On Efficient Deadlock Avoidance for Distributive Recursive Processes

César Sánchez, Henny B. Sipma, and Zohar Manna

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

Abstract. We study deadlock avoidance for resource allocation in distributed systems. While a general solution of distributed deadlock avoidance is considered impractical, we propose an efficient solution of the important particular case where the possible sequences of remote calls, modeled as call graphs, are known a-priori. The algorithm presented here generalizes to recursive processes our earlier work on deadlock avoidance for non-recursive call-graphs.
An essential part of this generalization is that when a new process is created, it announces the *recursive depth* corresponding to the number of times a recursive call can be performed in any possible sequence of remote invocations.

1 Introduction and Related Work

We study the problem of deadlock avoidance in distributed systems. In particular, we focus on distributed computations that can require invocations at different sites; at each of them a local resource must be obtained to proceed. Moreover, we assume that once a resource is obtained, it is locked at least until the return of all subsequent remote calls. If the computation flow performs a sequence of calls that arrives back at a site previously visited then a new resource is needed. This model arises, for example, in distributed real-time and embedded (DRE) architectures, when the policy *WaitOnConnection* is used to deal with nested up-calls (see, for example, [9, 8, 13]). In classical concurrency terminology, we are dealing with (distributed) counting semaphores with nested allocation/deallocation patterns.

We assume that all resources are finite and fixed a-priori, so deadlocks can be reached unless some mechanism controls the assignment of resources to processes. There are two classes of algorithms that implement deadlock-free assignment: deadlock prevention and deadlock avoidance. Deadlock detection is a third mechanism, very common in data-bases, but it is not relevant in this context because we do not assume that processes must guarantee any transactional semantics, or that their actions can be rolled back.

With *deadlock prevention* the possibility of a deadlock is broken by eliminating one of the necessary causes of the circular contention, at the price of decreasing the concurrency. One simple deadlock prevention method, known as "monotone locking" and widely used in practice (see [1]), first determines an arbitrary total order on the set of resources. At run-time, processes acquire the resources in increasing order. Some concurrency is lost, since some resources may have to be acquired—and therefore locked—before they are needed, but all allocation decisions can be taken locally. Therefore, no extra communication is required. A drawback of this approach is that programming discipline must be enforced, and its synthesis or verification is not easily automatizable.

Deadlock avoidance methods, on the other hand, make decisions about resource allocations based on whether granting a resource is *safe*, that is, whether the controller has an implementable strategy to enforce that all continuations of the concurrent executions are deadlock-free. To make this check feasible, these methods use extra information that processes publish when they are created. For example, in the classical (centralized) deadlock avoidance algorithm (the Banker's algorithm and sequels, see [3–5, 12, 10]) a process advertises the maximum number of resources of each type that it can request. A *general* solution of distributed deadlock avoidance, however, is considered impractical (see [11]). The reason is that—in some situations—different sites must agree to determine whether an allocation is safe, which requires the run of some distributed agreement algorithm. The communication cost involved usually outweighs the benefit gained by using deadlock avoidance as opposed to a simple and locally implementable deadlock prevention mechanism.

In this paper we present a distributed deadlock avoidance scheme that uses only tests over local data. Such a solution is possible with the extra assumption that the set of all possible sequences of remote invocations that processes can follow is known a priory. This is the case in many distributed scenarios, like DREs or flexible manufacturing systems (FMS). In DREs, for example, we envision a development cycle where all the information about call dependencies is extracted from components. This is then used during configuration to tailor the deadlock avoidance algorithm into efficient, and provably correct (i.e., deadlock-free) code that controls the resource allocations. Note that the nature of the computations, or the values they compute is not relevant here. Only the pattern of remote calls, that determines the resource allocations, and the fact that all local computations terminate is used.

Similarly, in [2], a resource allocation controller that provides deadlock avoidance is generated directly from the components' code, using game theoretic techniques. The possible allocation/deallocation patterns are richer than the ones covered here. However, they assume that the controller is centralized, so it is not directly applicable to distributed systems without the use of a consensus algorithm.

The rest of this document is structured as follows. Sections 2 and 3 summarize the computational model and the necessary notation. Section 4 shows how to generate minimal annotations for recursive call graphs. Section 5 studies the relation between node annotations at different recursive heights. Finally, Section 6 presents an open problem and Section 7 contains the conclusions.

2 Model of Computation

A distributed system is modeled as a pair $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ consisting of 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 manage a necessary and scarce local resource, for example a finite pool of threads or execution contexts. A call graph specification $\mathcal{G} : (V, \rightarrow)$ is a graph that models the possible flows of computations. A call graph node describes both the computation to be performed at a site and the resource needed. Each node has a unique name (for example, the method name) and a site associated with it. If $n = (f, r)$, we say that node n executes computation f and *resides* in site r. An edge $n \rightarrow m$ denotes a possible remote invocation; in order to complete the computation modeled by n the result of a call to m may be needed. If this call is performed, the resource associated with n will be locked at least until the invocation of m returns. If the computation name is unimportant we use $n{:}r$ instead of $n = (f{:}r)$ to represent that node n resides in site r. We 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 nodes. We use $n \equiv_{\mathcal{R}} m$ to indicate that nodes n and m reside in the same site.

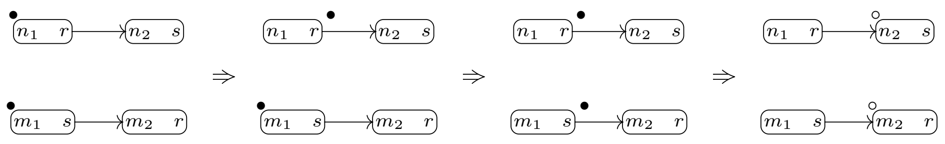
The execution of a system consists of processes, which can be created dynamically, executing computations that only perform remote calls according to the call graph specification. When a new process is spawned it announces its initial node. Incoming invocations require a new resource to run, while call returns perform a release.

Deadlocks can be reached if all requests are immediately granted, since resources are finite in each site and we impose no restriction on the topology of the call graph specification or the number of process instances.

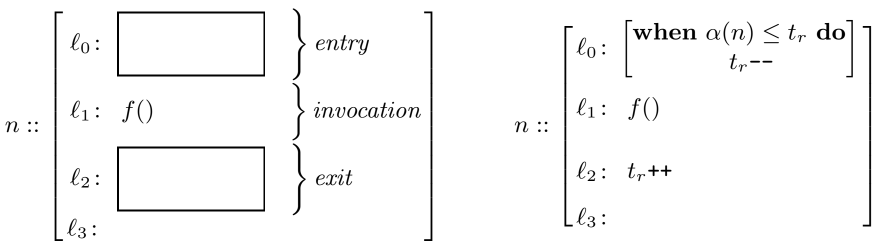
Example 1. The following diagram represents a system with reachable deadlocks if no controller is used. The set of sites is $\mathcal{R} = \{r, s\}$, the set of nodes is $V = \{n_1, n_2, m_1, m_2\}$, and the call graph edges are:

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

Let sites s and r have available exactly one unit of resource. If two processes are spawned, one instance of n_1 and one instance of m_1, all resources in the system will be locked after each process starts executing its root node:



Consequently, the allocation attempts for n_2 and m_2 nodes will be blocked indefinitely, so none of the two processes can progress. □



(a) Protocol Schema.

```
n ::
  ℓ0 :  when $\alpha(n) \leq t_r$ do
            $t_r$--
  ℓ1 :  $f()$
  ℓ2 :  $t_r$++
  ℓ3 :
```

(b) The protocol Basic-P.

Fig. 1. Protocol Schema and protocol Basic-P for node $n = (f{:}r)$.

A deadlock avoidance algorithm is a mechanism that controls the allocation of resources guaranteeing that no deadlock can be reached. Our deadlock avoidance algorithm consists of two parts:

1. a computation of *annotations* of the call graph nodes. This calculation is carried out statically. The annotation functions we consider are maps from nodes to positive numbers $\alpha : V \mapsto \mathbb{N}^+$.
2. a *protocol*: a piece of code that ensures, at runtime, that allocations and deallocations are safe. It consists of two stages: one that runs when a resource is requested, and another that executes when it is released. We want protocols that only inspect and modify local variables.

A schematic view of a protocol is shown in Fig. 1(a). 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 gain its desired resources.

3 Non-recursive Call Graphs

In [7] we present a deadlock avoidance algorithm for systems with only non-recursive call graphs. As background for the algorithm that covers recursive processes, we summarize the results in [7]. The call graph specification $\mathcal{G}$ for non-recursive processes can be described as a finite family of trees[1] $G_i : (V_i, \rightarrow_i)$, with node sets pairwise disjoint. The specification is then composed as $\mathcal{G} : (V, \rightarrow)$, with $V = \cup_i V_i$ and $\rightarrow = \cup_i \rightarrow_i$.

The simplest deadlock-avoidance protocol is Basic-P, shown in Fig. 1(b). The annotation of node n is denoted by $\alpha(n)$. The variable t_r, local to site r, controls the counting semaphore. It maintains the number of available units of resource available at site r, and it is initialized with the maximum amount of the resource T_r. If the annotation function α satisfies some properties, Basic-P prevents all deadlocks. To capture these properties we define the notion of an *annotated graph*.

Given a system $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ and an annotation α, the annotated graph $\mathcal{G}_\alpha : (V, \rightarrow, \dashrightarrow)$ is the graph formed by taking $\mathcal{G}$ and adding an edge $n \dashrightarrow m$ for

[1] all results can be extended to arbitrary DAGs.

every two nodes n and m that reside in the same site and satisfy $\alpha(n) \geq \alpha(m)$. If Basic-P is used, $n \dashrightarrow m$ captures whether a process executing m can make a process trying to gain access to n wait. Cycles in the graph that contain at least an $\rightarrow$ edge are called *dependency cycles.* The Annotation Theorem establishes when a protocol like Basic-P prevents deadlocks.

Theorem 1 (Annotation Theorem [7]). *If there is no dependency cycle in the annotated graph, and* Basic-P *is used to control allocations, then all runs are deadlock-free.*

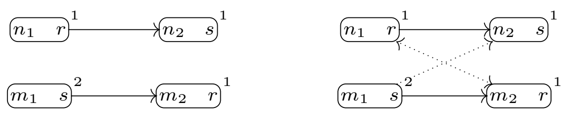
The absence of dependency cycles in the previous theorem is called the "Annotation Condition." To maximize the concurrency, annotations must be minimized, but creating cycles compromises deadlock-freedom. One possible annotation that satisfies the Annotation Condition is the *height* of a node in its call tree. However, in general, this annotation is far from optimal. In [6] we showed how minimal annotations can be computed. An annotation is minimal if no value can be decreased without violating the Annotation Condition. One algorithm that computes minimal annotations is:

Algorithm 1 (Minimal Annotations for Non-Recursive Graphs).

```
 1: {Order $N$ with $<$, a reverse topological order}
 2: {Let Below($S$) = $\{m \mid n \rightarrow^{+} m$ for some $n$ in $S\}$}
 3: {Let Reach($S$) = $\{m \mid \alpha(n) \geq \alpha(m)$ and $m < n$, for some $n$ in $S\}$}
 4: for $n = n_1$ to $n_{|N|}$ do
 5:     $S \leftarrow$ Below($n$)
 6:     repeat
 7:         $S \leftarrow S \cup$ Below(Reach($S$))
 8:     until fix-point
 9:     $\alpha(n) \leftarrow 1 + max\{\alpha(m) \mid m \in S$ and $m \equiv_{\mathcal{R}} n\}$
10: end for
```

This algorithm orders the nodes in some reverse topological order (descendants are always visited before ancestors), and calculates the annotations following that order. Each value is the minimum that creates no dependency cycle with previously calculated nodes. The algorithm first computes the set of nodes reachable following paths containing at least one edge in $\rightarrow$ (lines 6 to 8). Then, to prevent any of these paths from forming a cycle, it picks the minimum value that is sufficiently large (line 9).

Example 2. Consider the the system of Example 1 and total order $n_2 < m_2 < n_1 < m_1$. The annotations generated are shown on the left, and the annotated call graph on the right:



Basic-P with acyclic annotations guarantees deadlock-freedom, since the annotation $\alpha(m_1) = 2$ reserves the last thread in s for node n_2. If there is some process running in n_1 it can terminate and break the potential circular wait. □

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

Fig. 2. The protocol BasicRec-P for instance (n, k) of node $n = (f{:}r)$ at level k, with annotation $\alpha(n, k)$

4 Recursive Call Graphs

In this section we extend our deadlock avoidance schema to systems with recursive calls, that is, systems with call graphs that have cycles. The approach is similar to that presented in the previous section: the protocol only refers to local data and to static information about the call graphs provided a-priori. The only difference is that in addition the protocol refers to the recursion height of a function call. The new protocol, BasicRec-P, is shown in Fig. 2, where $\alpha(n, k)$ is the annotation of n, the node representing the function call, and k, the recursion height of the function call. We assume that each process at each function call can provide the protocol with its recursion height.

For the non-recursive call graphs presented before $\alpha(n)$ was precomputed for all nodes in the call graph specification and stored as part of the protocol. Execution of the protocol only involved looking up the annotation as function calls arrived. Given a maximum recursion height, k_{max}, we can, in principle, follow the same approach for recursive call graphs, that is, precompute the values of α for all nodes n for all k up to k_{max} and store them as part of the protocol. This is clearly not practical for large values of k_{max}. The alternative is to compute $\alpha(n, k)$ on the fly when needed. In DRE systems this is practical, however, only if α can be computed efficiently.

Below we will first show, given a call graph specification $\mathcal{G}$, how a minimal $\alpha(n, k)$ can be computed a-priori for any node n and recursion height k. This algorithm, however, is not suitable for on on-the-fly computation. Then, in Section 5 we propose some more alternatives for computing $\alpha(n, k)$ efficiently at run-time.

Call graph specifications are again a family of call trees $G_i : (V_i, \rightarrow_i)$, equipped with an additional set of edges $\twoheadrightarrow$. The set $V = \cup_i V_i$ is the set of nodes, and $\rightarrow = \cup_i \rightarrow_i$ is the set of *descending* edges. Edges in $\twoheadrightarrow$ are called *recursive* edges and can connect any two nodes. We use $\mapsto$ to denote edges in either $\rightarrow$ or $\twoheadrightarrow$. Call graph trees are composed into a call graph specification $\mathcal{G} : (V, \rightarrow, \twoheadrightarrow)$. To distinguish between occurrences of a node n at different recursive heights we define an *instance* of n at recursive height k as the pair (n, k). When the recursive height and the node are not relevant we use a, a_1, b_1,... to denote instances.

Definition 1 (Instance Call Graph). *Given a recursive call graph specification* $\mathcal{G} : (V, \to', \twoheadrightarrow')$ *an instance call graph* $\widehat{\mathcal{G}} : (I, \to, \twoheadrightarrow)$ *is a graph where*
1. $I = V \times \mathbb{N}$ *is the set of instances,*
2. $(n, k) \to (m, k)$ *exists precisely when* $n \to' m$ *occurs in* $\mathcal{G}$*, and*
3. $(n, k) \twoheadrightarrow (m, k - 1)$ *exists precisely when* $n \twoheadrightarrow' m$ *occurs in* $\mathcal{G}$*.*

We also use $\mapsto$ for $\to \cup \twoheadrightarrow$ on instances.

Example 3. The following diagram, on the left, shows a recursive specification containing nodes $\{n_1, n_2, n_3, n_4, n_5\}$. On the right we depict the graph with only the descending edges:

$$\begin{tikzcd}
& n_2\quad s \arrow[r] & n_3\quad r \arrow[dll, two heads, bend right] \\
n_1\quad t \arrow[ur] \arrow[dr] & & \\
& n_4\quad r \arrow[r] & n_5\quad s \arrow[ull, two heads, bend left]
\end{tikzcd}
\qquad
\begin{tikzcd}
& n_2\quad s \arrow[r] & n_3\quad r \\
n_1\quad t \arrow[ur] \arrow[dr] & & \\
& n_4\quad r \arrow[r] & n_5\quad s
\end{tikzcd}$$

The portion of the instance call graph up-to recursive height 3 is:

$$\begin{tikzcd}
& (n_2,3) \arrow[r] & (n_3,3) \arrow[dr, two heads] & & (n_2,2) \arrow[r] & (n_3,2) \arrow[dr, two heads] & & (n_2,1) \arrow[r] & (n_3,1) \arrow[dr, two heads] & & (n_2,0) \arrow[r] & (n_3,0) \\
(n_1,3) \arrow[ur] \arrow[dr] & & & (n_1,2) \arrow[ur] \arrow[dr] & & & (n_1,1) \arrow[ur] \arrow[dr] & & & (n_1,0) \arrow[ur] \arrow[dr] & & \\
& (n_4,3) \arrow[r] & (n_5,3) \arrow[ur, two heads] & & (n_4,2) \arrow[r] & (n_5,2) \arrow[ur, two heads] & & (n_4,1) \arrow[r] & (n_5,1) \arrow[ur, two heads] & & (n_4,0) \arrow[r] & (n_5,0)
\end{tikzcd}$$

The algorithm to compute minimal annotations requires a specific order on the nodes.

Definition 2 (admissible order). *A total order* $<$ *on the set of instances* I *is* admissible *whenever*
1. *it is well-founded (it has no infinite descending chains), and*
2. *it respects the reverse topological order (if* $a \mapsto b$ *then* $b < a$*).*

The two conditions ensure that for every instance a the set of smaller instances is finite and it contains all a's descendants. An example of an admissible order is a lexicographic order that first compares the recursion height, and then uses a reverse topological total order on the (finite) set of nodes. For a given admissible order the annotation for an instance a can be computed with:

Algorithm 2 (Minimal Annotations for Recursive Graphs).

```
1: {Let Below($S$) = $\{b \mid a \mapsto^+ b$ for some $a$ in $S\}$}
2: {Let Reach($S$) = $\{b \mid \alpha(a) \geq \alpha(b)$ and $b < a$ for some $a$ in $S\}$}
3: for $i$ in $a_1 < a_2 < \cdots < a$ do
4:     $S \leftarrow$ Below($i$)
5:     repeat
6:         $S \leftarrow S \cup$ Below(Reach($S$))
7:     until fix-point
8:     $\alpha(i) \leftarrow 1 + max\{\alpha(b) \mid b \in S$ and $b \equiv_{\mathcal{R}} i\}$
9: end for
```

Given an instance call graph $\widehat{\mathcal{G}} : \langle I, \rightarrow, \twoheadrightarrow\rangle$ and an annotation α on I, the annotated recursive graph $\widehat{\mathcal{G}}_\alpha : (I, \rightarrow, \twoheadrightarrow, \dashrightarrow)$ is defined by adding an edge $a \dashrightarrow b$ whenever $\alpha(a) \geq \alpha(b)$ and $a \equiv_{\mathcal{R}} b$. We use $\rightsquigarrow$ as $\rightarrow \cup \twoheadrightarrow \cup \dashrightarrow$. The following is a version of the Annotation Theorem for recursive call graphs:

Theorem 2 (Annotation Theorem for Recursive Call Graphs). *If there is no dependency cycle in the annotated recursive graph, and* BasicRec-P *is used to control allocations, then all runs are deadlock free.*

Proof. Let us proceed by contradiction. Let $\widehat{\mathcal{G}}_\alpha$ be an annotated recursive graph that does not contain dependency cycles but allows runs with deadlocks. Consider one such run and let a be the highest instance occurring in any process involved in the execution. We build a non-recursive specification $\mathcal{G}' : (V', \rightarrow')$ as follows:
- V' is the set of instances smaller than a, $V' = \{b \mid b \leq a\}$.
- There is an edge $a \rightarrow' b$ in $\mathcal{G}'$, whenever $a \mapsto b$ in the original recursive specification $\mathcal{G}$.

Note how the finite set of instances V' that are involved in the execution becomes the set of nodes of $\mathcal{G}'$. Now, let the annotation α' be the restriction of α to V'. The (non-recursive) annotated call graph $\mathcal{G}'_{\alpha'}$ does not contain any dependency cycle, so the system described by $\mathcal{G}'$ is deadlock free according to Theorem 1. However, a deadlock can be reached in $\mathcal{G}'$ following the same execution steps that led to a deadlock in $\mathcal{G}$, which is a contradiction. $\Box$

Clearly, Algorithm 2 is not suitable to compute values of $\alpha(n, k)$ on the fly. The example below suggests an alternative way of computing the annotations.

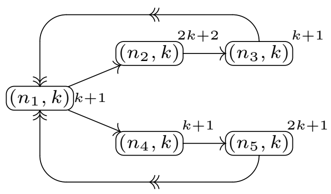
Example 4. Consider the specification call graph in Example 3. Let $<$ be the lexicographic order on instances that first compares the recursive height and then the nodes according to n_5, n_4, n_3, n_2, n_1. This is clearly a well-founded reverse topological order, so it is admissible for Algorithm 2. The annotations corresponding to the first 3 recursive heights are:

$$\begin{tikzcd}
 & (n_2,3)^{8} \arrow[r] & (n_3,3)^{4} \arrow[dr, two heads] & & (n_2,2)^{6} \arrow[r] & (n_3,2)^{3} \arrow[dr, two heads] & & (n_2,1)^{4} \arrow[r] & (n_3,1)^{2} \arrow[dr, two heads] & & (n_2,0)^{2} \arrow[r] & (n_3,0)^{1} \\
(n_1,3)^{4} \arrow[ur, tail] \arrow[dr, tail] & & & (n_1,2)^{3} \arrow[ur, tail] \arrow[dr, tail] & & & (n_1,1)^{2} \arrow[ur, tail] \arrow[dr, tail] & & & (n_1,0)^{1} \arrow[ur, tail] \arrow[dr, tail] & & \\
 & (n_4,3)^{4} \arrow[r] & (n_5,3)^{7} \arrow[ur, two heads] & & (n_4,2)^{3} \arrow[r] & (n_5,2)^{5} \arrow[ur, two heads] & & (n_4,1)^{2} \arrow[r] & (n_5,1)^{3} \arrow[ur, two heads] & & (n_4,0)^{1} \arrow[r] & (n_5,0)^{1}
\end{tikzcd}$$

It is easy to see that the recurrences that define the values are:

$$\begin{array}{llll}
\alpha(n_1,k) = \alpha(n_1,k-1) & +1 & \qquad & \alpha(n_1,0) = 1 \\
\alpha(n_2,k) = \alpha(n_5,k) & +1 & & \\
\alpha(n_3,k) = \alpha(n_3,k-1) & +1 & & \alpha(n_3,0) = 1 \\
\alpha(n_4,k) = \alpha(n_4,k-1) & +1 & & \alpha(n_4,0) = 1 \\
\alpha(n_5,k) = \alpha(n_2,k-1) & +1 & & \alpha(n_5,0) = 1
\end{array}$$

These recurrences have a simple solution, given by periodic affine functions shown on the left.



```
(* pseudo code for n_2=(f,s) *)
 k     := me.recursive_height;
 alpha := 2 * k + 2;
 cond_wait(less_eq_than(alpha,t_s),
           t_s_counting_semaphore);
 f();
 unlock(t_s_counting_semaphore);
```

In cases where there is a simple solution like this, the value of $\alpha(n,k)$ for any given k could be computed very efficiently at run-time, using the corresponding periodic function. For example, the pseudocode for the protocol BasicRec-P at node n_2 is shown above on the right. □

In the rest of the paper we show how to synthesize and verify, for each node n, a function f^n that computes the annotation $\alpha(n,k)$ efficiently, once the recursive height k is given.

The following lemma is used widely in our subsequent developments. All missing proofs appear in the appendix.

Lemma 1. *Given an acyclic and minimal annotation, for every node n, the annotations of its instances are monotonically non-decreasing on the recursion height, i.e., for all all positives λ, $\alpha(n,k+\lambda) \geq \alpha(n,k)$.*

5 Verifying Annotations

Ideally, minimal annotations would be computed by the protocol on the fly by means of a simple function, given an admissible order on the nodes of the call graph specification. Unfortunately, no such functions exist in general, for arbitrary admissible orders, as shown in the next section. We conjecture, however, that for simple orders, such as the lexicographic order used in the previous section, such functions do exist, but a method for generating them automatically is still an open problem. In this section we solve the easier problem of verifying that a given set of functions corresponds to an acyclic and minimal annotation of instances.

Below, we first establish necessary and sufficient conditions for such functions and then show that these conditions can be effectively checked for a particular class of functions, namely monotonically non-decreasing periodic functions.

5.1 Necessary and Sufficient Conditions

A *reachability function* $f^n_r : \mathbb{N} \mapsto \mathbb{N} \cup \{\infty\}$ for a node n and site r (not necessarily the site where n resides) maps a recursive height k into the maximum annotation of an instance residing in r that (n,k) can reach. In particular if n resides in r, $f^n_r(k)$ corresponds to the annotation of (n,k), that is, $\alpha(n,k) = f^n_r(k)$.

Example 5. The need for ∞ is exemplified by the following call graph specification:

$$\begin{tikzcd} \boxed{n \quad r} \arrow[r] & \boxed{m \quad s} \arrow[loop above, ->>] \end{tikzcd}$$

Since $\alpha(n, k) = 1$ for all k, then $(n, k) \rightsquigarrow (n, 1)$ and $(n, 1) \rightsquigarrow (n, k)$. Also, $\alpha(m, k) = k + 1$. Consequently, the associated reachability functions are:

$$f^n_r(k) = 1 \qquad f^n_s(k) = \infty \qquad f^m_r(k) = 0 \qquad f^m_s(k) = k + 1.$$

Therefore, since the set of annotations of nodes reachable by (n, k) has no upper bound, the first infinite ordinal ∞ is the best upper-bound. □

Given an annotation α, its associated reachability functions is defined as

$$f^n_r(k) \overset{\text{def}}{=} \mathit{max}\{\alpha(b) \mid \mathit{site}(b) = r \text{ and } (n, k) \rightsquigarrow^* b\}.$$

Conversely, given a set of reachability functions $\{f^n_r\}$ containing one function for each node n and site r, we can construct its associated annotation by taking:

$$\alpha(n, k) \overset{\text{def}}{=} f^n_r(k) \qquad \text{for } r = \mathit{site}(n).$$

Decision Problem. *Given a set of reachability functions $\{f^n_r\}$ decide whether its associated annotation α is acyclic and minimal.*

To establish a necessary and sufficient set of conditions for acyclicity and minimality we first define the following notions. Let the class of an instance a

$$[a] \overset{\text{def}}{=} \{b \mid a \equiv_{\mathcal{R}} b \text{ and } \alpha(a) \geq \alpha(b)\}$$

stand for the set of instances that reside in the same site as a and have a lower or equal annotation. Let

$$M^n_r(k) \overset{\text{def}}{=} \mathit{max}\begin{pmatrix} \{f^m_r(k) & \mid n \rightarrow m\} \\ \{f^m_r(k-1) & \mid n \twoheadrightarrow m\} \end{pmatrix}$$

stand for the maximum of the values of the reachability functions of all successors of node n. The desired conditions are stated as follows. For all sites r, and instances (n, k) and (m, l):

C1. If $(n, k) \rightsquigarrow (m, l)$ then $f^n_r(k) \geq f^m_r(l)$.
C2. If n resides in r, then $f^n_r(k) = M^n_r(k) + 1$.
C3. If n does not reside in r, then $f^n_r(k) = \mathit{max}\{M^m_r(l) \mid (m, l) \in [(n, k)]\}$.

The conditions C1–C3 are necessary, as established by the following theorem.

Theorem 3. *Given an acyclic and minimal annotation, its associated reachability functions satisfy* C1–C3*.*

The conditions C1-C3 are also sufficient, as stated by Theorem 4. The proof of the theorem relies on the following Lemma.

Lemma 2. *Let $\{f_r^n\}$ be a set of reachability functions satisfying* C1–C3 *and α its associated annotations. Then, every instance (n,k) can reach all instances in r with annotations smaller or equal $f_r^n(k)$.*

Theorem 4. *If a set of reachability functions $\{f_r^n\}$ satisfies conditions* C1–C3 *then its associated annotation is acyclic and minimal.*

Proof. For acyclicity we reason by contradiction. Assume that the annotation α is not acyclic and let

$$(n,k) \rightsquigarrow (n_1,k_1) \rightsquigarrow \ldots \rightsquigarrow (n_l,k_l) \rightsquigarrow (n,k)$$

be a cycle. Condition C1 ensures that if $(n,k) \rightsquigarrow^* (m,l)$ then $f_r^n(k) \geq f_r^m(l)$. Since every instance in the cycle can reach any other instance, all of them have the same values of the reachability functions for all sites. To be a dependency cycle at least one edge must be of the form $(n_i,k_i) \mapsto (n_{i+1},k_{i+1})$. Then, by condition C2, $f_r^{n_i}(k_i) \geq f_r^{n_{i+1}}(k_{i+1}) + 1$ for the site r where (n_i,k_i) resides. Moreover, $f_r^{n_i}(k_i) < \infty$ since it is an annotation. Consequently, $f_r^{n_i}(k_i) > f_r^{n_{i+1}}(k_{i+1})$, which is a contradiction.

For minimality, by Lemma 2, some descendant of $(n,k):s$ can reach some instance that resides in site s and has annotation $f_s^n(k) - 1$. If the annotation of (n,k) were decreased, then a dependency cycle could be created. □

5.2 Periodic Reachability Functions

We solve the decision problem for a simple class of functions, namely monotonically non-decreasing periodic functions, that we simply call periodic functions:

Definition 3 (Periodic functions). *A periodic function f is described by positive parameters A, D and $\langle B_0, \ldots, B_{D-1}\rangle$ and is denoted as:*

$$f : A\frac{k}{D} + \langle B_0 \ldots, B_{D-1}\rangle$$

The parameters must satisfy $B_i \leq B_j$ when $i \leq j$, and $B_{D-1} \leq (A + B_0)$. The value of function f on input k is

$$f(k) = A\frac{k}{D} + B[i] \qquad \textit{for } i = (k \textit{ mod } D).$$

For example, the first ten values of the periodic function $f(k) = 3\frac{k}{2} + \langle 1, 2\rangle$ are:

k	0	1	2	3	4	5	6	7	8	9
$f(k)$	1	2	4	5	7	8	10	11	13	14

Periodic functions arise as solutions of recurrences of the form:

$$\begin{cases} f(k+D) = A + f(k) & \\ \qquad f(i) = B[i] & \text{for } i = 0 \ldots D-1 \end{cases}$$

The restrictions $B_i \leq B_j$ when $i \leq j$, and $B_{D-1} \leq (A + B_0)$, ensure that the function is monotonically non-decreasing, as enforced by Lemma 1 for annotations.

To solve the decision problem for this class of functions we must show that conditions C1–C3 can be effectively checked. We describe an algorithm that decides whether these conditions hold, based on basic properties of periodic functions.

***Checking* C1** : Condition C1 can be decomposed into:

C1.a If $(n,k) \rightarrow (m,k)$ then $f^n_r(k) \geq f^m_r(k)$.
C1.b If $(n,k) \twoheadrightarrow (m,k-1)$ then $f^n_r(k) \geq f^m_r(k-1)$.
C1.c If $(n,k) \dashrightarrow (m,l)$ then $f^n_r(k) \geq f^m_r(l)$.

C1.a and C1.b correspond to tests of the form

$$f^n_r(k) \overset{?}{\geq} f^m_r(l)$$

for two periodic functions, which can be checked. For C1.c, let n and m reside in s, and let τ_{nm} give, for every k, the maximum value for which $f^n_s(k) \geq f^m_s(\tau_{nm}(k))$. This map, called a *translation* function, is periodic and can be generated from the given functions. Finally, the test for C1.c is:

$$f^n_r(k) \overset{?}{\geq} f^m_r(\tau_{nm}(k)).$$

Again, the composition of two periodic functions can be generated and is periodic, so this check is decidable.

***Checking* C2** : First, consider the shift transformation that translates a periodic function one unit:

$$(\delta f^n_r)(k) \overset{\text{def}}{=} f^n_r(k-1).$$

Then, the maximum of the descendants of a node, M^n_r, can be defined by computing the maximum, component-wise, of the functions f^m_r for all the direct descendants, and the functions (δf^p_r) for all recursive descendants. The maximum of periodic functions is an eventually periodic function, and can be effectively computed. C2 holds for n, which resides in s if:

$$f^n_s(k) \overset{?}{=} M^n_s(k) + 1.$$

***Checking* C3** : The check for C3 starts by calculating the maximum value of descendants $M^n_r(k)$ as previously, except that now the calculation is carried out for all other sites r different than where n resides. This function is then refined with the maximum for all other nodes m residing in the same site as n, considering the translation of m with respect to n. The check is then:

$$f^n_r(k) \overset{?}{=} max\{M^m_r(\tau_{nm}(k)) \mid \text{for all } m \equiv_{\mathcal{R}} n\}.$$

Example 6. Consider the specification

$$\begin{tikzcd}
\boxed{n_1 \quad r} \arrow[r] & \boxed{n_2 \quad s} \arrow[l, bend right, "\ll"']
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{m_1 \quad r} \arrow[r] & \boxed{m_2 \quad s} \arrow[r] & \boxed{m_3 \quad r} \arrow[r] & \boxed{m_4 \quad s} \arrow[r] & \boxed{m_5 \quad r} \arrow[llll, bend left, "\ll"]
\end{tikzcd}$$

And the reachability functions:

$$\begin{array}{ll}
f_r^{n_1}(k) = 3(\frac{k}{2}) + \langle 2,3\rangle & f_s^{n_1}(k) = k+1 \\
f_r^{n_2}(k) = 3(\frac{k}{2}) + \langle 1,2\rangle & f_s^{n_2}(k) = k+1 \\
f_r^{m_1}(k) = 3k+3 & f_s^{m_1}(k) = 2k+2 \\
f_r^{m_2}(k) = 3k+2 & f_s^{m_2}(k) = 2k+2 \\
f_r^{m_3}(k) = 3k+2 & f_s^{m_3}(k) = 2k+1 \\
f_r^{m_4}(k) = 3k+1 & f_s^{m_4}(k) = 2k+1 \\
f_r^{m_5}(k) = 3k+1 & f_s^{m_5}(k) = 2k
\end{array}$$

We sketch how the decision procedure shows that the associated annotation is acyclic and minimal, illustrating the checks for node n_1. C2.b holds vacuously, and C1.a holds since:

$$\begin{array}{lll}
f_r^{n_1}(k) \geq f_r^{n_2}(k) & \text{because} & 3(\frac{k}{2}) + \langle 2,3\rangle \geq 3(\frac{k}{2}) + \langle 1,2\rangle \\
f_s^{n_1}(k) \geq f_s^{n_2}(k) & \text{because} & k+1 \geq k+1
\end{array}$$

For C1.c, we show the check for node m_5 (for m_1 and m_3 is analogous). Node m_5 results in a translation of $\tau_{n_1 m_5} = \frac{k}{2}$, and then:

$$\begin{array}{lll}
f_r^{n_1}(k) \geq f_r^{m_5}(\frac{k}{2}) & \text{because} & 3(\frac{k}{2}) + \langle 2,3\rangle \geq 3(\frac{k}{2}) + \langle 1,1\rangle \\
f_s^{n_1}(k) \geq f_s^{m_5}(\frac{k}{2}) & \text{because} & k+1 \geq 2(\frac{k}{2})
\end{array}$$

For C2, observe that $M_r^{n_1}(k) = f_r^{n_2}(k)$ and then:

$$f_r^{n_1}(k) = M_r^{n_2}(k) + 1 \qquad \text{because} \qquad 3(\tfrac{k}{2}) + \langle 2,3\rangle = 3(\tfrac{k}{2}) + \langle 1,2\rangle + 1$$

Finally, for C3, observe that $\tau_{n_1 m_1} = \frac{k-1}{2} - 1$ and $\tau_{n_1 m_3} = \frac{k}{2}$. Therefore:

$$\begin{aligned}
f_s^{n_1}(k) &\overset{?}{=} max\{M_s^{n_1}(k), \quad M_s^{m_1}(\tfrac{k-1}{2} - 1), \quad M_s^{m_3}(\tfrac{k}{2}), \quad M_s^{m_5}(\tfrac{k}{2})\} \\
&= max\{k+1, \qquad 2\tfrac{k-1}{2} - 1 + 2, \qquad 2\tfrac{k}{2} + 1, \qquad 2\tfrac{k}{2}\} \\
&= k+1
\end{aligned}$$

holds. Similar checks hold for the other nodes. Therefore, the associated annotation is acyclic and minimal. □

6 A Negative Result and an Open Problem

The method described in Section 5 allows us to automatically check whether a set of periodic functions generate an acyclic and minimal annotation. By running a number of case studies, we have found that for a simple (lexicographic or at least periodic) order the annotations are periodic. The method can be automated by running Algorithm 2 for several iterations, extracting the candidate reachability functions, and then checking these functions with the decision procedure. It is not true however that, given an *arbitrary* admissible order $<$ on instances, the annotations generated are simple (for example periodic) functions for all nodes.

Example 7. Consider the recursive call graph specification of Example 3 and the following two total orders on the nodes, $<_1$ and $<_2$:

$$\begin{array}{ll} <_1: & n_5\ , n_4\ , n_3\ , n_2\ , n_1 \\ <_2: & n_3\ , n_2\ , n_5\ , n_4\ , n_1 \end{array}$$

Let the admissible order $<$ be defined as follows. Two instances are ordered $(n, k) < (m, l)$ whenever

1. $k < l$, or
2. $k = l$, and k is not a power of 2 and $n <_1 m$, or
3. $k = l$, and k is a power of 2 and $n <_2 m$.

The annotations for the first four levels are depicted:

$$\begin{tikzcd}
& (n_2,3)^{5} \arrow[rr] & & (n_3,3)^{6} \arrow[dr] & & (n_2,2)^{4} \arrow[rr] & & (n_3,2)^{5} \arrow[dr] & & (n_2,1)^{2} \arrow[rr] & & (n_3,1)^{3} \arrow[dr] & & (n_2,0)^{1} \arrow[rr] & & (n_3,0)^{1} \\
(n_1,3)^{4} \arrow[ur] \arrow[dr] & & <_1 & & (n_1,2)^{3} \arrow[ur] \arrow[dr] & & <_2 & & (n_1,1)^{2} \arrow[ur] \arrow[dr] & & <_1 & & (n_1,0)^{1} \arrow[ur] \arrow[dr] & & <_1 & \\
& (n_4,3)^{7} \arrow[rr] & & (n_5,3)^{5} \arrow[ur] & & (n_4,2)^{5} \arrow[rr] & & (n_5,2)^{3} \arrow[ur] & & (n_4,1)^{4} \arrow[rr] & & (n_5,1)^{2} \arrow[ur] & & (n_4,0)^{2} \arrow[rr] & & (n_5,0)^{1}
\end{tikzcd}$$

It is easy to see that $\alpha(n_5, k) = \alpha(n_2, k-1) + 1$ and

$$\alpha(n_2, k) = \begin{cases} \alpha(n_2, k-1) + 1 & \text{if } <_1 \text{ is used in level } k \\ \alpha(n_5, k) + 1 & \text{if } <_2 \text{ is used in level } k \end{cases}$$

Since $\alpha(n_5, k) + 1 = \alpha(n_2, k-1) + 1$, the solution of the recurrence is $\alpha(n_2, k) = 2o_2 + o_1$, where o_1 stands for the number of times $<_1$ was used up to level k, and o_2 for the number of times $<_2$ was used. Therefore $\alpha(n_5, k) = k + \lfloor \log\ k \rfloor$ which cannot be expressed as a periodic function on k. □

Even though this example shows that not all annotations grow according to periodic functions, we conjecture that if the order is simple, then the annotations will also be simple.

Open Problem. *If the order $<$ is simple (expressible as a periodic function, which includes lexicographic orders) then the annotations calculated by Algorithm 2 have associated periodic functions.*

7 Conclusions

We have presented an efficient deadlock avoidance schema for distributed systems with recursive processes. The essential feature that makes an efficient algorithm possible is that processes announce: (1) the call-graph that describes the possible sequences of calls that they can perform, and (2) the maximum number of recursive calls. The deadlock avoidance solution is implemented by a protocol executed at runtime. The protocol is a simple extension of the guard sections of counting semaphores, and perform only tests and operations over local data.

The protocol is parameterized by an annotation of the call-graph nodes, computed statically. We have presented an algorithm that generates these annotations. Even though it is still an open problem whether the algorithm always generates annotations that follow some simple growth functions, we have shown how to prove that a candidate function corresponds to minimal annotations. These candidates can be extracted automatically from the call graph by running the annotation calculation algorithm for several levels.

References

1. Andrew D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.
2. Luca de Alfaro, Vsihwanath Raman, Marco Faella, and Rupak Majumdar. Code aware resource management. In *Proc. of EMSOFT'05*, pages 191–202, 2005.
3. Edsger W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.
4. Arie N. Habermann. Prevention of system deadlocks. *CACM*, 12:373–377, 1969.
5. James W. Havender. Avoiding deadlock in multi-tasking systems. *IBM Systems Journal*, 2:74–84, 1968.
6. César Sánchez, Henny B. Sipma, Zohar Manna, Venkita Subramonian, and Christopher Gill. On efficient distributed deadlock avoidance for distributed real-time and embedded systems. In *Proc. of IPDPS'06*, 2006.
7. César Sánchez, Henny B. Sipma, Venkita Subramonian, Christopher Gill, and Zohar Manna. Thread allocation protocols for distributed real-time and embedded systems. In *Proc. of FORTE'05*, pages 159–173, 2005.
8. Douglas C. Schmidt. Evaluating Architectures for Multi-threaded CORBA Object Request Brokers. *Comm. of the ACM Special Issue on CORBA*, 41(10), 1998.
9. Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* Wiley & Sons, New York, 2000.
10. Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts.* John Wiley & Sons, Inc., Sixth edition, 2003.
11. Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems.* McGraw-Hill, Inc., 1994.
12. William Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall, Third edition, 1998.
13. Venkita Subramonian, Guoliang Xing, Christopher D. Gill, Chenyang Lu, and Ron Cytron. Middleware specialization for memory-constrained networked embedded systems. In *Proc. of RTAS'04*, 2004.

A Proofs

The following lemma establishes a fact about minimal annotations that is used in subsequent proofs.

Lemma 3. *Given an acyclic and minimal annotation α, for every (n, k) with $\alpha(n, k) > 1$ there is a path of the form*

$$(n, k) \mapsto (n_1, k_1) \rightsquigarrow \cdots \rightsquigarrow (m, l)$$

for some $n \equiv_{\mathcal{R}} m$ with $\alpha(n, k) = \alpha(m, l) + 1$.

Proof. Since the annotation is minimal, if $\alpha(n, k)$ is decremented there would be a cycle that would not exist otherwise. Let

$$(n, k) \rightsquigarrow (n_1, k_1) \rightsquigarrow \cdots \rightsquigarrow (m, l) \rightsquigarrow (n, k)$$

be one such cycle. The incoming and outgoing edges to (n, k) cannot be both $\dashrightarrow$ or both $\mapsto$ since otherwise, a cycle would exist in α without decrementing, contradicting acyclicity. If the outgoing edge is $\dashrightarrow$ then the cycle is still a path in α without decrementing, again a contradiction. Therefore, $(m, l) \dashrightarrow (n, k) \mapsto (n_1, k_1)$, and

$$(n, k) \mapsto (n_1, k_1) \rightsquigarrow \cdots \rightsquigarrow (m, l)$$

is the desired path. $\square$

Lemma 1. *Given an acyclic and minimal annotation, for every node n, the annotations of its instances are monotonically non-decreasing on the recursion height, i.e., for all all positives λ, $\alpha(n, k + \lambda) \geq \alpha(n, k)$.*

Proof. Let $<$ be an arbitrary admissible order. We reason by complete induction on $<$. Let us assume that the result holds for all $(n', k') < (n, k)$. If $\alpha(n, k) = 1$, then since all annotations are at least 1, the result follows immediately. Otherwise, by Lemma 3 consider a semi-cycle:

$$(n, k) \mapsto (n_1, k_1) \rightsquigarrow \cdots \rightsquigarrow (n_m, k_m)$$

witnessing that $\alpha(n, k) = \alpha(n_m, k_m) + 1$. By the graph structure, since $(n, k) \mapsto (n_1, k_1)$, then also $(n, k + \lambda) \mapsto (n_1, k_1 + \lambda)$. Also, by inductive hypothesis, $\alpha(n_1, k_1 + \lambda) \geq \alpha(n_1, k_1)$, or in other words $(n_1, k_1 + \lambda) \dashrightarrow (n_1, k_1)$. Then we can build the semi-cycle

$$(n, k + \lambda) \mapsto (n_1, k_1 + \lambda) \dashrightarrow (n_1, k_1) \rightsquigarrow \cdots \rightsquigarrow (n_m, k_m),$$

which implies that $\alpha(n, k + \lambda) \geq \alpha(n_m, k_m) + 1 = \alpha(n, k)$ as desired. $\square$

Theorem 3. *Given an acyclic and minimal annotation, its associated reachability functions satisfy* C1–C3.

Proof. Let $\{f_r^n\}$ be reachability functions associated to annotation α. We proof the conditions separately:

1. Let $(n,k) \rightsquigarrow (m,l)$, and b be the instance with maximum annotation that resides in r and (m,l) can reach. Then $f_r^m(l) = \alpha(b)$. Since $(m,l) \rightsquigarrow^* b$, then $(n,k) \rightsquigarrow^* b$ as well. Therefore $f_r^n(k) \geq \alpha(b) = f_r^m(k)$.
2. First, $f_r^n(k) \geq \mu_r^{(n,k)} + 1$ since otherwise there would be a dependency cycle. If $f_r^n(k) = 1$ the result follows vacuously since $\mu_r^{(n,k)} \geq 0$. If $f_r^n(k) = 1$, by Lemma 3, there is a path
$$(n,k) \mapsto (n_1,k_1) \rightsquigarrow^* (n_l,k_l)$$
with $n \equiv_{\mathcal{R}} n_l$ and $\alpha(n,k) = \alpha(n_l,k_l) + 1$ Therefore $\mu_r^{(n,k)} \geq f_r^{n_1}(k_1) \geq \alpha(n_l,k_l)$, and then $f_r^n(k) \leq \mu_r^n(k) + 1$.
3. First, $f_r^n(k) \geq \mu_r^{[(n,k)]}$ since every path from a successor of $[(n,k)]$ can be extended to a path from (n,k). We must show that $f_r^n(k) \leq \mu_r^{[(n,k)]}$. Let b be the maximum instance that resides in r and is reachable from some node in the class $[(n,k)]$. If there is no such a b, then either no instance is reachable, so $f_r^n(k) = 0 = \mu_r^{[(n,k)]}$, or the set of reachable instances is not upper-bounded, but then $f_r^n(k) = \infty = \mu_r^{[(n,k)]}$. Consider a path from (n,k) to b
$$(n,k) \rightsquigarrow \cdots \rightsquigarrow b$$
and let (m,l) be the first node in the path that does not belong to $[(n,k)]$. Then, $f_r^m(l) \geq \alpha(b)$ and consequently $\mu_r^{[(n,k)]} \geq \alpha(b) = f_r^n(k)$.

□

Lemma 2. *Let* $\{f_r^n\}$ *be a set of reachability functions satisfying* C1–C3 *and* α *its associated annotations. Then, every instance* (n,k) *can reach all instances in* r *with annotations smaller or equal* $f_r^n(k)$.

Proof. By induction on $w(n,k) = \sum_r f_r^n(k)$, called the *weight* of (n,k). Note that if $a \mapsto b$ then $w(a) > w(b)$, by C1 and C2. Let r be an arbitrary site and b an instance with annotation $\alpha(b) \leq f_r^n(k)$. We consider two cases:

1. (n,k) resides in r. If $\alpha(b) = f_r^n(k)$ the result holds trivially, since $(n,k) \rightsquigarrow^* b$. If $\alpha(b) < f_r^n(k)$, then let (m,l) be the direct descendant of (n,k) for which $f_r^n(k) = f_r^m(l) + 1$, whose existence is guaranteed by C2. By inductive hypothesis $(m,l) \rightsquigarrow^* b$, and then (n,k) can also reach b.
2. (n,k) does not reside in r. Let (m,l) be an instance with $(n'k') \mapsto (m,l)$ for some $(n',k') \in [(n,k)]$, and $f_r^n(k) = f_r^m(l)$. This is guaranteed by C3. The weight of (m,l) must be strictly smaller than that of (n,k), since all elements in the class of (n,k) have at most same weight as (n,k), and following an edge $\mapsto$ the weight decreases. Therefore, the inductive hypothesis ensures that (m,l) can reach b, so $(n,k) \rightsquigarrow (n',k') \mapsto (m,l) \rightsquigarrow^* b$ can also reach b.

□