Generating Efficient Distributed Deadlock Avoidance Controllers [*]

César Sánchez   Henny B. Sipma   Zohar Manna

Stanford University
Computer Science Department
Stanford, CA 94305 USA
E-mail: {cesar,sipma,zm}@CS.Stanford.EDU

Abstract

General solutions to deadlock avoidance in distributed systems are considered impractical due to the high communication overhead. In previous work we showed that practical solutions exist when all possible sequences of resource requests are known a priori in the form of call graphs; in this case protocols can be constructed that involve no communication. These run-time protocols make use of annotations of the call graph that are computed statically based on the structure of the call graph. If the annotations are acyclic, then deadlocks are unreachable.

This paper focuses on the computation of these annotations. We first show that our algorithm for computing acyclic annotations is complete: every optimal annotation can be generated. We then show that, given a cyclic annotation and a fixed set of resources, checking whether deadlocks are reachable is NP-complete. Finally, we consider the problem of computing minimal annotations that satisfy given constraints on the number of available resources. We show that the problem is NP-complete in the general case, but that it can be solved in polynomial time if the only restrictions are that the number of certain resources is 1, that is, these resources are binary semaphores.

Keywords: Scheduling, Deadlock Avoidance, Distributed Algorithms

[*]This research was supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, and by NAVY/ONR contract N00014-03-1-0939.

1. Introduction

Computations in distributed real-time and embedded systems (DRE) involve a distribution of method calls over multiple sites. At each site these computations need resources to proceed. Since resources are limited (such as a fixed-size pool of threads or locks protecting mutually exclusive regions), and multiple processes can be spawned at different sites deadlocks may arise [1, 2]. Traditionally three techniques are used to deal with deadlocks: detection, prevention and avoidance.

Deadlock detection is an optimistic method for concurrency control [3], where deadlocks are detected at run-time and corrected by, for example, the roll-back of transactions. This approach is common in databases, but in embedded systems it is usually not applicable, especially in systems interacting with physical devices. *Deadlock prevention* is a pessimistic method, that ensures statically that one of the necessary conditions for deadlock is broken. Monotone locking [4], widely used in practice, imposes a fixed total order in which resources are acquired. This strategy, however, imposes some burden on the programmer, and—often a more important concern in DRE systems—can substantially reduce performance, by artificially limiting concurrency.

Deadlock avoidance methods take a middle route. A run-time protocol implements a *resource allocation controller* that decides whether to grant a request based on resource availability and possible future requests. A resource is granted only if it is *safe*, that is, if the controller has a strategy such that all processes can complete. To make this possible, when a process enters the system it must inform the protocol about its resource utilization. The first algorithm following this approach was Dijkstra's Banker's algorithm [5, 6, 7] where each process reports the maximum number of resources that it can request during its execution.

This information is later used to decide, together with current utilization, to grant or deny requests. When resources are distributed across multiple sites, however, this approach to deadlock avoidance is harder, because different sites may have to consult each other to determine whether a particular allocation is safe. Because of this need for distributed agreement, a general solution to distributed deadlock avoidance is considered impractical [8]; the communication costs involved simply outweigh the benefits gained from deadlock avoidance over deadlock prevention.

In previous research [9] we have demonstrated that deadlock avoidance without communication (using only operations over local data to decide requests) is possible by providing additional information about resource usage, in the form of call graphs that represent all possible sequences of remote invocations that processes can perform. This technique is applicable to distributed systems in which processes perform method invocations at different sites and lock local resources until all remote calls have returned. In particular, if the chain of remote calls arrives back to a site previously visited a new resource is needed. This model arises, for example, in DRE architectures that use the *WaitOn-Connection* policy for nested up-calls [10, 11, 12]. In DRE systems, the call graphs can be extracted from the component specifications or from the source code directly by static analysis.

Our approach is based on annotations of the call-graph vertices, computed statically, and a run-time protocol. In [9] we showed that if the annotation is *acyclic* deadlocks are not reachable. If the annotation is cyclic deadlocks may not be reachable for a given set of resources. We have shown [13], however, that given sufficient resources, a deadlock is always reachable with a cyclic annotation[1]. In this paper we show that checking whether deadlock is reachable for a given fixed number of resources and a cyclic annotation is NP-complete. This new result adds support to the design principle to construct protocols based on acyclic annotations only.

Yet, to minimize resource requirements and maximize resource utilization, we want minimal annotations. In this paper, we show that the algorithm presented in [13] for computing acyclic annotations is complete: it can generate all minimal annotations (minimal annotations are not unique). This result is then used to determine the complexity of computing acyclic annotations when some constraints are imposed on the total number of resources available. For arbitrary constraints on the number of the different resources, checking whether an acyclic annotation exists is NP-complete. If, however, the constraints on the number of resources consist exclusively of constraints that state that the number of certain resources is one, then checking for the existence of an acyclic annotation can be performed in polynomial time.

These results emphasize the importance of accurate specifications and static analysis. The more specific the call graph, the easier it is to generate acyclic annotations, and fewer resources are needed to ensure deadlock free operation.

The rest of this paper is structured as follows. Section 2 introduces the computational model and recalls the basics of our deadlock avoidance algorithms [9, 13]. Section 3 shows that checking deadlock reachability for cyclic annotations is hard. In Section 5 we prove that the algorithm proposed in [13] is complete. Section **??** studies how to generate acyclic annotations given constraints on the resources. Section 6 concludes.

2. Model of Computation

We model a distributed system $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ as a set of *sites* and a *call graph* specification. Sites $\mathcal{R} : \{r_1, \ldots, r_{|\mathcal{R}|}\}$ model distributed devices that perform computations and handle a necessary and scarce local resource, for example a finite pool of threads or execution contexts. A call graph specification $\mathcal{G} : (V, \rightarrow, I)$ is an acyclic graph that captures all the possible flows of the computations. A call graph vertex $n = (f\!:\!r)$ models the method call f to be performed at site r (if the method name is unimportant we simply write $n : r$). An edge from $n = (f\!:\!r)$ to $m = (g\!:\!s)$ denotes a possible remote invocation of method g at site s. Vertices in I denote initial methods that processes can execute when spawned. In the remainder of this paper we will use $r, s, r_1, r_2, \ldots$ to refer to sites and $n, m, n_1, m_1, \ldots$ to refer to call graph vertices. Each site r stores some local data, including a constant T_r representing the total units of resource in r, and a variable t_r whose value represents the available resources at each point in time. Initially, $t_r = T_r$.

The execution of a system consists of processes, which can be created dynamically, executing computations that only perform remote calls according to the edges in the call graph. When a new process is spawned it announces the initial call graph vertex (from the set I) whose outgoing paths describe the remote calls that the process may perform. All invocations of a call graph vertex require a new resource in the corresponding site, while call returns release the resource. Hence,

[1]This resembles the Belady anomaly [14] in Operating Systems in the sense that a system can be deadlock free, but providing more resources can make the system have reachable deadlocks.

a process running a method n locks its resource at least until all the remote invocations initiated have returned.

There is no restriction on the topology of the call graph or on the number of process instances, and thus deadlocks can be reached if all requests for resources are immediately granted.

Example 1. *Consider a system with two sites $\mathcal{R} = \{r, s\}$, a call graph with four nodes $V = \{n_1, n_2, m_1, m_2\}$, and edges*

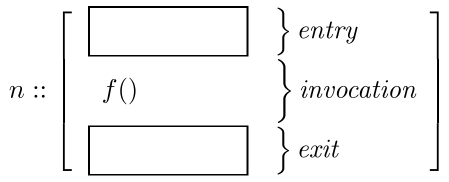
$$\begin{tikzcd}
\boxed{n_1\ \ r} \arrow[r] & \boxed{n_2\ \ s} \\
\boxed{m_1\ \ s} \arrow[r] & \boxed{m_2\ \ r}
\end{tikzcd}$$

If sites s and r each handle exactly two resources ($T_r = T_s = 2$) and four processes are created, two running n_1 and two running m_1, no more resources are available after each process starts its execution. Hence, the resulting state is a deadlock since none of the processes can proceed. □

To avoid deadlocks a protocol must be provided that restricts access to the resources. Our deadlock avoidance algorithms consist of two parts:

1. The offline computation of call-graph *annotations*, $\alpha : V \mapsto \mathbb{N}$, a map from call-graph nodes to natural numbers;
2. A run-time *protocol* that controls resource allocations and deallocations based on local data and call-graph annotations.

The protocol consists of two stages: one that runs when the resource is requested, and another that executes upon release. A schematic view of a protocol is shown below:



A process that is granted access into the method section is called *active*, while a process whose request is rejected is called *waiting*. We assume that the actions of the entry and exit sections of a protocol cancel each other, and that the successful execution of an entry section cannot help a waiting process to obtain its desired resources.

Intuitively, the annotation $\alpha(n : r)$ provides a measure of how many execution contexts site r should reserve for processes executing at other sites that may perform remote calls to r with lower annotations.

```
n :: [ [ when α(n) < t_r do ]
     [ [   t_r--             ]
     [
     [   f()
     [
     [   t_r++                 ]
```

Figure 1. BASIC-P

Thus, the annotation provides a static data structure that can be used by a protocol to ensure at run-time that there will be no cyclic dependencies between processes waiting for resources.

The simplest protocol based on this annotation is BASIC-P, shown in Figure 1. It grants a resource to a process executing a call graph node $n : r$ if $\alpha(n)$ is less than the number of resources available, represented by the local variable t_r. In previous papers we proved that this protocol avoids deadlock if the annotation is acyclic in the following sense. Given a system $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ and an annotation α, the annotated call graph $(V, \rightarrow, \dashrightarrow)$ adds to $\mathcal{G}$ one edge $n \dashrightarrow m$ for every pair of nodes n and m that reside in the same site and $\alpha(n) \geq \alpha(m)$. A node n depends on a node m, represented as $n \succ m$, if there is a path in the annotated graph from n to m that follows at least one $\rightarrow$ edge. The annotated graph is acyclic if no node depends on itself, in which case we say that the annotation is acyclic.

Theorem 1 (Annotation Theorem for BASIC-P [9])**.** *Given a system $\mathcal{S}$ and an acyclic annotation, if* BASIC-P *is used to control resource allocations then all executions of $\mathcal{S}$ are deadlock free.*

Example 2. *Reconsider the system described in Example 1. By granting resources whenever they are available, it implicitly assumes the following annotation graph,*

$$\begin{tikzcd}
\boxed{n_1\ \ r}^{0} \arrow[r] \arrow[dr, dashed, <->] & \boxed{n_2\ \ s}^{0} \\
\boxed{m_1\ \ s}^{0} \arrow[r] \arrow[ur, dashed, <->] & \boxed{m_2\ \ r}^{0}
\end{tikzcd}$$

in which all nodes have annotation 0. Clearly, this graph has a dependency cycle, and thus deadlock avoidance is not guaranteed. If we set $\alpha(m_1) = 1$, this dependency cycle is eliminated, resulting in the acyclic annotated call graph,

$$\begin{tikzcd}
\boxed{n_1\ \ r}^{0} \arrow[r] \arrow[dr, dashed, <->] & \boxed{n_2\ \ s}^{0} \\
\boxed{m_1\ \ s}^{1} \arrow[r] \arrow[ur, dashed] & \boxed{m_2\ \ r}^{0}
\end{tikzcd}$$

Deadlock is avoided by always reserving at least one *resource in site s for a remote call from a process P executing node n_1, so that P can complete.* □

In [9, 15] we showed that there is a spectrum of protocols that are all based on annotations. For all of them the maximum annotations labeling call-graph vertices determines the minimum number of resources the corresponding component must provide. Moreover, lower annotations allow more concurrency for a given set of resources. Thus our objective is to determine minimal acyclic annotations

Problem Statement Given a system specification $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ let $\langle \mathcal{S}, \alpha, \mathcal{T} \rangle$ denote the system $\mathcal{S}$ along with an annotation α and configuration of resources $\mathcal{T} : \{T_r = k_r\}_{r \in \mathcal{R}}$. In the next three sections we study the following problems:

1. $\langle \mathcal{S}, \alpha, \mathcal{T} \rangle$: For a given system $\mathcal{S}$ with cyclic annotation α and a given set of resources, does α guarantee absence of deadlock?
2. $\langle \mathcal{S}, ?, ? \rangle$: For a given system with no constraints on the configuration of resources, compute an acyclic annotation and determine the minimum resources required.
3. $\langle \mathcal{S}, ?, \{T_r = k_r\}_{r \in X} \rangle$: For a given system with the given constraints on the number of resources in the set $X \subseteq \mathcal{R}$, determine whether there exists an acyclic annotation.

3. Deciding Deadlock Reachability

In this section we show that checking deadlock reachability for a fixed number of resources when the annotation is cyclic is computationally hard. We first present a nondeterministic algorithm that decides in polynomial time whether a system has reachable deadlocks. Then, we introduce a reduction from 3-*CNF* to deadlock reachability that proves that the problem is NP-hard. In summary:

Lemma 2 (Finding Deadlocks). *Given a system $\mathcal{S}$, cyclic annotation α, and assignment of resources $\mathcal{T} : \{T_r = k_r\}_{r \in \mathcal{R}}$, there exists a non-deterministic algorithm that decides in polynomial time whether a deadlock is reachable.*

Proof. First, we say that a process is "relevant" in an execution if it is granted some resource. It is clear that if there is a run to a deadlock, then there is a run where only relevant processes exist; moreover, there is a run where only allocations are performed. From [16] we know that every reachable state of the system can be reached by Basic-P following only a series of allocations (with no deallocation). Moreover, there is a topological total order $<$ on the call-graph vertices such that every reachable state can be reached by allocations performed following that order (all allocations for a method are performed before any allocation for any lower method).

We construct an algorithm that guesses a deadlock state σ, and then guesses an order $<$ such that Basic-P reaches σ following $<$. The run is constructed by merging all allocations of a single method into a macro step (more than one process acquires a resource at the same node).

Given a system specification $\langle \mathcal{S}, \alpha, \mathcal{T} \rangle$ a state of the algorithm is a vector $\langle p_1, \ldots, p_{|V|} \rangle$, where entry p_j represents the number of processes active in call-graph method j. A macro step is represented by $\vdash^n_j$, corresponding to n processes gaining access to method j:

$$\langle p_1, \ldots, p_j, \ldots, p_{|V|} \rangle \vdash^n_j \langle p_1, \ldots, p_j + n, \ldots, p_{|V|} \rangle$$

where the only entry modified is p_j. A run is a sequence $\sigma_1 \vdash^{n_1}_{j_1} \sigma_2 \vdash^{n_2}_{j_2} \ldots \vdash^{n_k}_{j_k} \sigma_k$ of macro steps. A run is legal if σ_1 is $\langle 0, \ldots, 0 \rangle$, and if every state σ_{i+1} is obtained from σ_i by a legal (macro)allocation. It is easy to establish that a macro allocation $\vdash^{n_i}_{j_i}$ is legal, by checking that:

1. the enabling condition of Basic-P for method j_i holds in σ_i for all the n_i processes,
2. the only entry modified is p_{j_i} which is increased in exactly n_i units, and
3. the parent node j of node j_i satisfies $p_j \geq p_{j_i} + n_1$, i.e., there are enough caller processes to perform all the remote calls.

A legal run follows a total order $<$ if all the steps are carried out in $<$ order: $j_i < j_{i+1}$ for all i. This implies that the maximum length of a run that follows some total order is $|V|$.

The state of the algorithm can be encoded in size linear in the specification, each step can be checked in linear time, and the final state being a deadlock can also be checked in linear time, as desired. □

Lemma 3 (Deadlock Reachability). *Given a system $\mathcal{S}$, cyclic annotation α and assignment of resources $\mathcal{T} : \{T_r = k_r\}_{r \in \mathcal{R}}$, deciding whether a deadlock is reachable is NP-hard.*

Proof. The proof proceeds by reducing 3-*CNF* to deadlock reachability. Given a 3-*CNF* formula φ, we create a system specification

$$\langle \mathcal{R}, \mathcal{G}, \alpha, \{T_r = 1\}_{r \in \mathcal{R}} \rangle$$

whose size is linear in the size of the formula, and which has deadlocks reachable if and only the formula is satisfiable. We use C_j for the clauses in φ and X_i for its variables.

Sites: $\mathcal{R}$ includes one site c_i per clause C_i and one site r_j per variable X_j:

$$\mathcal{R} \overset{\text{def}}{=} \{c_i\} \cup \{r_j\}.$$

Methods: There are two call-graph methods per variable, x_j and $\overline{x_j}$, both running in site r_j. Similarly, for every clause C_i there are two methods a_i and b_i, both residing in the corresponding site c_i:

$$\begin{aligned} V = &\{(x_j\!:\!r_j), (\overline{x_j}, r_j) \mid \text{for every variable } X_j\} \cup \\ &\{(a_i\!:\!c_i), (b_i\!:\!c_i) \mid \text{for every clause } C_i\} \end{aligned}$$

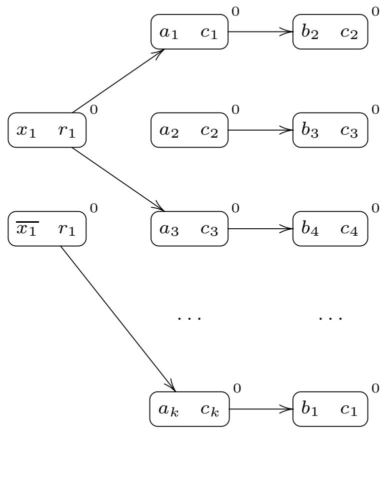
Resources: The total number of resources is $T_{r_i} = 1$ for all variable sites and $T_{c_j} = 1$ for all clause sites.

Annotations: The annotation of every node is 0, the only possible value. Consequently, since $T_r = 1$, there can be at most one active process running every method.

Edges: Each method a_i is connected to the method b_{i+1} of the next clause including, $a_k \to b_1$ for the last clause. Given that all annotation values are 0, this immediately creates a cycle in the annotated graph:

$$a_1 \to b_2 \dashrightarrow a_2 \to b_3 \cdots a_k \to b_1 \dashrightarrow a_1.$$

We also add an edge between a variable x_i to all the clauses where X_i appears in positive form and one edge between $\overline{x_i}$ to all the clauses where X_i it appears in negative form. For example, if X_1 appears in clauses C_1 and C_3, and $\overline{X_1}$ appears in clause C_k, the call-graph will include:



Since, for all variables, the call-graph methods x_i and $\overline{x_i}$ reside in the same site and the annotation is 0, in any execution, at most one of them can have an active process. This corresponds to picking a valuation for the variable X_i. Then, the only clause methods that can have active processes are those with some process in a predecessor node: this corresponds to a clause being satisfied. Therefore, there is a run to a deadlock (exercising the only cycle in the graph), if and only if all the clauses can be satisfied. □

Theorem 4. *Deciding whether $\langle \mathcal{S}, \alpha, \mathcal{T} \rangle$ has reachable deadlocks, where α is cyclic, is NP-complete.*

4. Generating All Minimal Annotations

We demonstrate a complete algorithm to generate minimal acyclic annotations. An acyclic annotation is minimal if no annotation value of any node in the call graph can be reduced without creating cycles; it follows that reducing more than one value also creates cycles. The algorithm is complete in the sense that it can generate every acyclic annotation.

Fig. 2 shows an algorithm that computes minimal acyclic annotations for a system $\langle \mathcal{R}, \mathcal{G} \rangle$. It is parametrized by a reverse topological order of the call-graph nodes. This order is followed to calculate the value of the annotations, which ensures that when calculating $\alpha(n)$, all descendants of n have been calculated.

```
 1: {Order $N$ in reverse topological order}
 2: {Let $\text{Reach}_X = \{m \mid x(\to \cup \dashrightarrow)^* m, x \in X\}$}
 3: {Let $\text{Site}_n = \{m \mid n \equiv_{\mathcal{R}} m\}$ }
 4: for $n = n_1$ to $n_{|N|}$ do
 5:     $R = \{n\}$
 6:     repeat
 7:         $R \leftarrow \text{Reach}_R$
 8:     until fix-point
 9:     if $R \cap \text{Site}_n$ is empty then
10:         $\alpha(n) = 0$
11:     else
12:         $\alpha(n) = 1 + \max\{\alpha(m) \mid m \in R \cap \text{Site}_n\}$
13:     end if
14: end for
```

Figure 2. CalcMin**: Algorithm for computing minimal acyclic annotations**

The algorithm can generate every acyclic annotation, simply by providing the right order. This is shown by, given a minimal acyclic annotation α, defining an order $<_\alpha$ such that CalcMin generates α.

Lemma 5. *Every minimal acyclic annotation can be produced by* CalcMin.

Proof. Given a minimal acyclic annotation α, let $<$ be a reverse topological order such that:

1. if $\alpha(n) > \alpha(m)$ and $n \equiv_{\mathcal{R}} m$ then $n > m$, and
2. if $n \succ m$ then $n > m$.

There is one such order since the annotation is acyclic. We show by induction on $<$ that CalcMin$(<)$ generates an acyclic minimal annotation β with $\beta(n) = \alpha(n)$ for every node n. Let n be an arbitrary node. Clearly, all nodes m in Reach$_R$ at step 8 satisfy $n \succ m$, so $\beta(m) = \alpha(m)$ by the inductive hypothesis. Then, $\beta(n) \leq \alpha(n)$ at line 12 since otherwise there would be cycles in α. Assume, by contradiction, that $\beta(n) < \alpha(n)$. In this case replacing $\alpha(n)$ by $\beta(n)$ makes α still an acyclic annotation which contradicts the minimality of α. □

5. Resource Constraints

Often constraints are imposed on the number of resources available in certain sites. Since the algorithm presented in the previous section generates all minimal acyclic annotations, this algorithm immediately provides a decision procedures for the question whether an acyclic annotation exists that accommodates these constraints. Indeed we can guess a reverse topological order $<$, generate the annotation α with CalcMin, $(<)$ and check whether α satisfies $\mathcal{T}$. Therefore:

Lemma 6. *Checking whether there is an acyclic annotation for* $\langle \mathcal{S}, ?, \{T_s = k_s\}_{s \in X}\rangle$ *is in NP.*

In the following two subsections we show that the problem $\langle \mathcal{S}, ?, \{T_r = k_r\}_{r \in X}\rangle$, that is, checking whether there exists an annotation α that satisfies the constraints $\{T_r = k_r\}_{r \in X}$ is NP-hard for arbitrary values of k_r, but that it can be decided in polynomial time for problems where $k_r = 1$ for all $r \in X$.

5.1 Arbitrary number of Resources

Lemma 7. *The problem* $\langle \mathcal{S}, ?, \{T_s = k_s\}_{s \in X}\rangle$ *is NP-hard.*

Proof. We use a reduction from 3-*CNF*. First, every formula φ can be transformed into an equisatisfiable formula φ' by rewriting each clause $C_j : (x_1 \vee x_2 \vee x_3)$, where x_1 stands for a variable X_1 or its negation $\overline{X_1}$, as follows

$$C'_j : (x_1 \vee Y_j \vee Z_j) \wedge (x_2 \vee \overline{Y_j}) \wedge (x_3 \vee \overline{Z_j}).$$

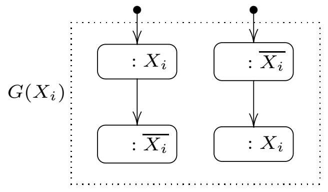
The auxiliary variables Y_j and Z_j separate the occurrences of the x_i from the original formula. We say that φ' is in *separated* normal form (SNF). This transformation increases the number of variables by at most $2|C|$, with $|C|$ the number of clauses, and increases the number of clauses by a factor of 3, so the generated formula is linear in the size of the original one.

Given a formula φ in SNF we build a distributed system $\mathcal{S} : \langle \mathcal{R}, \mathcal{G}\rangle$—linear in the size of the formula—and a problem specification $\langle \mathcal{S}, ?, \{T_s = k_s\}_{s \in \mathcal{R}}\rangle$, such that there is a one-to-one correspondence between acyclic annotations and satisfying valuations of φ.

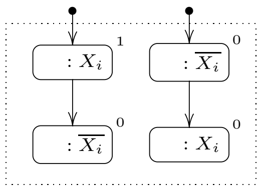
Resources: The resources are set to $\{T_r = 2\}$ for all sites. This enforces all feasible annotations to be $\alpha(n) \leq 1$.

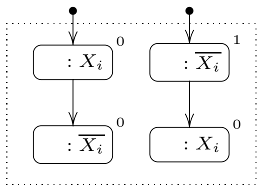
Sites: For each of the variables x_i (similarly for y_j and z_j) in the formula we introduce two sites, X_i and $\overline{X_i}$: one represents the positive occurrences of the variable, and the other the negative occurrences.

Methods and Edges: For each of the variables in the formula we introduce the following gadget:



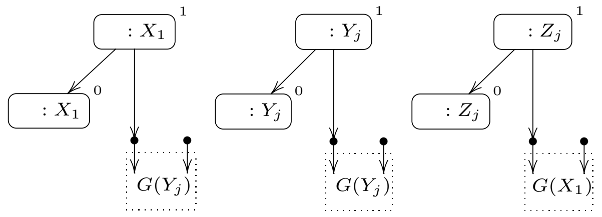
The only two possible acyclic annotations of this gadget that respect the constraints $T_{X_i} = 2$ and $T_{\overline{X_i}} = 2$ are:



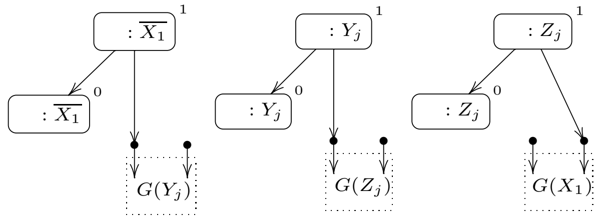


There is a one-to-one correspondence between acyclic annotations of these gadgets and valuations as follows: an annotation $\boxed{: X_i}^{1}$ denotes that variable X_i is false in the corresponding valuation, while $\boxed{: \overline{X_i}}^{1}$ X_i is true in the corresponding valuation.

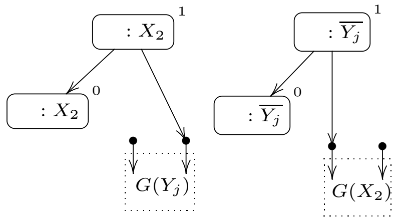
For each variable occurrence in a clause $C_j : (x_1 \vee Y_j \vee Z_j)$ we introduce the following gadget if x_1 occurs positive (i.e., $x_1 = X_1$). We also show the only possible acyclic annotation of this gadget:



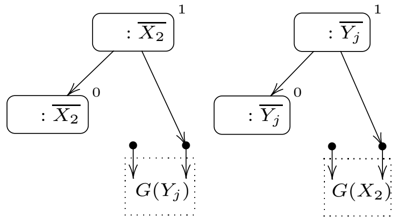
Similarly, if x_1 occurs negative (i.e., $x_1 = \overline{X_1}$):



Similarly, for the clauses $(x_2 \vee \overline{Y_j})$ if $x_2 = X_2$, the gadget is:



If $x_2 = \overline{X_2}$ the gadget is:



The gadget for the third clause $(x_3 \vee \overline{Z_j})$ is analogous. The separation variables y_j only occur once in positive form and once in negative form, so the only possible cycles in the graph, once a valuation has been picked, involve all the upper nodes in some clause gadget. This cycle exists if and only if all the clauses are unsatisfied. Therefore, if all clauses are satisfied the induced annotation has no cycles, and if there is an annotation with no cycles the corresponding valuation is satisfying. This reduction implies that checking whether a graph admits an acyclic annotation, with restrictions $\{T_r = k_r\}$ for $k_r \geq 2$ is NP-hard. □

5.2 Mutual Exclusion Resources

The problem $\langle \mathcal{S}, ?, \{T_s = k_s\}_{S \subseteq \mathcal{R}} \rangle$ becomes tractable if $k_s = 1$ for all $s \in S$, that is, if all constraints specify binary semaphores.

Lemma 8. *The problem $\langle \mathcal{S}, ?, \{T_s = 1\}_{S \subseteq \mathcal{R}} \rangle$ is in P.*

Proof. Consider the partially annotated graph G^α that only considers $\alpha(n) = 0$ for all methods residing in sites $s \in S$ that are marked as binary semaphores (0 is the only possible annotation for these nodes). If there are dependency cycles in G^α then there is no acyclic annotation since all annotated graphs extend G^α. Let $<$ be any reverse topological order that extends $\succ$. One such order exists since G^α is acyclic.

Now, the algorithm CalcMin($<$) generates an acyclic annotation β. When $n\!:\!s$ is visited in the algorithm, no children of n can reach any node in s, since that would imply that there is some ancestor of a node in s that has been computed. Therefore, $\beta(n)$ receives value 0 in line 10, and β extends α as desired. □

In case the technique described in Lemma 5 fails there is no protocol that can provide deadlock avoidance without communication. An alternative solution is to use deadlock prevention to break the cycles in the partially annotated call graph G^α: in some cases, a resource that will be needed in the future is reserved (and locked) before getting the actual resource needed.

Lemma 5 also provides an efficient conservative procedure to check the feasibility of the general problem, presented above. Some of the constrained resources can be restricted to be binary semaphores. Then Lemma 5 can be used to check feasibility: every solution with binary semaphores is a solution of the general problem.

6 Conclusions

The ability to compute minimal acyclic annotations for the call graphs is an important requisite for the successful application of our deadlock avoidance protocols for distributed systems. Minimal annotations minimize the least number of resources that must be provided and maximize concurrency.

We have demonstrated an algorithm that can compute all minimal acyclic annotations. We showed that imposing constraints on the number of available resources makes the problem of computing minimal acyclic annotations NP-hard, except when the constraints only involve binary semaphores, in which case the problem is in P.

We reemphasized the importance of using acyclic annotations. We already knew that in a system with cyclic annotations increasing the number of resources may turn a deadlock free system into a system with reachable deadlocks. Here we showed that checking whether a particular number of resources does indeed admit reachable deadlocks is NP-complete.

References

[1] W. Stallings, *Operating Systems: Internals and Design Principles*, Third ed. Upper Saddle River, NJ: Prentice Hall, Inc., 1998.

[2] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, Sixth ed. New York, NY: John Wiley & Sons, Inc., 2003.

[3] C. Papadimitriou, *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[4] A. D. Birrell, "An introduction to programming with threads," Digital Equipment Corporation Systems Research Center, Research Report 35, 1989.

[5] E. W. Dijkstra, "Cooperating sequential processes," Technological University, Eindhoven, the Netherlands, Tech. Rep. EWD-123, 1965.

[6] A. N. Habermann, "Prevention of system deadlocks," *Communications of the ACM*, vol. 12, pp. 373–377, 1969.

[7] J. W. Havender, "Avoiding deadlock in multitasking systems," *IBM Systems Journal*, vol. 2, pp. 74–84, 1968.

[8] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems.* New York, NY: McGraw-Hill, Inc., 1994.

[9] C. Sánchez, H. B. Sipma, V. Subramonian, C. Gill, and Z. Manna, "Thread allocation protocols for distributed real-time and embedded systems," in *25th IFIP WG 2.6 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'05)*, ser. LNCS, F. Wang, Ed., vol. 3731. Taipei, Taiwan: Springer-Verlag, October 2005, pp. 159–173.

[10] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* New York: Wiley & Sons, 2000.

[11] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers," *Communications of the ACM Special Issue on CORBA*, vol. 41, no. 10, pp. 54–60, Oct. 1998.

[12] V. Subramonian, G. Xing, C. D. Gill, C. Lu, and R. Cytron, "Middleware specialization for memory-constrained networked embedded systems," in *Proc. of 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04).* IEEE Computer Society Press, May 2004.

[13] C. Sánchez, H. B. Sipma, Z. Manna, V. Subramonian, and C. Gill, "On efficient distributed deadlock avoidance for distributed real-time and embedded systems," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06).* Rhodas, Greece: IEEE Computer Society Press, 2006.

[14] L. A. Belady, R. A. Nelson, and G. S. Shedler, "An anomaly in space-time characteristics of certain programs running in a paging machine," *Communications of the ACM*, vol. 12, no. 6, pp. 349–353, 1970.

[15] C. Sánchez, H. B. Sipma, Z. Manna, and C. Gill, "Efficient distributed deadlock avoidance with liveness guarantees," in *To appear in the Proceedings of the 6th Annual ACM Conference on Embedded Software (EMSOFT'06).* Seoul, South Korea: ACM Press, 2006.

[16] C. Sánchez and H. B. Sipma, "Reachable state spaces of distributed deadlock avoidance algorithms," Stanford Computer Science, STeP Group, Tech. Rep. 8-1, June 2006, available from http://theory.stanford.edu/~cesar.