of the first, and eliminates some spurious deadlocks using an approximation of constrains 2 and 3a, but at an increase in cost. We have outlined modifications which further increase precision at the cost of additional execution time. We have shown that efficient stall detection is not possible in programs with independent conditional branches (including loops), but that the use of intertask data dependencies can allow us to disprove stalls in a limited number of situations.

Acknowledgements. We wish to acknowledge the valuable insights and assistance of our colleagues B. R. Badrinath, William Landi, and Thomas J. Marlowe.

References

[Apt83] Apt, K. R.
"A static analysis of CSP programs."
Lecture Notes in Computer Science #164 - Logics of Programs (Proceedings, 1983) Springer-Verlag, pp. 1-17.

[ADWR86] Avrunin, G.S., Dillon, L. K., Wileden, J. C., and Riddle, W. E.
"Constrained expressions: adding analysis capabilities to design methods for concurrent software systems."
IEEE Trans. on Software Eng., February 1986, 278-291.

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

[Dill90] Dillon, L. K.
"Verifying general safety properties of Ada tasking programs."
IEEE Trans. on Software Eng., January 1990, 51-63.

[EP88] Emrath, P. A. and Padua, D. A.
"Automatic detection of nondeterminacy in parallel programs."
SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, 1988, 89-99.

[CD73] Coffman, E. O. and Denning, P. J.
Operating systems theory.

Prentice-Hall, 1973.

[Hech77] Hecht, M. S.

"Flow analysis of computer programs."

Elsevier North-Holland, 1977.

[Knut71] Knuth, D. E.

"An empirical study of FORTRAN programs."

Software Practice and Experience 1, 1971, 105-133.

[LC89] Long, D. L. and Clarke, L. A.

"Task interaction graphs for concurrency analysis."

Eleventh International Conf. on Software Engineering, 1989, 44-52.

[McDo89] McDowell, C. E.

"A practical algorithm for static analysis of parallel programs."

Journal of Parallel and Distributed Computing **6**, 1989, 515-536.

[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.

[MSS89] Murata, T., Shenker, B., and Shatz, S. M.

"Detection of Ada static deadlocks using Petri net invariants."

IEEE Trans. on Software Eng., March 1989, 314-326.

[Saxe77] Saxena, A.

"Static detection of deadlocks."

CU-CS-122-77, Dept. of Computer Science, University of Colorado at Boulder, 1977.

[Tayl83a] 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.

[YT88] Young, M. and Taylor, R. N.

"Combining static concurrency analysis with symbolic execution."

IEEE Trans. on Software Eng., October 1988, 1499-1511.

A Deadlock complexity proofs.

Theorem 2 *Detecting deadlock cycles which are valid under constraints 1 and 3a in the sync graph of a program is NP-hard.*

Proof method: Reduction to 3-satisfiability.

Proof: Given an m-term conjunction $C_1C_2...C_m$, where each clause C_i is a disjunct of three literals $(L_i^1 + L_i^2 + L_i^3)$, and each literal L_i^j is a variable $v_k, 1 \leq k \leq n$ or its negation. Construct the following program[8], as shown in Figure 6.

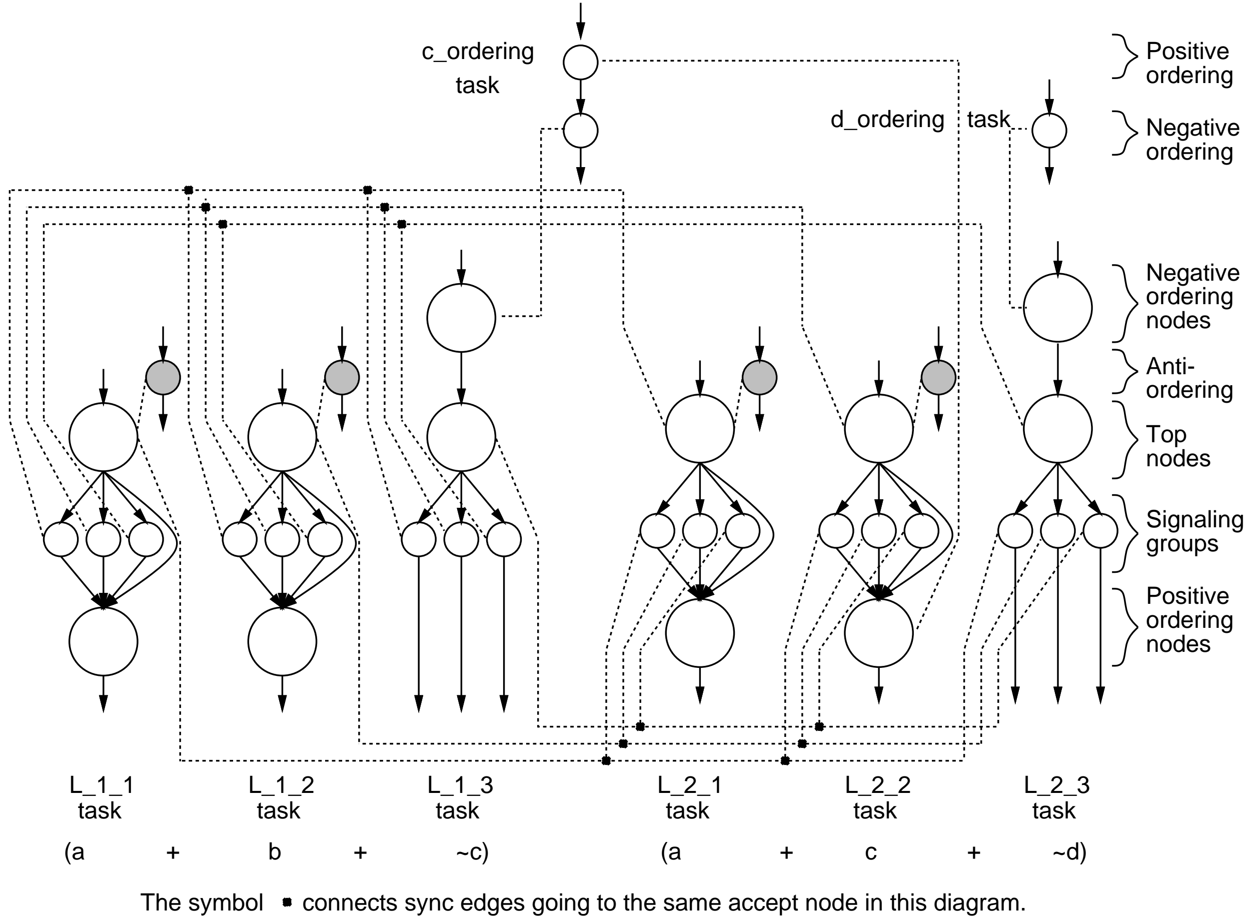

Figure 6: Sync graph of a program constructed corresponding to the 3-CNF expression $(a + b + \sim c) \times (a + c + \sim d)$.

- For each literal L_i^j corresponding to a positive variable v_k, construct the task of Figure 7 (a). Let $q = (i \mod m) + 1$. Task `L_i_j` corresponds to literal L_i^j, and contains five nodes. The first node, `accept s_i_j`, accepts a signal from a task corresponding to the previous conjunct $C_{(i-1) \bmod m}$, or from the *anti-ordering* task `a_i_j`. The anti-ordering task (and the control structure of `L_i_j`) prevents negative literals from being ordered with respect to each other, or with respect to positive literals that do not correspond to the same variable.

  The three nodes `L_q_1.s_q_1` through `L_q_3.s_q_3` form a *signaling node group*. One of these nodes may be executed, based on a random boolean value. Thus, the control flow path through `L_i_j` actually executed cannot be determined statically. The three nodes of the signaling node group create sync edges from L_i^j to each top node in the tasks created for C_q.

[8] The program generated by this procedure is not necessarily stall free. This is not of concern within the proof, since we are considering only the problem of detecting constrained deadlock cycles.

The last node in the task, `v_k_ordering.v_k_positive_i_j`, is used to insure that the top node for `L_i_j` is executed before the top node of any task corresponding to the negated variable $\sim v_k$. (Recall that $L_j^i = v_k$.) This node is used to establish an ordering between literal nodes corresponding to positive and negated instances of the same variable, and is called an *order-sending* node.

(a)
```
task body L_i_j is
begin
  accept s_i_j;
  if randomBoolean() then
    if randomBoolean() then
      L_q_1.s_q_1;
    else if randomBoolean() then
      L_q_2.s_q_2;
    else if randomBoolean() then
      L_q_3.s_q_3;
    end if;
  end if;
  v_k_ordering.v_k_positive_i_j;
end L_i_j;


task body a_i_j is
begin
  L_i_j.s_i_j;
end a_i_j;
```

(b)
```
task body L_i_j is
begin
  v_k_ordering.v_k_negative;
  accept s_i_j;
  if randomBoolean() then
    L_q_1.s_q_1;
  else if randomBoolean() then
    L_q_2.s_q_2;
  else if randomBoolean() then
    l_q_3.s_q_3;
  end if;
end L_i_j;
```

(c)
```
task body v_k_ordering is
begin
  accept v_k_positive_i_j;  -- Repeat for all i and j
    .                       -- where L_i_j is positive
    .                       -- variable v_k - omit if
    .                       -- no such literal
  accept v_k_negative;
end v_k_ordering;
```

Figure 7: Templates for positive literal and anti-ordering (a), negative literal (b), and ordering (c) tasks.

- For each literal L_i^j corresponding to a negated variable $\sim v_k$, construct the task of Figure 7 (b). The tasks corresponding to negated literals differ only in that the order-sending nodes are placed at the beginning of the task and send the signal `v_k_ordering.v_k_negative`.

The tasks corresponding to the literals of an entire clause form a *clause task group.* There are sync edges from the signaling node group in each task to all top nodes of the tasks in the next clause task group.

- For each variable v_k for which negative literals exist, create the *ordering task* of Figure 7 (c). The ordering tasks force all negative top nodes corresponding to a variable to execute after all positive top nodes for the same variable. The nodes of an ordering task are called *order-accepting* nodes.

Each literal task is connected via sync edges from the signaling node group to the top nodes of all literal tasks in the next clause task group. The top nodes of each literal task `L_i_j` are ordered with respect to other top nodes iff they correspond to positive and negated instances of the same variable. To prove this, consider the cases where two top nodes a and b may be ordered. Either both are negative, both are positive, or one is negative and the other positive. If both are positive, they cannot be ordered since they are free to start when the program starts.

Both positive: Top nodes of positive tasks cannot be ordered relative to each other, since all are free to execute at the start of the program.

Both negative: See Figure 8. The notation $a < b$ means that a always finishes before b starts, and $a = b$ if both nodes necessarily finish at the same time. $a \leq b$ if one or the other of these two cases is always true[9]. Without loss of generality, assume $a < b$. Since $a < b$, it must also be true that $a \leq b_i$ for all immediate control flow predecessors b_i of b; otherwise some

[9]Note: $a \not< b$ does not imply $b \leq a$, nor does $a \not\leq b$ imply $b < a$.

predecessor would be free to complete execution and allow b to start before a finished. Node b has only one such predecessor: the order-sending node b_0. $a < b_0$ is not possible, since b_0 is free to start when its task starts. Therefore $a \leq r_b$ for all nodes r_b which make a rendezvous with b_0. There is only one such node: the negative order-accepting node for the variable v associated with b. Recall that r_b is the last node in the ordering task, since b corresponds to $\sim v$. r_b is preceded by all order-accepting nodes for positive literals for v. So some set of control flow predecessors of r_b must not be able to make a rendezvous until after a has completed. Consider such a predecessor r_c, which has a rendezvous with positive order-sending node c_1. c_1 can execute if anti-ordering node c_a makes a rendezvous with c and the control flow path from c to c_1 is taken. This sequence does not depend on any event in a's task. Therefore, there is always a way for b to start execution before a finishes, and $a \not< b$.

One negative, one positive: Let a be the positive top node and b be the negative top node. $a < b$ if they correspond to the same variable, since this is forced by the order-sending nodes and ordering task for the variable. $b \not< a$ under any conditions, since a is free to start at the beginning of the task. If a and b do not correspond to the same variable, the proof that they are not ordered is similar to that for the case of two negative top nodes.

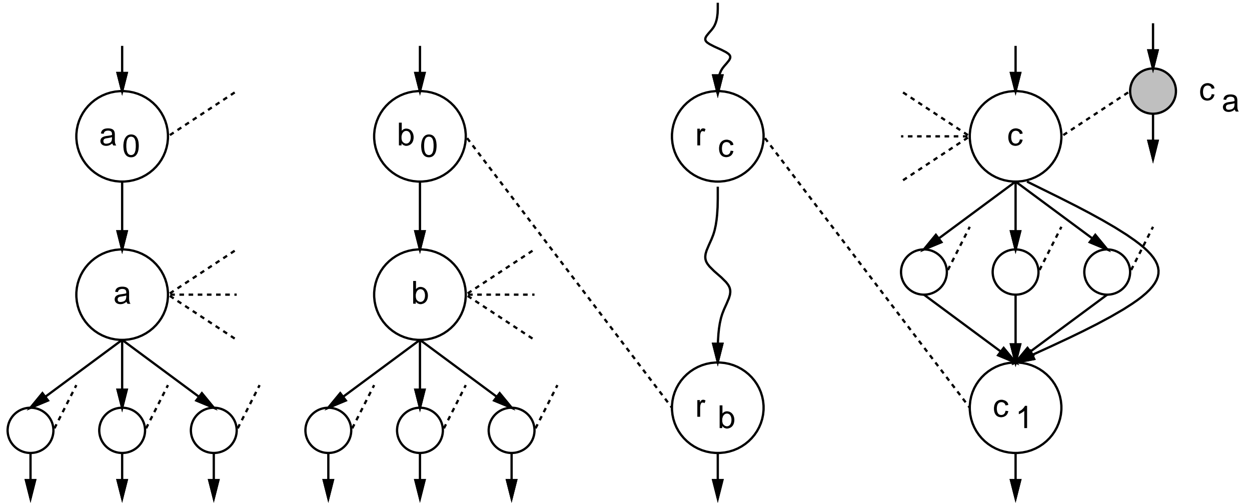

Figure 8: Ordered pair of negative top nodes.

NP-hard: Consider the possible cycles in the sync graph of the program. Any cycle through an ordering task must enter it at a node r and exit it to an order-sending node a_{ord}. Thus, r and a_{ord} are head nodes in the cycle and $r < a_{ord}$. Thus, any deadlock cycle involving an ordering task has a pair of ordered head nodes, and is invalid under constraint 3a.

Since each anti-ordering task is connected to only one positive top node, the anti-ordering tasks do not participate in cycles. Any valid deadlock cycles must therefore involve only the sync edges between literal tasks. These sync edges go from the signaling node group of each literal task in a clause group to all top nodes of tasks in the next clause task group; sync edges similarly go from the last to the first clause task group. No sync edges go between tasks in the same clause task group. Therefore, any deadlock cycle with no sequenceable head nodes contains a top node of at least one literal task in each clause group as a head node. Further, given one literal from each clause, a cycle through the literal tasks exists corresponding to those literals.

A setting of the literal values in the conjunct will be consistent iff no pair of literal values correspond to positive and negated instances of the same variable. A cycle through the literal tasks will correspond to a feasible deadlock iff there are no two literal tasks whose top nodes are sequenceable. The top nodes of two literal tasks are sequenceable iff the literal tasks correspond to positive and negated instances of the same variable. Therefore there is a feasible deadlock in the program iff the conjunct is satisfiable. Thus, if we demand that head nodes on a deadlock execution wave not be sequenceable, approximating deadlock detection is NP-hard. □

Theorem 3 *Given an arbitrary sync graph, detecting a deadlock cycle which is valid under constraints 1 and 2 is NP-complete.*

Proof outline: Similar to Theorem 2. Given a 3-CNF expression, construct a sync graph such that a deadlock cycle without head nodes connected by sync edges exists iff the expression is satisfiable. Construct a single task for each literal as before; the ordering nodes (and control flow edges from the top node to them) are omitted. Insert a sync edge between the top nodes of any pair of tasks for positive and negative literals for the same variable. These edges cannot add new deadlock cycles; any cycle using such an edge would have to enter and exit a top node through sync edges, which is prohibited under constraint 1b.

A deadlock cycle without connected top nodes corresponds to an instantiation of the variables which satisfies the expression. Such a cycle exists if the 3-CNF expression is satisfiable.

Given a deadlock cycle proposed by some algorithm, its head nodes can be readily identified and checked to see if any pair is connected by a sync edge. The problem is therefore in class NP. □

Note that the sync graph constructed in this proof cannot in general correspond to an actual program. In particular, consider chains of sync edges in the sync graph. If a program is translated into a sync graph, one end of each sync edge in such a chain must be in the accepting task for the edge's signal type. The sync graph of the proof does not observe this restriction.