On Efficient Distributed Deadlock Avoidance for Real-Time and Embedded Systems

César Sánchez[*] Henny B. Sipma[*] Zohar Manna[*]
Venkita Subramonian[†] Christopher Gill[†]

[*]Stanford University
Computer Science Dept.
Stanford, CA 94305 USA
{cesar,sipma,zm}@CS.Stanford.EDU

[†]Washington University
Dept. of Computer Science and Engineering
St. Louis, MO 63130 USA
{venkita,cdgill}@CSE.wustl.EDU

Abstract

Thread allocation is an important problem in distributed real-time and embedded (DRE) systems. A thread allocation policy that is too liberal may cause deadlock, while a policy that is too conservative limits potential parallelism, thus wasting resources. However, achieving (globally) optimal thread utilization, while avoiding deadlock, has been proven impractical in distributed systems: it requires too much communication between components.

In previous work we showed that efficient local thread allocation protocols are possible if the protocols are parameterized by global static data, in particular by an annotation of the global call graph of all tasks to be performed by the system. We proved that absence of cyclic dependencies in this annotation guarantees absence of deadlock.

In this paper we present an algorithm to compute optimal annotations, that is annotations that maximize parallelism while satisfying the condition of acyclicity. Moreover, we show that the condition of acyclicity is in fact tight and exhibits a rather surprising anomaly: if a cyclic dependency is present in the annotation of the call graph and a certain minimum number of threads is provided, deadlock is reachable. Thus, in the presence of cyclic dependencies, increasing the number of threads may introduce the possibility of deadlock in an originally deadlock free system.

1. Introduction

In this paper we study a problem common to several forms of resource allocation in distributed real-time and embedded (DRE) systems. We focus our study on distributed middleware architectures, and in particular on thread allocation in the presence of "nested upcalls". We use "thread" to mean an "execution context", a resource necessary to run a computation. We assume that a dispatching mechanism (i.e., a *reactor* [1]) in each distributed computational node manages access to a local thread pool of a constant size that is fixed at design time. A nested upcall is produced when a method running in reactor A invokes a remote method in reactor B, which in turn invokes, possibly through further calls, a method in A. Nested upcalls can be produced in a variety of middleware implementations [1, 2, 3]. One common way to deal with nested upcalls is *Wait-OnConnection*: every time a method is invoked, the calling task holds on to its thread until the call returns. Consequently, any subsequent invocation to the same reactor—including nested upcalls—requires a new thread to execute. A common goal in the design of DRE systems is to calculate a bound on resources. Provided with a fixed number of threads, this strategy can lead to deadlocks. How to avoid these deadlocks is the problem studied in this paper.

An execution of a system consists of a concurrent flow of "tasks" (e.g., method upcalls) competing for threads, which can lead to a classic deadlock situation. Deadlock occurs when a set of tasks is in a "circular wait" state, where each task in the set is waiting for a thread currently held by another task in the set while locking a thread that is, in turn, needed by one of the other tasks. The phenomenon of deadlock has

[*]This research was supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, by ARO grant DAAD19-01-1-0723, and by NAVY/ONR contract N00014-03-1-0939.

[†]Supported in part by NSF CAREER grant CCF-0448562.

1-4244-0054-6/06/$20.00 ©2006 IEEE

been studied extensively in the context of computer operating systems [4]. The two classical solutions are deadlock *prevention* and deadlock *avoidance*.

Deadlock prevention is not possible for *WaitOnConnection* because (1) threads can only be allocated to one task a time, (2) a thread is held until the call chain is finished, which may require subsequent threads in the same and other reactors, and (3) a thread being held by one task cannot be released to another task.

Deadlock avoidance methods keep the system in a safe set of states where the circular chain of resource contention that produces the deadlock does not occur. Our work follows this approach. A classic deadlock avoidance algorithm for non-distributed systems is Dijkstra's Banker's algorithm [5], which initiated much follow-up research [6, 7, 8, 9] and is still the basis for most current algorithms. Its key characteristic is the use of a combination of static knowledge (the maximum amount of resources needed by each task) and dynamic knowledge (the number of resources currently available) to make its dynamic decisions about resource allocation. This approach has been further refined in deadlock avoidance algorithms for Flexible Manufacturing Systems (FMSs) [10, 11, 12, 13], which usually incorporate more static knowledge about the processes to maximize concurrency without sacrificing deadlock freedom.

For distributed systems, on the other hand, it was observed in [14, 15] that a *general* solution to distributed deadlock avoidance is impractical, since it requires global atomic actions, or distributed synchronization. In our previous work [16], however, we show that efficient deadlock avoidance algorithms exist in the *particular* case of systems in which the distributed call graphs are nonrecursive and known a priori. For DRE systems this is usually the case.

An efficient distributed deadlock avoidance algorithm should access only local data at run-time. This can be complemented, however, by static global data. The protocols we proposed in [16] are parameterized by annotations that are computed statically, based on the call graphs. The efficiency (dispatching throughput) and correctness (deadlock avoidance) depend on these annotations. In [16] we established an *annotation condition* that guarantees deadlock avoidance. The first contribution of this paper is an algorithm to compute annotations that satisfy the annotation condition while maximizing thread utilization and thus throughput. This algorithm appears in Section 4.

The second contribution of this paper, which appears in Sections 5 and 6, is a proof that the annotation condition is tight: violating it compromises deadlock freedom. Moreover we show that if the annotation condition is violated, the following *anomaly*[1] can occur: a system that is deadlock free with a certain number of threads can reach deadlock with a larger number of threads. This is clearly undesirable from an engineering perspective, since it is common practice to "overprovision", that is to assign extra resources to seek higher confidence that safety and performance requirements will be met. Therefore, to avoid this situation, the annotation condition must be satisfied.

Sections 2 and 3 summarize the computational model and present the Annotation Theorem. Finally, Section 7 presents our conclusions and describes possible lines of further research.

2. Model of Computation

We model a DRE system $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ as a set of reactors $\mathcal{R} : \{r_1, \ldots, r_k\}$ and a set of call graphs $\mathcal{G} : \{G_1, \ldots, G_m\}$. The reactors model the distributed components where the local thread allocation is managed and the computations are performed. The call graphs model the flows of possible computations as finite trees $\langle N = (M \times \mathcal{R}), E \rangle$, where each node (f, r) in N consists of a method name f (from a set M of possible methods), and the reactor r that executes f. We use the notation $n : \langle f \!:\! r \rangle$ to represent that node n consists of method f that runs in reactor r. An edge in E between $\langle f : r \rangle$ and $\langle g : s \rangle$ represents that f, in the course of its execution in r may invoke method g in reactor s.

Each reactor r maintains a set of local variables V_r and has a number of preallocated threads (represented by the constant $T_r \geq 1$). V_r also contains a local variable t_r whose value represents the number of available threads in r. Initially, in every reactor r, $t_r = T_r$.

A *protocol* implements the distributed deadlock avoidance algorithm and consists of two pieces of code: one that runs when an incoming invocation requests a new thread, and another that executes when a method has terminated and the thread can be released. The protocol runs in the reactor where the corresponding method resides, and can be different for each node in each call graph. In particular, two different methods that run in the same reactor can run different protocols.

Using these protocols, the invocation of a method corresponds to the execution of the abstract program shown in Fig. 1. The entry section, labeled ℓ_0, checks a condition on the local variables of the reactor to ensure the availability of the thread and the safety of its assignment. If the condition is satisfied, then some local

[1] This anomaly resembles the Belady anomaly [17] that assigning more resources can degrade a system's performance.

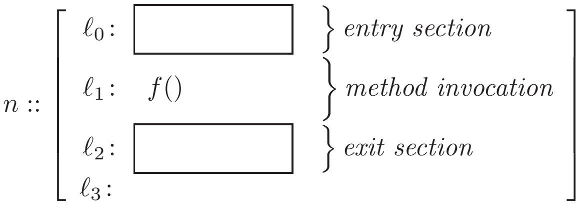


Figure 1. Protocol schema for node $n : \langle f : r \rangle$**.**

variables are modified accordingly, and the task enters the method invocation section, labeled ℓ_1. Upon termination (which might require the return of some remote calls), the exit section, labeled ℓ_2 is run, which updates local variables to record the release of the thread. The entry and exit sections are executed atomically.

Multiple instances of these protocols can be running concurrently in each reactor. Each instance is called a *task*. Formally, the state of a task is modeled as a labeled call graph:

Definition 1 (Labeled Call Graph)**.** *A labeled call graph* (G, γ) *is an instance of a call graph* $G \in \mathcal{G}$ *and a labeling function* $\gamma : N_G \mapsto \{\perp, \ell_0, \ell_1, \ell_2, \ell_3\}$ *that maps each node in the call graph to a protocol location, or to* $\perp$ *for method calls that have not been performed yet.*

For example, when a new task is spawned, the corresponding call graph G is labeled as follows: $\gamma(root(G)) = \ell_0$, and $\gamma(n) = \perp$ for the rest of the nodes. The subtree of an annotated call graph models the state of a sub-task. We use "task" to refer to both sub-tasks and proper tasks. If the root of a task is labeled ℓ we say that the task is in ℓ. A task is said to be *active* if it is in ℓ_1 or in ℓ_2, and *waiting* if it is in ℓ_0.

The *state* $\sigma : \langle I, s_{\mathcal{R}} \rangle$ of a system $\mathcal{S}$ consists of a set I of existing (proper) tasks and valuations $s_{\mathcal{R}}$ for the local variables V_r of all reactors. The initial state of a system Θ entails that $I = \varnothing$ (since there is no task running initially in the system) and the initial conditions Θ_r for all reactors r. In particular Θ_r entails that $t_r = T_r$, since all threads are initially available.

A *run* of a system is an infinite sequence of states $\sigma_0, \sigma_1, \ldots$ such that (1) σ_0 is an initial state ($\sigma_0 \vDash \Theta$) and (2) every state is obtained from a previous state by a system's transition, i.e., for every i, σ_{i+1} results from σ_i by taking one of the following transitions τ:

1. **Creation**: A new task is added to I, with all nodes labeled $\perp$.
2. **Method invocation**: a sub-task labeled $\perp$, and whose parent task is labeled ℓ_1, changes to ℓ_0.
3. **Method entry**: a waiting task whose enabling condition, according to the protocol for its root node, is satisfied changes from ℓ_0 to ℓ_1 and updates the variables according to the protocol.
4. **Method execution**: a task in ℓ_1 all of whose descendants are labeled $\perp$ or ℓ_3 changes its label to ℓ_2.
5. **Method exit**: a task in ℓ_2 executes the action corresponding to the exit section of its protocol and changes the label to ℓ_3.
6. **Deletion**: a proper task in ℓ_3 is removed from I.
7. **Silent**: the state of the system is preserved.

All transitions except **Creation** and **Silent** are called *progressing* transitions, since they correspond to the progress of some existing task. Note that each task only has a finite number of reachable labelings before it is deleted. Moreover, no labeling is visited twice.

We assume that the protocols satisfy the following two conditions:

1. The effect of the entry and exit sections on the local variables of V_r cancel each other.
2. If the entry condition of some node n is disabled, after the execution of the entry by some other task, it is still disabled.

These two properties state that all the threads granted to a task are returned when it finishes, and that the assignment of threads to a different task cannot help a waiting task to gain access to its desired resources. In particular, spawning new tasks cannot help deadlocked tasks. We are now ready to define deadlock formally :

Definition 2 (Deadlock)**.** *A state* σ *is called a deadlock if some task is waiting, but only non-progressing transitions are enabled.*

If a deadlock is reached, the tasks involved cannot progress. Intuitively, each of the tasks has locked some threads that are necessary for other tasks to complete, but none of them has enough resources to terminate.

Deadlock-avoidance Protocols In [16] we introduced protocols that avoid deadlock states and proved their correctness. These protocols are based on *annotations* of call graphs, which are extra static information attached to the call graph nodes and inspected in the protocols. Formally, let $N = \bigcup_i N_i$ be the (disjoint) union of the nodes in the call graphs; an annotation is a map $\alpha : N \mapsto \mathbb{N}$. Examples of possible annotations are the height of the node in the call graph, or the local-height (maximum number of nodes that reside in the same reactor in any descending path).

The first protocol, Basic-P, is shown in Fig. 2 (see also [16]). In the entry section, access is granted only if the number of resources indicated by the annotation is less than or equal to the number of threads available. Note that when access is granted, not all resources—as indicated by the annotation—are immediately reserved, since t_r is only decremented by one.

Basic-P can be optimized by exploiting the observation that tasks that need only one thread can always terminate once the thread is granted, independently of other tasks. The protocol Efficient-P, shown in Fig. 3 (see also [16]) uses an extra local variable p_r to keep track of the *potentially* available threads, while still using t_r for the number of threads actually available.

3. The Annotation Theorem

Deadlock is caused by cyclic dependencies. In general, the interference between different concurrent tasks has to be considered. We introduce the notion of a global call graph to capture all possible interferences:

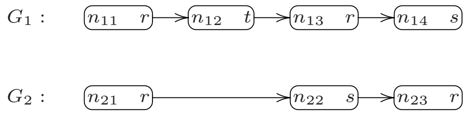
Definition 3 (Global Call Graph)**.** *Given a system* $\mathcal{S} : \langle \mathcal{R}, \{G_1, \ldots, G_m\}\rangle$ *and an annotation function* α*, the global call graph* $G_{\mathcal{S},\alpha} : \langle N, \rightarrow, \dashrightarrow\rangle$ *consists of:*

- N: $\bigcup_i N_i$*, the union of all call graph nodes;*
- $\rightarrow$: $\bigcup_i \rightarrow_i^+$*, the union of the descendant relations of all call graphs* G_i*, where* $\rightarrow_i^+$ *is the transitive closure of* $\rightarrow_i$*;*
- $\dashrightarrow$ *is defined as:*

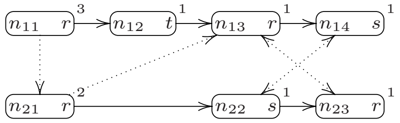
  $$\{(v, w) \mid \alpha(v) \geq \alpha(w) \text{ and } reactor(v){=}reactor(w)\}$$

  where v *and* w *may belong to different call graphs.*

Example 1. *Consider the following call graphs (function names are omitted for clarity):*



Consider the annotation $\alpha(n_{11}) = 3$*,* $\alpha(n_{12}) = \alpha(n_{13}) = \alpha(n_{14}) = 1$*, and* $\alpha(n_{21}) = 2$*,* $\alpha(n_{22}) = \alpha(n_{23}) = 1$*. The following figure shows the global call graph where the solid lines indicate edges in* $\rightarrow$ *and the dotted lines indicate edges in* $\dashrightarrow$*. Transitive edges are omitted for clarity.*



$$n :: \left[\begin{array}{ll} \ell_0: & \left[\begin{array}{c}\textbf{when } \alpha(n) \leq t_r \textbf{ do}\\ t_r\text{--}\end{array}\right] \\ \ell_1: & f() \\ \ell_2: & t_r\text{++} \\ \ell_3: & \end{array}\right]$$

Figure 2. Protocol Basic-P **for node** $n : \langle f : r\rangle$**.**

Definition 4 (Dependency Relation)**.** *Given a global call graph* $G_{\mathcal{S},\alpha} : \langle N, \rightarrow, \dashrightarrow\rangle$*,* $v \in N$ *is dependent on* $w \in N$*, written* $v \succ w$*, if there exists a path from* v *to* w *consisting of edges in* $\rightarrow \cup \dashrightarrow$ *with at least one edge in* $\rightarrow$*.*

For instance, in the global call graph of Example 1 above, $n_{11} \succ n_{22}$, but $n_{11} \not\succ n_{21}$. A global call graph has a *cyclic dependency* if for some node v, $v \succ v$. The global call graph of Example 1 contains the following cyclic dependency: $n_{13} \succ n_{13}$, since $n_{13} \rightarrow n_{14} \dashrightarrow n_{22} \rightarrow n_{23} \dashrightarrow n_{13}$. The following theorem justifies that the protocols introduced above are indeed distributed deadlock avoidance algorithms:

Theorem 5 (Annotation Theorem, from [16])**.** *Given a system* $\mathcal{S}$ *and annotation* α*, if the global call graph* $G_{\mathcal{S},\alpha}$ *does not contain any cyclic dependencies, then both* Basic-P *and* Efficient-P *used with* α *guarantee absence of deadlock.*

A system $\mathcal{S}$ with annotation α satisfies the *annotation condition* if there is no cyclic dependency in its global call graph $G_{\mathcal{S},\alpha}$.

4. Minimal Annotations

In this section we show how to compute globally acyclic annotations efficiently.

An annotation α is called *minimal* if it does not have dependency cycles and reducing the value of the annotation for one node produces a dependency cycle. It is easy to see that in a minimal annotation decreasing the value of any number of nodes while preserving the rest also generates a dependency cycle. Moreover, the annotation of any leaf in a minimal annotation is 1.

To compute a minimal annotation, we iterate through the nodes in the global call graph in a reverse topological order (with respect to the local descendant relation $\rightarrow$), i.e., we visit a node m before a node n if $n \rightarrow m$. At the iteration for node n we compute the minimum value of $\alpha(n)$ such that no cycle exists among the nodes already visited. When n is visited, the following sets are computed:

$$n :: \left[\begin{array}{ll} \ell_{0\_1}: & \left[\begin{array}{c} \textbf{when } 1 \le t_r \textbf{ do} \\ t_r\texttt{--} \end{array}\right] \\ \ell_{1\_1}: & f() \\ \ell_{2\_1}: & t_r\texttt{++} \\ \ell_{3\_1}: & \end{array}\right] \qquad n :: \left[\begin{array}{ll} \ell_{0\_2}: & \left[\begin{array}{c} \textbf{when } \alpha(n) \le p_r \ \wedge \ 1 \le t_r \textbf{ do} \\ \langle p_r\texttt{--},\ t_r\texttt{--}\rangle \end{array}\right] \\ \ell_{1\_2}: & f() \\ \ell_{2\_2}: & \langle t_r\texttt{++},\ p_r\texttt{++}\rangle \\ \ell_{3\_2}: & \end{array}\right]$$

If $\alpha(n) = 1$ If $\alpha(n) > 1$

Figure 3. The protocol Efficient-P **for node** $n : \langle f\!:\!r\rangle$**.**

```
 1: {Order $N$ in reverse topological order}
 2: {Let $\mathrm{React}_n = \{m \mid \mathrm{reactor}(m){=}\mathrm{reactor}(n)\}$}
 3: for $n = n_1$ to $n_{|N|}$ do
 4:     if $n$ is a leaf then
 5:         $\mathrm{Below}_n \leftarrow \varnothing$
 6:         $\mathrm{Cand}_n \leftarrow \varnothing$
 7:         $\alpha(n) \leftarrow 1$
 8:     else {$n$ has $D = \mathrm{Desc}_n$ direct descendants}
 9:         $\mathrm{Below}_n \leftarrow \varnothing$
10:         new $= D$
11:         repeat
12:             $\mathrm{Below}_n \leftarrow \mathrm{Below}_n \cup \mathrm{new}$
13:             new $\leftarrow \bigcup_{i \in \mathrm{Below}_n} (\mathrm{Below}_i \cup \mathrm{Reach}_i)$
14:         until new $\subseteq \mathrm{Below}_n$
15:         $\mathrm{Cand}_n \leftarrow \mathrm{Below}_n \cap \mathrm{React}_n$
16:         $\alpha(n) \leftarrow 1 + \max\ \{\alpha(m) | m \in \mathrm{Cand}_n\}$
17:     end if
18:     $\mathrm{Reach}_n \leftarrow \varnothing$
19:     for all $m$ visited, with $m \in \mathrm{React}_n$ do
20:         if $\alpha(m) \le \alpha(n)$ then
21:             $\mathrm{Reach}_n \leftarrow \mathrm{Reach}_n \cup \{m\}$
22:         end if
23:         if $\alpha(n) \le \alpha(m)$ then
24:             $\mathrm{Reach}_m \leftarrow \mathrm{Reach}_m \cup \{n\}$
25:         end if
26:     end for
27: end for
```

Figure 4. Algorithm for minimal annotations

1. The set $\mathrm{Desc}_n = \{m \mid n \rightarrow m\}$ of all the direct descendants of n.
2. The set Below_n containing all nodes that can be reached from descendants of n by any sequence of $\rightarrow$ or $\dashrightarrow$:

$$\mathrm{Below}_n = \{m \mid \exists d \in \mathrm{Desc}_n,\ d(\rightarrow \cup \dashrightarrow)^* m\}$$

3. Let React_n be the set of nodes residing in the same reactor as n. We define Cand_n as the set of all possible nodes (candidates) that can precede n in a cyclic dependency:

$$\mathrm{Cand}_n = \{m \mid m \in \mathrm{Below}_n,\ \text{and } m \in \mathrm{React}_n\}.$$

To see that every candidate to precede n in a cyclic dependency belongs to Cand_n, consider a cycle C that contains n; clearly the node m preceding n in C has to reside in the same reactor, since according to the reverse topological order, no ancestor of n has been visited. On the other hand, the successor s of n in C is either: (1) a descendant of n (which is covered in Below_n) or (2) resides in the same reactor as n. Case (2) is impossible since otherwise $m \dashrightarrow n \dashrightarrow s$ and, by removing n we can form a cycle for $s \succ s$ without using n. This contradicts the inductive hypothesis that prior to visiting n the annotation has no cycles.

Finally, to break condition (1) we prevent $m \dashrightarrow n$ for all candidates m from Cand_n by assigning the following annotation for n:

$$\alpha(n) = 1 + \max\{\alpha(m) \mid m \in \mathrm{Cand}_n\}.$$

Note that with this annotation n does not appear in any cycle with nodes visited previously. Moreover, by picking a value smaller than $\alpha(n)$ a cycle can be created, which justifies the minimality of the annotation. Different minimal annotations can be computed by picking different iteration orders.

Even a naïve implementation of this algorithm that computes the Below_n, Desc_n and Cand_n sets from scratch at every iteration exhibits a low polynomial complexity: Fig 4 shows a direct implementation. Lines 8-13 compute the set of nodes that are reachable from the direct descendants of n using any number of $\rightarrow$ or $\dashrightarrow$, which takes less than $|N|^2$ steps. Lines 18-25 maintain in Reach_n the set of nodes reachable with $\dashrightarrow$, which takes less than $|N|$ steps. Therefore, the program finishes in $O(|N|^3)$. Note that this computation is performed at design time, and the annotations calculated are used but not changed at run-time.

This off-line approach could in fact be applied online at a longer time scale than that of the functions themselves, for example as part of the DRE system admission control for new functionality at run-time.

5. Cycles and Unavoidable Deadlocks

This section introduces preliminary definitions that will be used in Section 6 to prove that deadlocks are always reachable if the Anotation Condition is not satisfied.

Dependency Cycles A cyclic dependency can occur by a sequence of nodes $v_1, \dots, v_k$, with $v_1 = v_k$ such that:

1. for all i, $v_i \to v_{i+1}$ or $v_i \dashrightarrow v_{i+1}$, and
2. for some j, $v_j \to v_{j+1}$.

Without loss of generality, since both relations $\to$ and $\dashrightarrow$ are transitive, if one such sequence exists, then there is another sequence such that edges from $\to$ and $\dashrightarrow$ alternate:

Definition 6 (Dependency Cycle)**.** *A dependency cycle consists of two sequences of nodes $\langle [A_1, \dots, A_k], [B_1, \dots, B_k] \rangle$, of the same length, such that for all i there is an $\to$ edge from A_i to B_i and a $\dashrightarrow$ edge from B_i to A_{i+1} (here + stands for addition modulo $k+1$):*

$$\begin{tikzcd}
A_1 \arrow[d] & A_2 \arrow[d] & A_3 \arrow[d] & {} & A_k \arrow[d] \\
B_1 \arrow[ur, dotted] & B_2 \arrow[ur, dotted] & B_3 \arrow[ur, dotted] & \cdots & B_k \arrow[llllu, dotted]
\end{tikzcd}$$

The nodes A_i are called A-nodes or "above" nodes, and nodes from B are called B_i-nodes or "below" nodes. We say that a dependency cycle is *simple* if all A-nodes reside in a different reactor.

Lemma 7. *If a global call graph $G_{\mathcal{S},\alpha}$ has a dependency cycle, then it also has a simple dependency cycle.*

Proof. By contradiction, assume that there is a dependency cycle but no simple dependency cycle. Let $C = \langle [A_1, \dots, A_k], [B_1, \dots, B_k] \rangle$ be one dependency cycle with the minimum number of pairs of A-nodes that reside in the same reactor. In C there are A_i and A_j that reside in the same reactor (w.l.o.g. we assume $j > i$). Then, either $\alpha(A_i) \geq \alpha(A_j)$ or $\alpha(A_j) \geq \alpha(A_i)$. In the first case:

$$\begin{tikzcd}
A_{i-1} \arrow[d] & A_i \arrow[d] \arrow[rr, dotted, bend left] & & A_j \arrow[d] \\
B_{i-1} \arrow[ur, dotted] & B_i & & B_j
\end{tikzcd}$$

by transitivity of $\dashrightarrow$, $B_{i-1} \dashrightarrow A_j$ and therefore the nodes

$$\langle [A_1, \dots, A_{i-1}, A_j, \dots, A_k], \\ [B_1, \dots, B_{i-1}, B_j \dots, B_k] \rangle$$

form a dependency cycle of strictly fewer pairs of A-nodes running in the same reactor. Similarly, if $\alpha(A_j) \geq \alpha(A_i)$, as shown in:

$$\begin{tikzcd}
A_i \arrow[d] & & A_{j-1} \arrow[d] & A_j \arrow[d] \arrow[lll, dotted, bend right] \\
B_i & & B_{j-1} \arrow[ur, dotted] & B_j
\end{tikzcd}$$

By transitivity $B_{j-1} \dashrightarrow A_i$, and then the sub-graph $\langle [A_i, \dots, A_{j-1}], [B_i, \dots, B_{j-1}] \rangle$ forms a dependency cycle with fewer coincidences. In both cases the minimality of the dependency cycle C is contradicted. Therefore there is a simple dependency cycle, as desired. □

The following definition models the sequence of calls that a task must perform to enter a node and execute the corresponding method:

Definition 8 (Path)**.** *A path in a call graph G_j is a sequence of nodes, starting from the root, that follows the descendant relation $\to_j$. The path* leading *to a node n is the ordered sequence of its ancestors. The nodes $n_1, \dots, n_{k-1}$ are called* internal *nodes of the path π : $(n_1, \dots, n_{k-1}, n_k)$.*

Unavoidable deadlocks We introduce the notion of "unavoidable deadlocks" to aid in reasoning about system states that will inevitably reach a deadlock. An unavoidable deadlock state will reach a deadlock if the tasks involved are scheduled to execute, so either the tasks starve or the system will reach a deadlock, but they cannot progress to termination. Unavoidable deadlocks are easier to produce in proofs than deadlocks.

Definition 9 (Unavoidable Deadlock)**.** *A state σ with tasks $I = \{P_i\}$ is an unavoidable deadlock state if no P_i terminates in any state reachable from σ.*

An alternative characterization is given by:

Lemma 10. *If in state σ no task can individually proceed to completion when continuously scheduled, then σ is an unavoidable deadlock.*

Proof. (Sketch) We show that if σ is not an unavoidable deadlock state, then there is a task that can proceed to termination when continuously scheduled. Consider the shortest run extending σ for which an existing task terminates. Clearly, there is no creation of new tasks since we could produce a strictly shorter run by removing all the corresponding transitions. If all transitions in the extension are related to the terminating task P, we are done. If not, pick one of the transitions τ that is not related to P; since τ does not increase resources, all transitions that are subsequently enabled in the run were enabled had τ not been taken. Therefore, we can produce a shorter run by removing τ. □

6. Producing Deadlocks

In this section we show that in every scenario whose global call graph contains a dependency cycle, given enough resources, a deadlock can be reached.

Example 2 below shows a scenario with an annotation and initial resources that does not lead to a deadlock in spite of the presence of a cyclic dependency. We then increase the reactor's resources by just enough threads to produce a deadlock.

Given a system consisting of a global call graph with a dependency cycle we prove the existence of resources such that some run leads to deadlock. We calculate these resources by generating a system's run such that when all the A-nodes in a simple dependency cycle are entered by tasks, no more tasks can visit any A-node, and consequently no B-node can be visited either. In effect, the tasks cannot proceed beyond the A-nodes thus reaching an unavoidable deadlock state.

We construct the run as follows. Let $\langle[A_1,\ldots,A_m],[B_1,\ldots,B_m]\rangle$ be a simple dependency cycle in a global call graph G, for the (distinct) reactors $r_1,\ldots,r_m \subseteq \mathcal{R}$. Let $\pi_1,\ldots,\pi_m$ be the paths leading to the A-nodes $A_1,\ldots,A_m$. We build an execution by spawning k_i tasks for each path π_i and scheduling the k_i tasks to gain access to the nodes in π_i simultaneously.

During the construction of this execution we generate a set of constraints (on the possible values of k_i and the total number of threads $\{T_r\}$) in order for all the sets of tasks k_i to be able to reach their target node A_i and exhaust the threads after they gain access to A_i's method section. A global constraint consisting of the conjunction of all these intermediate constraints captures for which values the run exists. Finally, we prove that this global constraint is satisfiable. Each solution corresponds to an actual execution of the system that reaches an unavoidable deadlock.

The first constraints capture the properties that the sets of resources are not empty and that at least one task follows each path:

$$\bigwedge_r T_r \geq 1 \qquad\qquad \bigwedge_{\pi_i} k_i \geq 1 \qquad (1)$$

A (macro) step in the execution consists of all the k_i tasks that follow some path π_i entering the method section of a node n in π_i. In terms of the computational model described in Sec. 2 above, this corresponds to k_i consecutive executions of the **Method entry** transition. We use (n,k_i) to denote that all the k_i tasks gain access to the method section of node n, and say that the k_i tasks "visit" node n. We use H to represent the set of all visits during an execution:

$$H \overset{\text{def}}{=} \{(n,k_i) \mid n \text{ belongs to path } \pi_i\}.$$

Observe that, in principle, the same node n could belong to different paths, if they share a common prefix, and therefore there can be more than one visit to the same node n (for different k_i's).

The steps of the paths can be interleaved in many ways, each of which leads to different runs and is captured by different constraints. We consider any total order $<$ on H that respects the topological order of each path, that is, if n appears before m in path π_i then $(n,k_i) < (m,k_i)$. Also, we say that a total order is *admissible* if it also satisfies that every A -node is the last node visited residing in its reactor (i.e., if (n,k_j) resides in r_i then $(n,k_j) < (A_i,k_i)$.) Finally, we define $(H_r,<_r)$ to be the projection of $(H,<)$ for nodes that reside in reactor r:

$$H_r \overset{\text{def}}{=} \{(n,k_i) \mid (n,k_i) \in H \text{ and reactor}(n){=}\ r\}.$$

Note that the order $<$ is admissible precisely when every A_i is maximum in $<_{r_i}$.

The set of constraints that implies the unavoidable deadlock is reached is:

- **Threads are exhausted**: For all reactors r_i all threads are exhausted after the k_i tasks visit A_i

  $$\psi_i \;:\; T_{r_i} - \sum_{(n,k_j)\in H_{r_i}} k_j = \alpha(A_i) - 1. \qquad \text{(C1)}$$

  Note that $\sum_{(n,k_j)\in H_{r_i}} k_j$ corresponds to the total number of threads reserved in this execution in reactor r. This constraint forces the remaining threads in r to be $\alpha(A_i)-1$, and therefore insufficient for any subsequent visit to B_{i-1}.

- **The run is feasible**: For all intermediate nodes $(n,k_i) \in H$, with $n \neq A_i$ the protocol allows thread allocation for all k_i tasks. Assuming that node n resides in r, this is expressed by

  $$\varphi_{(n,k_i)} \;:\; T_r - \sum_{(m,k_j)\leq_r (n,k_i)} k_j \geq \alpha(n) - 1. \qquad \text{(C2)}$$

To show that an unavoidable deadlock is reachable it is sufficient to show that the following global constraint – together with (1) – is satisfiable:

$$\Phi \;:\; (\bigwedge_{\nu\in H} \varphi_\nu) \wedge (\bigwedge \psi_i)$$

A solution to Φ provides the initial resources and the number of tasks following each path that produce the unavoidable deadlock. To see that Φ is satisfiable we first simplify (C1) as:

$$T_{r_i} = \sum_{(n,k_j)\in H_r} k_j + \alpha(A_i) - 1. \tag{C1'}$$

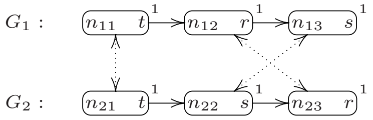
This equation gives a means to compute the value of T_{r_i} once all the k_i values (and the order $<$) are determined. Since $\alpha(A_i) \geq 1$ and $k_i \geq 1$, then $T_{r_i} \geq 1$ and this equation is consistent with (1).

Now, using (C1') we simplify the constraint (C2) corresponding to $\varphi_{(n,k_i)}$ to:

$$\sum_{(m,k_j)>_r(n,k_i)} k_j \geq \alpha(n) - \alpha(A_i). \tag{C2'}$$

The following example illustrates the use of this technique to construct a run.

Example 2. *Consider a scenario with the following call graphs G_1 and G_2 (function names are again omitted), where we also display the annotations and the corresponding global call graph:*



This global call graph has a simple dependency cycle $\langle[n_{12}, n_{22}], [n_{13}, n_{23}]\rangle$. If the resources allocated are $T_r = 1$, $T_s = 1$ and $T_t = 1$ no deadlock is reachable. To see this, it is enough to observe that only one task can be granted access to either node n_{11} or n_{21}, so the facts that $T_t = 1$ and that these nodes have annotation 1 serialize the access to the rest of the nodes in the call graphs. This serialization actually breaks the cyclic contention expressed by the annotation condition.

However, we show that allocation of more reactor threads (by increasing T_t) could lead to a deadlock. We illustrate this now using the technique outlined above. The two paths leading to A-nodes are $\pi_1 = (n_{11}, n_{12})$ and $\pi_2 = (n_{21}, n_{22})$. Let k_1 denote the number of tasks following π_1 and k_2 following π_2. The set of visits is:

$$H = \{(n_{11}, k_1), (n_{12}, k_1), (n_{21}, k_2), (n_{22}, k_2)\}.$$

We pick the following admissible total order $<$, that respects the topological order of π_1 and π_2 and for which the A-nodes are the last visits of their reactors:

$$(n_{11}, k_1) < (n_{21}, k_2) < (n_{12}, k_1) < (n_{22}, k_2).$$

Consequently, the set of constraints is:

$$\begin{array}{lcl} T_t - k_1 & \geq & \alpha(n_{11}) - 1 \\ T_t - k_1 - k_2 & \geq & \alpha(n_{21}) - 1 \\ T_r - k_1 & = & \alpha(n_{12}) - 1 \\ T_s - k_2 & = & \alpha(n_{22}) - 1 \end{array}$$

Simplifying with the numerical values of the annotation α and using the substitutions (C1') and (C2'):

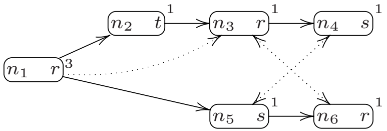
$$\begin{array}{lclccc} T_t - k_1 & \geq & 0 & T_r & = & k_1 \\ T_t - k_1 - k_2 & \geq & 0 & T_s & = & k_2 \end{array}$$

This system is clearly satisfiable, as shown by picking $k_1 = k_2 = 1$, and $T_r = 1$, $T_s = 1$ and $T_t = 2$. In other words, the following sequence leads to an unavoidable deadlock: (1) let resources be $T_r = 1$, $T_s = 1$ and $T_t = 2$; (2) two tasks are spawned, one instance (P_1) of call graph G_1 and another (P_2) of call graph G_2; (3) P_1 advances through n_{11}, and then P_2 advances to n_{21}; (4) finally, P_1 advances to n_{12} and P_2 enters n_{22}. At this point $t_r = t_s = 0$, and consequently none of the tasks can proceed independently to completion, so the deadlock is unavoidable. The execution of this example is depicted graphically in Fig. 5.

The previous discussion shows that if we violate the annotation condition, even if we come up with a set of resources ($T_r = 1$, $T_s = 1$ and $T_t = 1$) that avoids deadlock, if we allocate more resources ($T_r = 1$, $T_s = 1$ and $T_t = 2$) there is a possibility of deadlock.

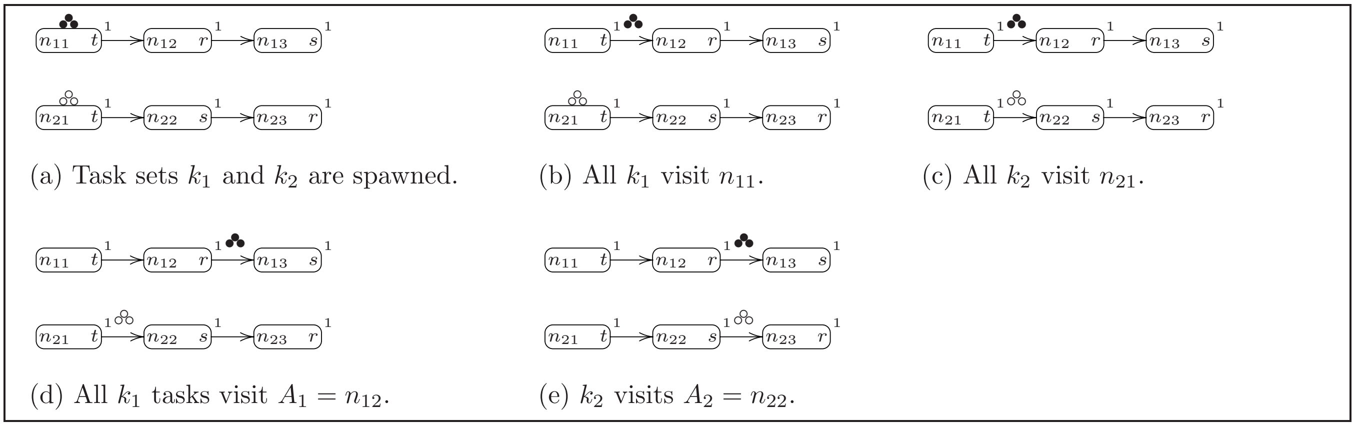
The following example shows a more sophisticated scenario, were paths leading to the nodes causing the deadlock have common ancestors:

Example 3. *Consider a scenario with a single call graph G with annotations $\alpha(n_1) = 3$, and $\alpha(n_2) = \alpha(n_3) = \alpha(n_4) = \alpha(n_5) = \alpha(n_6)$ as we show in the following global call graph:*



There is a simple dependency cycle $\langle[n_3, n_5], [n_4, n_6]\rangle$, which generates the paths $\pi_1 : (n_1, n_2, n_3)$ and $\pi_2 : (n_1, n_5)$. Path π_1 represents the sequence of nodes that are visited prior to n_3, while path π_2 contains the sequence ending in n_5. Observe that node n_1 is shared among the two paths.

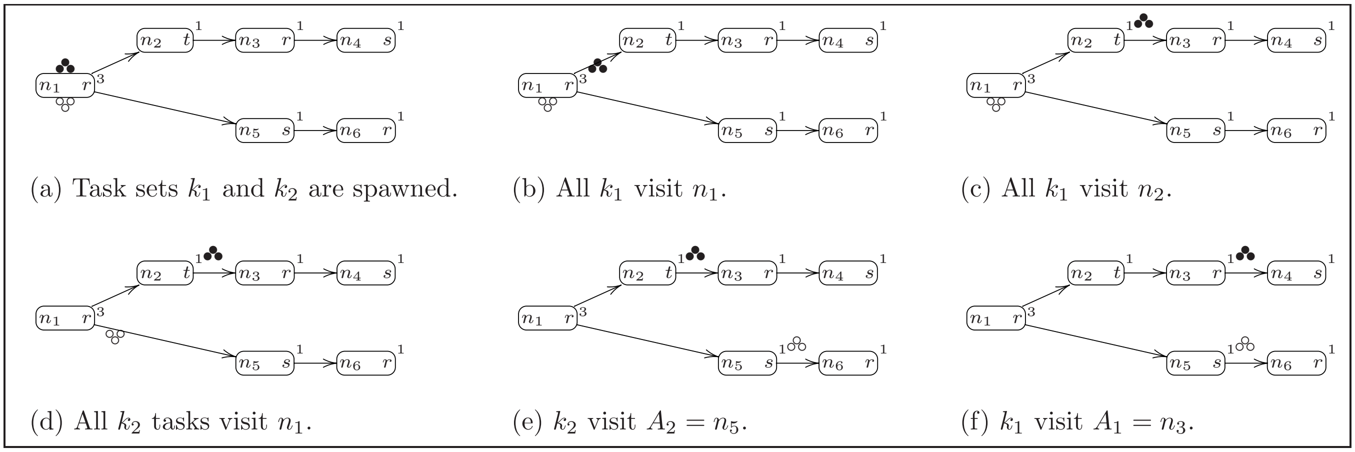
Let k_1 tasks follow path π_1 and k_2 tasks follow π_2. The set of visits for these two paths is $H = \{(n_1, k_1), (n_2, k_1), (n_3, k_1), (n_1, k_2), (n_5, k_2)\}$. One admissible total order is: $(n_1, k_1) < (n_2, k_1) < (n_1, k_2) < (n_5, k_2) < (n_3, k_1)$. This order corresponds to the run displayed in Fig 6.



(a) Task sets k_1 and k_2 are spawned. (b) All k_1 visit n_{11}. (c) All k_2 visit n_{21}.

(d) All k_1 tasks visit $A_1 = n_{12}$. (e) k_2 visits $A_2 = n_{22}$.

Figure 5. The run constructed in Example 2. The k_1 tasks are denoted by ♣and the k_2 tasks by ♧.



(a) Task sets k_1 and k_2 are spawned. (b) All k_1 visit n_1. (c) All k_1 visit n_2.

(d) All k_2 tasks visit n_1. (e) k_2 visit $A_2 = n_5$. (f) k_1 visit $A_1 = n_3$.

Figure 6. The run constructed in Example 3.

The corresponding set of constraints is:

$$\begin{aligned}
T_r - k_1 &\geq \alpha(n_1) - 1\\
T_t - k_1 &\geq \alpha(n_2) - 1\\
T_r - k_1 - k_2 &\geq \alpha(n_1) - 1\\
T_s - k_2 &= \alpha(n_5) - 1\\
T_r - (k_1 + k_2) - k_1 &= \alpha(n_3) - 1
\end{aligned}$$

which rewrite, according to (C1') and (C2'), into:

$$\begin{array}{rclcrcl}
k_2 + k_1 & \geq & 2 & \quad & T_s & = & k_2\\
T_t - k_1 & \geq & 0 & & T_r & = & k_1 + k_2 + k_1\\
k_1 & \geq & 2 & & & &
\end{array}$$

This system of equations is clearly satisfiable. One possible solution is $k_1 = 2$, $k_2 = 1$, and $T_r = 5$, $T_s = 1$, $T_t = 2$. In other words, if the reactors r, s and t have initially available 5, 1 and 2 threads (resp.), then 2 tasks can be spawned to follow the path π_1, and 1 to follow π_2 which causes a deadlock.

If a scenario violates the annotation condition a deadlock is possible:

Theorem 11. *If an annotation has dependency cycles then given enough resources a deadlock is reachable.*

Proof. Let $\mathcal{S}$ be a system and α an annotation such that the global call graph $G_{\mathcal{S},\alpha}$ has dependency cycles. Consider a simple cycle C, which by Lemma 7 always exists. There is an admissible order for visiting the nodes in C: the order $<$ where first all internal nodes of every path π_i are visited in topological order, and then all A_i are visited is admissible.

Using any admissible order the generated set of constraints (C1') and (C2') is satisfiable. The following values of k_i and T_r satisfy all the equations: (1) if a reactor r does not appear in any equation then let $T_r = 1$. (2) Take k_i to be the largest value of the right hand side of any formula $\varphi_{(n,k_i)}$ where k_i appears. This way, all (C2') are satisfied. (3) Compute the values of T_{r_i} using (C1'). (4) Finally, if some reactor r is visited in some intermediate node but no A_i resides in r, simply pick T_r to be the addition of all other elements appearing in all equations involving T_r. All these are of type (C2) which are then satisfied. □

Using the same construction we can show that Φ is still satisfiable even if we add extra constraints of the form $T_r > c$ for constants c. Therefore, given any set of resources, a scenario with reachable deadlocks can always be built by allocating more resources.

7. Conclusions

We have presented an efficient method to compute optimal annotations that meet the annotation condition. We have also shown that the annotation condition captures the essence of deadlock avoidance: if this condition is violated then by increasing the resources of a deadlock free system a deadlock can be reached.

The deadlock avoidance schema presented here assumes that call graphs do not have cycles. Even though this is still useful for DRE systems, it is an important open problem to see under which conditions cyclic global call graphs, for example with simple forms of recursion, can be handled. It is also interesting to see to what extent these techniques can be applied to FMSs, where efficient variations of (centralized) deadlock avoidance have been developed in recent years.

References

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

[2] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers," *CACM Special Issue on CORBA*, vol. 41, no. 10, 1998.

[3] V. Subramonian, G. Xing, C. Gill, C. Lu, and R. Cytron, "Middleware specialization for memory-constrained networked embedded systems," in *Proc. RTAS'04*, 2004.

[4] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons, 2003.

[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," *CACM*, vol. 12, pp. 373–377, 1969.

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

[8] R. C. Holt, "Some deadlock properties of computer systems," *ACM Computing Surveys*, vol. 4, pp. 179–196, 1972.

[9] T. Araki, Y. Sugiyama, and T. Kasami, "Complexity of the deadlock avoidance problem," *2nd IBM Symp. on Math. Foundations of Computer Sci.*, pp. 229–257, 1971.

[10] S. A. Reveliotis, M. A. Lawley, and P. M. Ferreira, "Polynomial-complexity deadlock avoidance policies for sequential resource allocation systems," *IEEE Trans. on Automatic Control*, vol. 42, no. 10, pp. 1344–1357, 1997.

[11] Z. A. Banaszak and B. H. Krogh, "Deadlock avoidance in flexible manufacturing systems with concurrently competing process flow," *IEEE Trans. on Robotics and Automation*, vol. 6, no. 6, pp. 724–734, 1990.

[12] K. Xing, B. Hu, and H. Chen, "Deadlock avoidance policy for petri-net modeling of flexible manufacturing systems with share resources," *IEEE Trans. on Automatic Control*, vol. 41, no. 2, pp. 289–295, 1996.

[13] P. M. Merlin and P. J. Schweitzer, "Deadlock avoidance in store-and-forward networks–I: Store-and-forward deadlock," *IEEE Trans. on Comm.*, vol. 28, no. 3, 1980.

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

[15] M. Singhal, "Deadlock detection in distributed systems," *IEEE Computer*, vol. 22, no. 11, pp. 37–48, 1989.

[16] C. Sánchez, H. Sipma, V. Subramonian, C. Gill, and Z.Manna, "Thread allocation protocols for distributed real-time and embedded systems," in *Proc. of FORTE'05*, 2005.

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