DEADLOCK AVOIDANCE FOR DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

César Sánchez
May 2007

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Zohar Manna) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Hector García-Molina)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Christopher D. Gill)
Washington University, St. Louis, MO

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Henny B. Sipma)

Approved for the University Committee on Graduate Studies.

Abstract

This thesis studies how to prevent deadlocks in distributed real-time and embedded systems. Deadlocks are undesirable states of concurrent systems, characterized by a set of processes in a circular wait state, in which each process is blocked trying to gain access to a resource held by the next one in the chain. Solutions can be classified into three categories:

- Deadlock detection is an optimistic approach. It assumes that deadlocks are infrequent and detects and corrects them at runtime. This technique is not applicable to real-time systems since the worst case running time is long. Moreover, in embedded systems actions cannot be undone.
- Deadlock prevention is a pessimistic approach. The possibility of a deadlock is broken statically at the price of restricting concurrency.
- Deadlock avoidance takes a middle route. At runtime each allocation request is examined to ensure that it cannot lead to a deadlock.

Deadlock prevention is commonly used in distributed real-time and embedded systems but, since concurrency is severely limited, an efficient distributed deadlock avoidance schema can have a big practical impact. However, it is commonly accepted (since the mid 1990s) that the general solution for distributed deadlock avoidance is impractical, because it requires maintaining global states and global atomicity of actions. The communication costs involved simply outweigh the benefits gained from avoidance over prevention.

This thesis presents an efficient distributed deadlock avoidance schema that requires no communication between the participant sites, based on a combination of static analysis and runtime protocols. This solution assumes that the possible sequences of remote calls are available for analysis at design time. This thesis also includes protocols that guarantee liveness, and mechanisms to handle distributed priority inversions.

Acknowledgements

First, I thank Zohar Manna for being my Ph.D. advisor. He has always been available whenever I needed his help. It is not a coincidence that under his guidance, many world renowned researchers have matured. I am really proud to become a member of the selective group of Zohar's advised students.

I am especially thankful to Henny Sipma for her support during all these years. We have worked together in many research endeavors, and she has always been ready for a research discussion or a personal advice. She has constantly been a careful listener, pointing out the important aspects of the situation at hand.

I would like to thank Hector García-Molina for being a member of my oral and reading committee in a short notice, and to David Dill and Marc Pauly for serving in my oral committee. Many thanks to Chris Gill for all the discussions about interesting research directions in distributed real-time and embedded systems' research. During these meetings we identified some of the initial problems whose solutions are contained in this dissertation. Thanks also for serving in my reading committee.

Many thanks to the members of the REACT research group: Aaron Bradley, Michael Colón, Bernd Finkbeiner, Sriram Sankaranarayanan, Matteo Slanina, Calogero Zarba, and Ting Zhang for all the lively discussions about research problems. Special thanks to Bernd, Matteo, Henny and Sriram for all the coffee and informal talk. Congratulations to Matteo for surviving the hard adventure of being my officemate. Thanks to all the wonderful friends that I have met during my years at Stanford. It has been a pleasure to share this time with a very diverse crowd of acquaintances, now friends for life. Many thanks to Maggie McLoughlin and Wendy Cardamone for their administrative support.

I would like to thank my first mentor, Ángel Álvarez, for creating the opportunity for me (and many others) to expand our horizons.

Last, but not least, thanks to all of those who, unconditionally, always believed in me,

when not even I did myself: my father José Antonio, my mother Adela, my brothers Marcos, Enrique and Alejandro. Also, thanks to my children Martín and Emma for understanding that I devoted to my research many moments that naturally belonged to them. Finally, many, many thanks to my wife Teresa. Her encouragement and patience has made this work possible. Teresa, this thesis is dedicated to you, with all my love.

Financial Acknowledgements

This research contained in this thesis 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, by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014, and by NAVY/ONR contract N00014-03-1-0939.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation studies the deadlock problem in the context of distributed real-time and embedded (DRE) systems, and in particular how to design systems that are deadlock-free by construction while optimizing the concurrency and utilization of resources. This first chapter includes an overview of the contributions contained in the rest of the dissertation, and discusses alternative approaches and related work.

1.1 Overview

Embedded systems are reactive systems that maintain an ongoing interaction with their physical environment, and are required to function in a wide range of scenarios (from low energy availability to a diversity of physical conditions). The software deployed in these systems cannot receive user assistance for correct functioning. In *real-time* systems, correctness is defined not only in terms of a functional specification but also in terms of a timing specification: not only must the response be the correct one, but it must be produced within a specific time interval. In *distributed* systems computations are performed in parallel by several processing units that are placed in different locations and are connected through a communication network. Distributed real-time and embedded systems are (often large) computational infrastructures used to control a variety of artifacts across a number of sites, with typical applications ranging from avionics to automotive control systems.

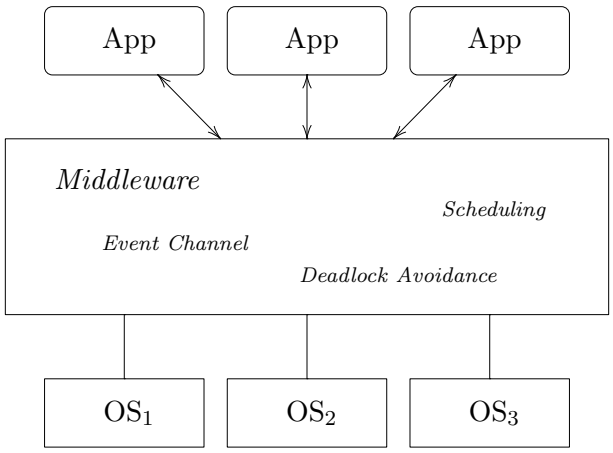


Figure 1.1: Middleware for DRE systems

1.1.1 Middleware for DRE Systems

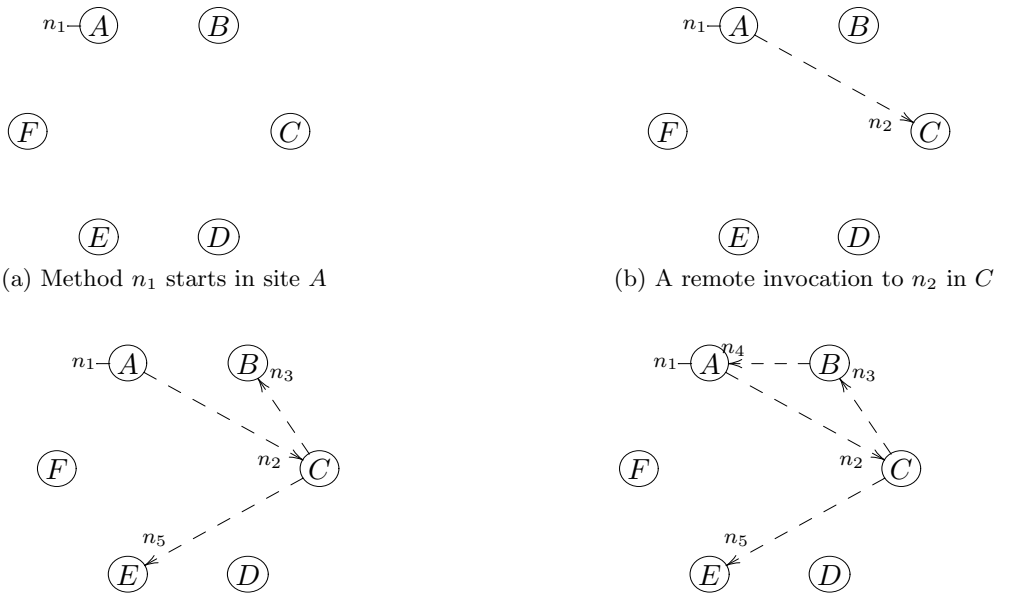
Modern distributed real-time and embedded systems are built using a common layer of software, called *middleware*, which serves two purposes. The first goal is to ease the development of applications by abstracting away the particular details of the hardware and operating system that executes in each computational site. The second purpose is to provide a family of services that are common to many applications, simplifying component design and increasing reusability while allowing specific optimizations for a particular deployment. This approach, though, can only be successful if the middleware services have clean semantics that make their behavior predictable. Moreover, rich interfaces between services and client components enable the validation of designs based on the combination of independent validations of components and services. Figure 1.1 depicts the middleware approach to DRE systems.

One example of a middleware service is event correlation. Many middleware platforms provide an event channel that supports the Publish/Subscribe Architecture. Application components publish events that may be of interest to other components, by sending them to the event channel. Components can also subscribe to the event channel to express interest in receiving certain events. There are differences, however, in what kind of subscriptions are supported. Most platforms, including Gryphon [ASS+99], Ace-Tao [SLH97], Siena [CRW01], and Elvin [SA97], support simple "event filtering": components can subscribe with a list of event types and the event channel notifies the component each time an

event of one of those types is published. A slightly more expressive mechanism is "event content filtering", in which components in their subscriptions can specify predicates over the data included in the event. Notification, however, is still based on the properties of single events. A more sophisticated subscription mechanism is "event correlation". It allows subscriptions in the form of temporal patterns. A component is notified only when a sequence of events has been published that satisfies one of the patterns. An implementation of this mechanism must maintain state: it may have to remember events it observed and may even have to store events that may have to be delivered to a component at a later stage. Event correlation is attractive because it separates the interaction logic from the application code and reduces the number of unnecessary notifications. In [SSS+03, SSSM05a, SSSM05b] we introduced a formal model of event correlation as a middleware service and proved its correctness. Prototype implementations were integrated in the state-of-the art middlewares ACE/TAO [Insty] and Facet [HCG01].

This thesis focuses on another middleware service: deadlock-free resource allocation. Ensuring efficient and correct concurrent resource allocation in DRE systems is an important and compelling problem. The resource that we focus on in this dissertation is threads needed to dispatch requests to execute methods. Ideally, the utilization of resources should be optimized so, in principle, requests should be granted whenever possible. However, whenever there are concurrent processes competing for a finite set of resources, there is a potential risk of reaching a deadlock. A deadlock is an undesirable state of concurrent systems in which a set of processes is blocked indefinitely, even though the resources present may be potentially sufficient for the participating processes to progress and eventually complete. In current practice, middleware architectures are realized in software frameworks designed according to documented best practices. These guidelines, however, do not guarantee absence of deadlock [SG04]. A more formal basis is needed to increase the reliability of DRE systems in this respect.

This thesis studies a formal model of resource allocation in distributed systems for resources that are requested and released in a nested manner: an acquired resource is not released until all resources acquired after it have been released (like in call sequences). The main motivation for this model is thread allocation, in which processes running in one processor can make two-way method calls to execute code that runs in other processors, including "nested upcalls". A nested upcall is produced when a process executing method n in processor A performs a remote invocation to another method that runs in processor



(a) Method n_1 starts in site A

(b) A remote invocation to n_2 in C

(c) The execution of n_2 performs two remote invocations: n_5 in E and n_3 in B

(d) The nested upcall n_4 in site A is invoked

Figure 1.2: A sequence of remote calls with a nested upcall

B, which in turn results in one or more method calls (possibly through other invocations transitively) in A before n returns. Nested upcalls can occur in the context of a variety of middleware concurrency architectures, including the Leader/Followers [SSRB00] approach used in TAO [Sch98, Insty] and the Half-Sync/Half-Async [SSRB00] approach used in one variant of nORB [Centy]. Figure 1.2 shows an example of a nested upcall, in a system with six sites: $\{A, B, C, D, E, F\}$. In (a), a process is started to run method n_1 in A. During the execution of n_1 a remote invocation of method n_2 in C is produced, depicted in (b). Figure (c) shows how, in turn, the execution of n_2 performs two more remote invocations of methods n_5 and n_3. Finally, a nested upcall to method n_4 is produced during the execution of n_3, shown in (d).

There are two commonly used alternative strategies to deal with nested upcalls. The first is known as *WaitOnConnection*, where the process holds on to the thread used to execute the caller method. With this strategy any nested upcalls to methods will have to acquire a new thread to run. The second approach relies on the use of a *reactor*, a technique for multiplexing asynchronously arriving events onto one or more threads [SSRB00]. This second approach is known as *WaitOnReactor*, in which a process releases the thread

after the method call is made. To preserve the semantics of the two-way call, the reactor maintains a stack and keeps track of the order in which methods were invoked, such that they are exited in reverse order. Both approaches have advantages and disadvantages. A disadvantage of *WaitOnConnection* is that threads cannot be reused while the process is waiting for the remote invocation to return, which can lead to a deadlock. A disadvantage of *WaitOnReactor* is that the stack must be unwound in *last-in-first-out* order, resulting in blocking delays for the completion of methods initiated earlier, which can lead to deadline violations. This may be especially problematic in systems with multiple agents, where this strategy can lead to unbounded delays.

This thesis focuses on mechanisms for deadlock-free thread allocation in distributed real-time and embedded systems that use the *WaitOnConnection* allocation policy.

1.1.2 Deadlock

The phenomenon of deadlock has been studied extensively in the context of computer operating systems [SGG03, Sta98]. It is well known [CES71] that there are four necessary conditions (sometimes known as Coffman's conditions) for a system to reach to a deadlock:

1. *mutual exclusion*: processes require the exclusive use of a resource, that is, every resource is either assigned to a single process or available.

2. *hold while waiting*: processes hold on to resources while waiting for additional required resources to become available.

3. *no preemption*: only the process holding a resources may determine when it is released.

4. *circular wait*: there is a closed chain of processes in which each process is waiting for a resource held by the next process in the chain.

Traditionally three techniques are used to deal with deadlocks:

- *Deadlock detection* is an optimistic method for concurrency control [Pap86, GMUW01], where deadlocks are detected at runtime and corrected by, for example, the roll-back of transactions. Hence, one of the first three conditions is broken. This approach is common in databases, but in embedded systems it is usually not applicable, especially in systems interacting with physical devices. In particular, in DRE systems that use the *WaitOnConnection* policy (1) threads can only be allocated to one process at

a time, (2) a thread is held until the call is finished, which may require subsequent threads in the same and other processing sites, and (3) a thread being held by one process cannot be released in order to be used by another process.

- *Deadlock prevention*, also known as monotone-locking [Hav68, Bir89], breaks the fourth condition statically. First, a total order on the resources is fixed. At runtime, this order (which is not necessarily the order of use) is followed to acquire the resources. Thus, a process that needs a resource n and *may* require in the future the use of a resource m with a lower index in the order, must acquire m before n. This strategy may be used in DRE systems if a high assurance that deadlocks will not be reached is required. However, this technique imposes some burden on the programmer, and—often a more important concern—it substantially reduces performance, by artificially limiting concurrency.

- *Deadlock avoidance* takes a middle route. At runtime there is a controller that keeps the system in a "safe" state where the circular chain of resource contention that produces the deadlock does not occur. The controller tries to maximize concurrency by granting requests as long as the system is guaranteed to remain in a safe state. A classic deadlock avoidance algorithm for centralized systems is Dijkstra's Banker's algorithm [Dij65], which initiated much follow-up research [Hab69, Hol72, ASK71] 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 potentially demanded by each process) and dynamic knowledge (the resource availability) to make its dynamic decisions about resource allocation. For distributed systems, on the other hand, it was observed in [SS94, Sin89] that a *general* solution to distributed deadlock avoidance is impractical, since it requires global atomic actions, or distributed synchronization. A distributed deadlock prevention algorithm usually outperforms a deadlock avoidance solution based on global synchronization running the Banker's algorithm on top.

Even though an efficient deadlock avoidance algorithm could have a big impact on the system's performance and analyzability, distributed real-time and embedded systems usually implement a deadlock prevention algorithm. This thesis studies an efficient deadlock avoidance mechanism for distributed real-time and embedded systems. This solution is

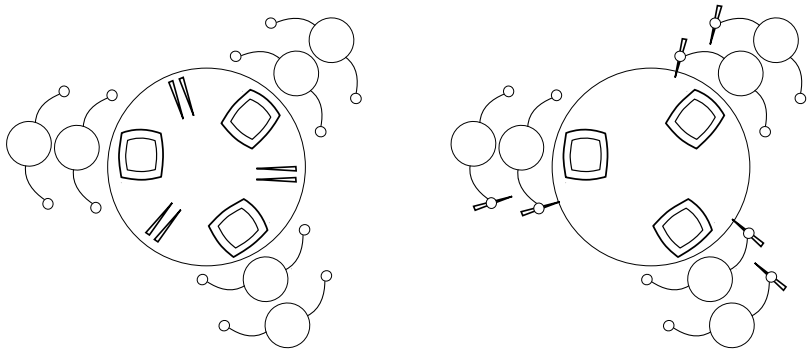


Figure 1.3: The dynamic dining philosophers, and a deadlock state

enabled by requiring more static information about the processes than in the Banker's algorithm.

1.1.3 The Dynamic Dining Philosophers Problem

We illustrate the class of deadlock scenarios that we deal with in this dissertation with the introduction of the *dynamic dining philosophers* problem, a variation of the classical dining philosophers [Dij71, LR81, CM84, AS90]. In this new setting several philosophers can arrive at any point in time to a table in order to have dinner. When a new philosopher arrives, he chooses a place to sit down from a predetermined set of places around the table. If there are other philosophers already eating in the same spot, all of them will share the food.

To aid the philosophers, the table contains a collection of chopsticks, organized in piles around the table in between every two consecutive eating spots. The chopsticks in each pile are shared among the philosophers eating at the two adjacent places. In order to eat their food, each philosopher needs to take two chopsticks, one from his left and one from his right. For example, Figure 1.3 (left) shows a scenario with three spots and two philosophers waiting to eat in each spot. In this example, the three piles contain two chopsticks.

In order to eat their desired dinner the philosophers behave as follows. Each philosopher tries to grab the left and right chopsticks sequentially, in no predetermined order. Philosophers are stubborn, so once a philosopher gets a chopstick he will not return it until he gets the other and completes his dinner. After finishing he returns both chopsticks and leaves the table. A deadlock situation can be reached if several philosophers arrive to each place (as many as chopsticks are in, for example, the right pile), and each grabs a chopstick from his right. At this point, there are no remaining chopsticks on the table, no philosopher will

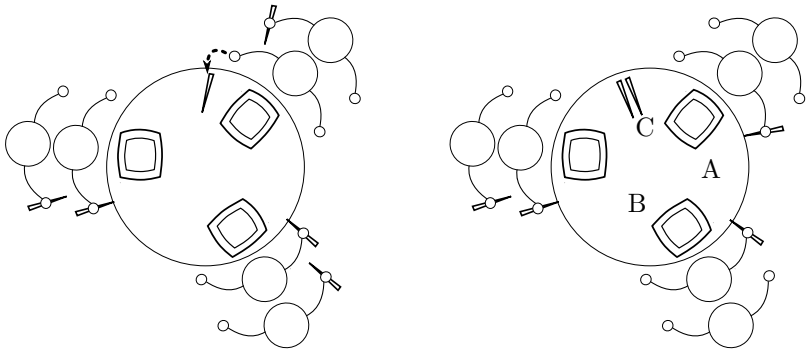


Figure 1.4: Dynamic dining philosophers, using deadlock detection and prevention

return a chopstick, and no philosopher will get his dinner. This situation is depicted in Figure 1.3 (right).

The dynamic version of dining philosophers extends the classical setting with the possibility of multiple philosophers arriving and leaving dynamically, and multiple resources (chopsticks). In the analogy to DRE systems, the philosophers represent processes, the chopstick piles are the computation nodes, and each individual process represents a resource.

Figure 1.4 (left) depicts a deadlock detection and recovery solution, where some mechanism detects the deadlock situation, and one philosopher is required to return a chopstick to the table. Figure 1.4 (right) shows a solution based on deadlock prevention where each philosopher is instructed to get the chopsticks in some predetermined order: for example, all philosophers must get the right chopstick first, except for one particular spot at the table in which all philosophers get the left chopstick first. Based on the labeling of places shown in Figure 1.4 (right), philosophers pick chopsticks from piles in the order of increasing alphabetical order.

Finally, Figure 1.5 shows the deadlock avoidance solution. On the left, in the *centralized* version, every philosopher can inspect the global situation (how many philosophers are present in each place together with the chopsticks they hold) before deciding to grab a new chopstick. A safe state is one in which all philosophers can be ordered in a list such that each philosopher can get his resources and finish his dinner, assuming that all philosophers higher in the list have returned their resources. Clearly, a safe state is not a deadlock. In this centralized solution, a philosopher can check whether such a list exists in case he picks a chopstick, and only then take it. This case is an instance of the Banker's

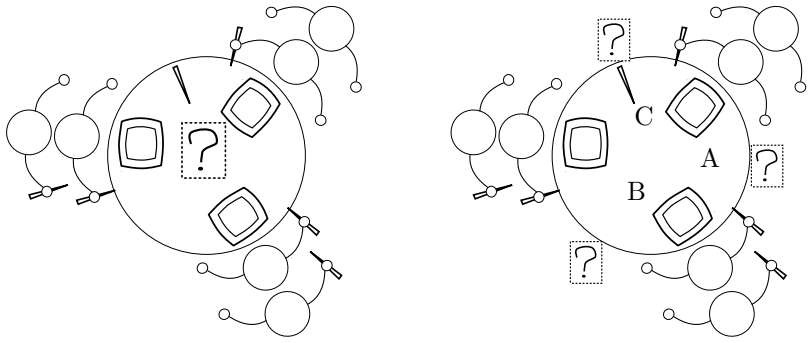


Figure 1.5: Solution to the dynamic dining philosophers using avoidance

algorithm for this simple scenario. However, our main interest is in a *distributed* algorithm, where a philosopher can only see the state of the pile he intends to pick a chopstick from. This situation is depicted in Figure 1.5 (right). One distributed solution for the dynamic dining philosophers problem works as follows. First, the piles are ordered, for example as in Figure 1.5 (right). Then, each philosophers can decide to pick his two chopsticks in increasing or decreasing alphabetical order. A philosopher that follows an increasing alphabetical order can proceed as he wishes with no restriction. A philosopher that follows a decreasing alphabetical order makes sure that he does not exhaust the pile when picking his first chopstick. In this solution, only the labeling of piles, which is statically determined, and the local availability is inspected. In this dissertation we prove that this solution is correct, and generalize it to arbitrary control flow and resource availability scenarios.

1.2 Contributions

The main contribution of this thesis is to show that efficient distributed deadlock avoidance for DRE systems is possible if the set of remote invocation sequences is known statically, before deployment. An "efficient" solution in this context is a resource allocation mechanism that optimizes concurrency (granting as many requests as possible) while being implemented in a completely distributed fashion: in order to decide whether an allocation is safe, only operations over local data are performed, and no messages are exchanged. Where the Banker's algorithm showed that deadlock avoidance in centralized systems is possible assuming only that each process announces its maximum resource utilization, the solution presented in this thesis illustrates how distributed deadlock avoidance is possible if the sequences of resource

allocations are known statically. In other words, the algorithms presented in this dissertation display a trade-off between static knowledge and efficient solvability of the distributed deadlock avoidance problem.

These solutions are realized by a combination of static analysis and runtime protocols. The static analysis extracts from the components' code the exact (or an over-approximation of the) set of possible sequences of remote invocations that each method can initiate. This information is used to compute annotations—one number for each of the methods executable in the system— which are agreed to by all processing sites before the system starts running. The runtime protocols implement the resource allocation policy, and decide dynamically—depending on resource availability—whether to grant each request.

A second contribution of this thesis is that individual liveness—every process eventually progresses—can also be guaranteed using a more sophisticated runtime protocol. The drawback is that each operation (allocation or deallocation) is no longer implementable in constant time, but depends logarithmically on the size of the system description and the size of the resource pool.

The third contribution consists of a distributed priority inheritance algorithm based on the aforementioned deadlock avoidance protocols. All solutions previously known to deal with distributed priority inversionseither severely limit the sequences of resource allocations, or use distributed versions of the priority ceiling protocol, which is computationally expensive. Our solution uses a more efficient priority inheritance mechanism. This solution is made possible by starting from an already deadlock free allocation system.

1.3 Related Work

1.3.1 Intra-resource Deadlock

The deadlock situations mentioned above are caused by the interleaving of accesses to different resources. A second kind of deadlock can be caused by the incorrect handling of accesses to a single resource, which we call intra-resource deadlock.

An intra-resource deadlock is a state in which a set of processes is waiting to be granted permission to access a single resource, but no future action will help any of these processes to gain access. Absence of intra-resource deadlock is one of the requirements of a solution to mutual exclusion, together with exclusive access and, sometimes, lack of starvation. Intra-resource deadlocks in centralized systems are typically avoided through the use of (counting)

semaphores, based on atomic actions. This solution was first proposed by Dijkstra for the THE operating system [Dij68]:

$$\left[\begin{array}{ll} \ell_0\colon & \left[\begin{array}{c}\textbf{when } 0 < s \textbf{ do}\\ s\texttt{--}\end{array}\right]\\ \ell_1\colon & \textit{critical}\\ \ell_2\colon & s\texttt{++}\\ \ell_3\colon & \end{array}\right]$$

This description of a semaphore follows the notation that we will use later for distributed deadlock avoidance protocols, where $\{l_0, \ldots, l_3\}$ represent program locations. If the value of s is initially 1, this solution is called a binary semaphore; if $s > 1$ initially, it is called a counting semaphore. Each of the statements is executed atomically. In particular, a statement of the form

$$\left[\begin{array}{c}\textbf{when } \textit{guard} \textbf{ do}\\ \textit{action}\end{array}\right]$$

executes the *action* atomically if the *guard* is satisfied. If the guard is not satisfied, the process is blocked until the system reaches a state in which it is satisfied. The statement *critical* represents the critical section, that processes are required to execute in mutual exclusion.

In distributed systems, resource allocation without intra-resource deadlock is commonly known as the distributed mutual exclusion problem. Different solutions have been proposed depending on whether the access control to the resource is distributed among the participants or it is centrally handled by a predetermined site.

1. two classes of *distributed access control* algorithms have been proposed [SABS05]:

   - Permission-based: processes communicate with each other to decide which one will gain access next. A requesting process is granted the resource if there is unanimity [RA81, CR83, RA83] among the participants about the safety of the access. This condition can be relaxed to majority quorums [Gif79, Tho79, Mae85], later improved to majorities in coteries [GMB85, AA91]. The message complexities range from $2(N-1)$ in the original Ricart-Agrawala algorithm [RA81], where every participant must cast a vote, to $O(log\ N)$ best case (with no failures) in the modern coterie-based approaches.

- Token-based: the system is equipped with a single token per resource, which is held by the process that accesses it. Upon release, the holding process passes the token to one of the requesting participants. A distributed dynamic data-type is maintained to select the next recipient. The most popular token-based distributed mutual exclusion algorithms are Suzuki-Kasami's [SK85], which exhibits a complexity of $O(N)$ messages per access, and Raymond's [Ray89] and Naimi-Tehel's approach [NTA96], which use spanning trees to obtain an average of $O(log\ N)$, still exhibiting a $O(N)$ worst case.

2. In *centralized access control* [AD76] a distinguished site arbitrates all accesses to a resource, trivially reducing the communication complexity to $O(1)$ message per request. Hence, the problem of distributed mutual exclusion is reduced to its centralized version. The drawback of this approach is fault-tolerance, since the controller site becomes a single point of failure. This strategy is preferred if it is natural to couple a resource tightly to a particular processing site. After all, if the printer has an attached participant processor, the best strategy to arbitrate accesses to the printing service is to let the printer processor resolve all requests. Centralized access control is the strategy followed in this dissertation to prevent intra-resource deadlocks.

1.3.2 Priority Inversions

It is common in real-time systems to assign priorities to processes in order to achieve a higher confidence that critical tasks will be accomplished in time. A dynamic scheduling decision function then arbitrates to favor processes with higher priority. A priority inversion is produced when a high priority process is blocked by a lower priority one. State-of-the-art DRE middleware solutions, based for example on CORBA ORBs, may suffer priority inversions [SMFGG98]. This thesis shows how priority inversions can be handled efficiently, when using a distributed deadlock avoidance algorithm as a building block.

The conventional techniques to deal with priority inversions are the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP). These were introduced in [SRL90] as methods to bound priority inversions in hard real-time systems with shared resources and static priorities. Later, they were extended to dynamic priorities (for example earliest deadline first) in [Spu95] (see [But05] for a wider discussion). PIP can bound blocking times if processes are periodic, and a bound on the running time of each process and all its critical

sections is known. Since PIP does not prevent deadlocks, PCP was introduced to also deal with deadlocks, at the cost of reducing concurrency further.

In [Mue99] Müller presents a solution to both distributed priority inversion and deadlocks, based on a distributed implementation of the priority ceiling protocol together with a token-based algorithm for distributed mutual exclusion. While PCP deals with priority inversions and guarantees that no inter-resource deadlock can occur, its implementation involves a high message complexity overhead. Before an allocation is granted, the ceiling of each resource that is globally in use must be queried; this includes asking whether a resource is assigned or free. Even though this solution is more general than the solution presented in this dissertation (in [Mue99] it is not assumed that call graphs are known), we benefit from using a deadlock-avoidance algorithm—that guarantees the absence of deadlocks—to then deal with priority inversions using the simpler and more efficient priority inheritance protocol (PIP). Our priority inheritance protocol allows more concurrency and involves *no communication when priority inversions are not present*, which is locally testable. Moreover, when inversions do exist, the priority inheritance procedure requires only one-way communication, and does not need any return information to proceed.

1.3.3 Flexible Manufacturing Systems

Resource allocation has received a lot of attention recently in the control community, especially in the context of automated production systems (see, for example, [Rev05]), with the popularization of flexible manufacturing systems (FMS). FMS are fabrication plants where a variety of different products can be manufactured sharing a common collection of basic processing units. Deadlock potentially can be reached because each processing unit has fixed sized input and output buffers. In the first FMS plants, deadlocks were resolved by a human operator, who had to release items from buffers and reset some units. A second generation of solutions employed ad-hoc or simplistic methods to prevent deadlocks. Modern solutions are based on the specification of fabrication paths, typically as Petri Nets, and the static generation of a scheduler that tries to optimize utilization while preventing deadlock. The main interest then is to study the complexity of computing optimal schedulers, and to develop efficient methods to generate safe approximations.

While our approach can be considered a refinement of the Banker's algorithm that exploits the availability of resource paths, there are two differences between FMS and the problems studied in this thesis. First, the systems considered in FMS are centralized.

All processing units have, at all times, an accurate view of the system's global state. As mentioned earlier, adapting centralized approaches to asynchronous distributed systems is usually impractical. The second difference is the nature of the resource handled. In manufacturing plants resources are slots in the queue of a fabrication unit, and consequently they are acquired and released in a sequential or "chained" manner. In this dissertation we focus on resources acquired in a nested fashion. It remains as future research to assess the applicability of the techniques developed here to distributed flexible manufacturing systems.

1.4 Structure of the Dissertation

The rest of this document is structured as follows. Chapter 2 describes the formal model of computation, and introduces the properties (deadlock, liveness, etc) that will be studied later. We introduced this general model in [SSS+05], and it is the basis for all further developments. Chapter 3 contains the most basic protocols that guarantee deadlock avoidance. These protocols depend on annotations of the call graphs, which must be computed statically. Properties of these annotations, in particular how to compute them efficiently, is the subject of Chapter 4. The basic protocols first appeared in [SSS+05], and different aspects of the annotations where explored in [SSM+06] and [SSM07b]. Chapter 5 shows that individual liveness can be achieved by a more sophisticated protocol, and presents the trade-offs in implementing this new protocol, which was studied in [SSMG06]. Chapters 6 and 7 introduce two applications. The former contains results on how new protocols can be shown correct without having to build new proofs from scratch [SSM07a], while in the latter we present a solution to priority inversions in distributed systems, published in [SSGM06]. Finally, Chapter 8 presents the conclusions.

Chapter 2

Model of Computation

This chapter introduces the model of computation that will be used in the rest of the dissertation. This is a very general model of resource allocation for DRE systems that comprises many different classes of resource types and call semantics. We start by defining an abstract notion of a distributed system, and presenting the schema of a runtime protocol. Then, we derive the notion of a state transition system that captures the dynamic behavior of a distributed system, and introduce the properties, including absence of deadlock, that will be studied later. Several protocols will be designed in the next chapters to enforce these properties. All these protocols are instances of the protocol schema. Finally, the chapter introduces allocation sequences, used to characterize the class of protocols under consideration. Allocation sequences will be used later to derive correctness results.

2.1 System Model

A distributed system consists of a set of computational sites connected via an asynchronous network. Each site is capable of executing some predefined methods, controls a fixed set of resources, and maintains a collection of local variables. A method is an implementation, in a programming language like C or Java, of the steps to be taken to accomplish some useful task. One important instruction in these methods is the remote invocation to another method in a distant site. At runtime, an instance of an execution of a method is called a process. We will use *resource* or *thread* interchangeably to refer to the execution context that is needed to run a method. Since the distributed system under consideration is asynchronous, no site has access to the variables stored in other sites.

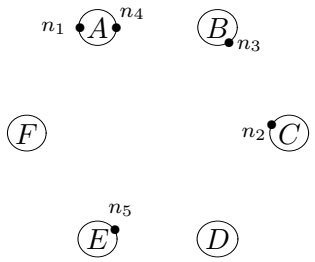


Figure 2.1: A system description

During the execution of a system, several processes can be spawned dynamically, so multiple processes may be running concurrently. Each of these processes will need a local resource to execute its method. Since for every scenario, the number of methods are finite (given by the program to be deployed), there is a finite set of kinds of processes, which we will capture by a call graph. A call graph abstracts the program by considering only the methods and their possible remote invocations: paths in the call graph represents all the possible sequences of remote invocations that a process can perform. In practice, this information can be present explicitly in the specification or extracted from the implementation by static program analysis: a simple analysis consists on listing the remote invocations that appear in the code of a given method and add the corresponding edges.

We now formalize this notion of a system.

Definition 2.1.1 (System Description). *A system description consists of a finite set $\mathcal{R}$ of computational* sites *and a set of* methods *M. Each method m is assigned to a unique site A that can execute it; we say that m resides in A, and write $m{:}A$.*

We use $site(m)$ to denote the site where method m resides. Also, $m \equiv_{\mathcal{R}} n$ is used to represent that methods m and n reside in the same site. We will use lower case letters n, m, n_1, n_2 to range over method names. Figure 2.1 shows an example of a system description with six sites $\mathcal{R} : \{A, B, C, D, E, F\}$ and five methods $M : \{n_1, n_2, n_3, n_4, n_5\}$. Methods n_1 and n_4 reside in site A, while methods n_2, n_3 and n_5 reside in C, B and E respectively.

At runtime, a process will start its execution by running some method. The methods that can be invoked by newly created processes are called *initial*. During the execution of a method, a remote invocation to another method that resides in a different site may be required. Each new method invoked by a process requires a new resource in the site where the method resides. For ease of exposition we assume that the only method invocations are

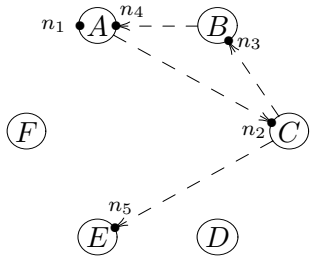


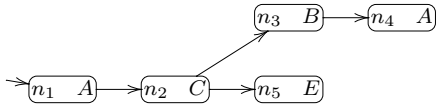
Figure 2.2: A sequence of remote invocations

remote calls. From the point of view of resource allocation, invocations to local methods can be run in the same thread and implemented as conventional function calls. A call graph captures the relations between methods and their remote invocations.

Definition 2.1.2 (Call Graph). *Given a system description with sites $\mathcal{R}$ and methods M a call graph $(N, \rightarrow, n)$ consists of a graph $(N, \rightarrow)$, where N is a subset of methods from M. The method $n \in N$ is initial. An edge $m_1 : A \rightarrow m_2 : B$ denotes that method m_1, in the course of its execution in A, may invoke method m_2 in remote site B.*

If the graph $(N, \rightarrow)$ does not contain any cycle we say that the call graph is acyclic. A nested upcall is a path that ends in a method residing in a site A, such that some other method in the path also resides in A.

Example 2.1.3. Consider the sequence of remote invocations depicted in Figure 2.2. It corresponds to the following call graph



where $n_1 : A$ is the only initial method, represented by  . This call graph is acyclic, and it contains a nested upcall to method $n_4 : A$. ⌟

Definition 2.1.4 (System). *Given a system description with sites $\mathcal{R}$ and methods M, we define a DRE system $\mathcal{S}$ as a tuple $\langle \mathcal{R}, M, \mathcal{G} \rangle$ where $\mathcal{G} : \{G_1, \ldots, G_{|\mathcal{G}|}\}$ is set of distinct acyclic call graphs.*

Formally, the requirement of acyclic call graphs to be distinct is given by $N_i \cap N_j = \varnothing$ given call graphs $G_i : (N_i, \rightarrow_i, n_i)$. This assumptions simplifies the theoretical treatment and

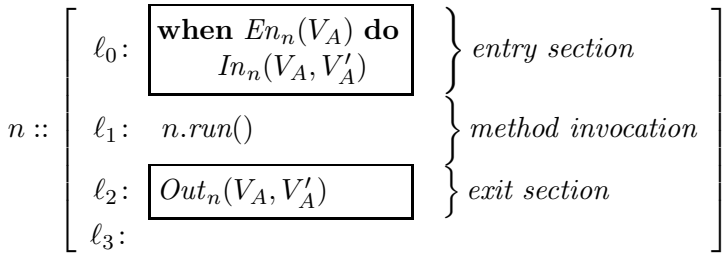


Figure 2.3: Protocol schema

does not imply a restriction in the systems modeled, since each method can be (conceptually) replicated for each call graph in which it appears. Also, without loss of generality we assume that every method listed in the system appears in some call graph (formally, $\cup_i N_i = M$) and that every method is reachable from an initial method. Otherwise, a system that accepts the same runs and has all methods reachable can be obtained by removing non-reachable methods. Moreover, every acyclic graph can be turned into a tree by replicating methods if necessary. In this way, $\mathcal{G}$ is equivalent to a forest (collection of trees) in which each method occurs in exactly one tree. We will sometimes merge all the call graphs into a *global call graph*, defined as $(M, \rightarrow, I)$, where $I = \cup_i n_i$ is the set of all possible initial methods.

Each site A manages a fixed number of pre-allocated resources (threads). Although in many modern operating systems threads can be spawned dynamically, many DRE systems pre-allocate fixed sets of threads to avoid the relatively large and variable cost of thread creation and initialization. We assume that each site A maintains a set of local variables V_A, called *protocol variables* that will be used for controlling the resource allocation. V_A includes the constant $T_A \geq 1$ that denotes the total number of resources present in A, and a variable t_A whose value represents the number of available threads (threads not being used). Initially, $t_A = T_A$ for every site A. The allocation of resources is controlled by a piece of code that runs in the site before and after the methods are dispatched. This notion of resource allocation manager is captured by runtime protocols.

2.2 Runtime Protocols

A runtime *protocol* for controlling the resource allocation in site A to execute method $n : A$ is implemented by a program, containing sections of code executed before and after method n is dispatched, that are analogous to the operations **P** and **V** in conventional

$$n :: \left[\begin{array}{ll} \ell_0\colon & \left[\begin{array}{c}\textbf{when } \mathit{true} \textbf{ do}\\ \textbf{skip}\end{array}\right] \\ \ell_1\colon & n.\mathit{run}() \\ \ell_2\colon & \textbf{skip} \\ \ell_3\colon & \end{array}\right]$$

Figure 2.4: The protocol EMPTY-P

semaphores [Dij68]. In practice, this code can be different for different call graph methods even if they reside in the same site (typically, the code is the same but parametrized by the call graph method). The schematic structure of a protocol that controls the execution of a method $n : A$ appears in Figure 2.3. Upon invocation, the entry section checks resource availability by inspecting local variables V_A of site A. If the predicate $En_n(V_A)$, called the enabling condition, is satisfied we say that the entry section is *enabled*. In this case, the request can be granted and the local variables updated according to the logical relation $In_n(V_A, V'_A)$, where V'_A stands for the values of the protocol variables after the action is taken. In case the enabling condition is not satisfied, the process must wait. We assume that the entry section is executed atomically, as a *test-and-set*. The method invocation section executes the code of the method, which may perform remote calls according to the outgoing edges from method n in the call graph. The method invocation can only terminate after all its own invoked calls have terminated and returned. The exit section releases the resource and may update some local variables in site A, according to the relation $Out_n(V_A, V'_A)$. In_n is called the *entry action* and Out_n is called the *exit action* of the protocol. Note that this call semantics are more general than synchronous semantics, since the execution can continue after a remote invocation, for example to perform more remote invocations. We assume, though, that all results of all calls must be collected before the caller terminates.

Example 2.2.1. Figure 2.4 illustrates the simplest possible protocol, EMPTY-P, where the code is shown for method $n : A$. The enabling condition is always satisfied, and no variable is modified in the entry or exit section (represented by the void statement **skip**). Note that the protocol EMPTY-P does not restrict the number of allocations to be less than the number of resources. This property is studied in Section 2.4. ⌟

This model of a protocol captures our goal of designing protocols that decide resource allocations without exchanging any messages between remote sites. We call these protocols *local protocols*.

2.3 Dynamic Behavior

The dynamic behavior of a DRE system is represented by sequences of global states, where each state contains (1) the set of running processes, indicating for each of them a protocol location, and (2) the valuation of the protocol variables of every site. To model the state of a process we introduce the notion of labeled call graph.

Definition 2.3.1 (Labeled Call Graph). *Let ℓ_0, ℓ_1, ℓ_2, and ℓ_3 be protocol location labels representing the progress of a process, as illustrated in Figure 2.3. A labeled call graph (G_i, γ) is an instance of a call graph $G_i : (N_i, \rightarrow_i, n_i) \in \mathcal{G}$ and a labeling function $\gamma : N_i \mapsto \{\perp, \ell_0, \ell_1, \ell_2, \ell_3\}$ that maps each method in the call graph to a protocol location, or to $\perp$ for method calls that have not been performed yet.*

The state of a process is modeled formally as a labeled call graph. A subtree of a labeled call graph corresponds to the state of a subprocess. A subprocess is an execution started in a remote site to run a method serving a remote invocation. We will use the term "process" to refer to either a process or a subprocess, and use "proper process" and "subprocess" explicitly only when the distinction is relevant. A process is *active* if its root method is labeled ℓ_1 or ℓ_2, that is, if the process has been granted a resource. A process is *waiting* if it is labeled ℓ_0, and *terminated* if it is labeled ℓ_3. A terminated process has already returned its assigned resource. We use upper-case letters $P, Q, P_1, \ldots$ to range over processes. To simplify the presentation, given a process state $P = (G, \gamma)$ we use $\gamma(P)$ as an alias of $\gamma(root(P))$. We also say that process P is in location ℓ if $\gamma(P) = \ell$.

Definition 2.3.2 (Global State). *A global state of a DRE system is a pair $\langle \mathcal{P}, s_{\mathcal{R}} \rangle$ consisting of a finite indexed set of processes $\mathcal{P}$ together with their local states, and a valuation $s_{\mathcal{R}}$ of the local variables in all sites.*

For example, the initial global state of a system consists of an empty set of processes and a valuation $s_{\mathcal{R}}$ for all variables of every site, in particular $s_{\mathcal{R}}(t_A) = T_A$ to denote that all resources are initially available.

A system $\mathcal{S} : \langle \mathcal{R}, M, \mathcal{G} \rangle$ gives rise to the *state transition system* (see [MP95]) $\Psi : \langle V, \Theta, \mathcal{T} \rangle$. V is a set of variables that describe system states, Θ is a logical predicate that captures the possible initial states, and $\mathcal{T}$ is a logical relation describing the state changes that define the system dynamics. Formally,

- V : $\{I\} \cup V_{\mathcal{R}}$ is a set of variables, containing I to index the set of existing processes, and the local protocol variables $V_{\mathcal{R}} = \bigcup_{A \in \mathcal{R}} V_A$ of every site. A valuation of V is a global state, where the value of I is the set of currently running processes $\mathcal{P}$.

- Θ : $I = \varnothing \wedge \bigwedge_{A \in \mathcal{R}} \Theta_A$: the initial condition, specifying initial values for the local variables and initializing the set of processes to the empty set.

- $\mathcal{T}$: a set of state transitions consisting of the following global transitions:

  1. **Creation:** A new process P (where P is a fresh process index) is created, with initial status $\gamma(n) = \perp$ for all methods n in its call graph. The valuation of I is updated to add P. The logical relation that describes a **creation** transition is:

     $$\tau_1 : I' = I + P \wedge \mathit{pres}(V_{\mathcal{R}})$$

     where $+$ represents the addition of P to I, and the predicate $\mathit{pres}(V_{\mathcal{R}})$ states that all variables in $V_{\mathcal{R}}$ are preserved.

  2. **Process initialization:** Let $P \in I$ be an existing proper process with $\gamma(P) = \perp$. A process initialization changes the annotation of P to ℓ_0. Formally,

     $$\tau_2 : \gamma(P) = \perp \wedge \gamma'(P) = \ell_0 \wedge \mathit{pres}(V_{\mathcal{R}})$$

  3. **Method invocation:** Let P be an existing process and Q be a proper subprocess of P with $\gamma(Q) = \perp$ and $\gamma(\mathit{parent}(Q)) = \ell_1$. This corresponds to a remote invocation that creates Q. The method invocation transition changes the annotation of Q to ℓ_0:

     $$\tau_3 : \gamma(Q) = \perp \wedge \gamma(\mathit{parent}(Q)) = \ell_1 \wedge \gamma'(Q) = \ell_0 \wedge \mathit{pres}(V_{\mathcal{R}})$$

  4. **Method entry:** Let Q be a waiting process, running method $n : A$, whose enabling condition is satisfied. The method entry describes that process Q is granted its resource. Transition τ_4 changes the label of Q to ℓ_1 and updates the local variables in its site according to the protocol for $n{:}A$. Formally,

     $$\tau_4 : \gamma(Q) = \ell_0 \wedge \mathit{En}_n(V_A) \wedge \gamma'(Q) = \ell_1 \wedge \mathit{In}_n(V_A, V_A') \wedge \mathit{pres}(V_{\mathcal{R}} - V_A)$$

5. **Method execution:** Let Q be a process in ℓ_1 such that all its descendants are labeled $\perp$ or ℓ_3. This transition denotes the termination of Q. The status of Q is updated to ℓ_2:

$$\tau_5 : \ \gamma(Q) = \ell_1 \ \ \wedge \ \ \bigwedge_{D \in descs(Q)} \Big[\gamma(D) = \perp \ \vee \ \gamma(D) = \ell_3\Big] \ \wedge \ \gamma'(Q) = \ell_2 \ \wedge \\ \wedge \ \ pres(V_{\mathcal{R}})$$

6. **Method exit:** Let Q be a process in ℓ_2, running method $n : A$. The method exit transition moves Q to ℓ_3 and updates the variables in site A according to the exit action for $n : A$. Formally,

$$\tau_6 : \ \gamma(Q) = \ell_2 \ \wedge \ \gamma'(Q) = \ell_3 \ \wedge Out_n(V_A, V'_A) \wedge pres(V_{\mathcal{R}} - V_A)$$

7. **Deletion:** A proper process P that has terminated (is labeled ℓ_3) is removed from I:

$$\tau_7 : \gamma(P) = \ell_3 \ \wedge \ I' = I - P \ \wedge \ pres(V_{\mathcal{R}})$$

8. **Silent:** In the silent transition all variables are preserved:

$$\tau_8 : pres(V)$$

All transitions except **creation** and **silent** are called *progressing* transitions, since they correspond to the progress of some existing process. The system, as defined, is nondeterministic. It assumes an external environment that determines when new processes are created and selects which transitions are taken. Since a remote invocation is modeled by the **method invocation** transition τ_3, this notion of uncontrolled environment models the uncertainty in the delivery of messages natural to the asynchronous underlying network. Also, in a situation in which different processes compete for a resource, the environment decides which one proceeds to the invocation section, by scheduling its **method entry** transition. Note also that an arbitrary number of processes can be spawned dynamically, as long as each of them is assigned a process type (a labeled call graph) and a unique index.

The only restriction that we impose on the environment is that no progressing transition can be continuously enabled without being taken. Therefore, unless there are no processes running or the system is deadlocked, an infinite sequence of silent transitions cannot occur because a progressing transition must be eventually scheduled. Moreover, since all call

graphs are acyclic, this fairness condition implies that all processes will eventually terminate if granted their demanded resources.

We are now ready to define the notion of run, which captures a possible execution of the DRE system. We will later define properties that protocols must impose on all runs of a system.

Definition 2.3.3 (Run). *A* run *of a system is an infinite sequence $\sigma_0, \sigma_1, \ldots$ of global states such that σ_0 is an initial state, and for every i, there exists a transition $\tau \in \mathcal{T}$ such that σ_{i+1} results by taking τ from σ_i.*

2.4 Properties

In this section we formally define properties to study the correctness of the protocols. Most of these properties are presented as *invariants*.

Definition 2.4.1 (Invariant). *Given a system $\mathcal{S}$, an expression ψ over the system variables of $\mathcal{S}$ is an invariant if it is true in every state of every run of $\mathcal{S}$.*

An expression can be proved invariant by showing that it is inductive or implied by an inductive expression. An expression ψ is *inductive* for a transition system $\langle V, \Theta, \mathcal{T} \rangle$ if it is implied by the initial condition, $\Theta \rightarrow \psi$, and it is preserved by all its transitions, $\psi \wedge \tau \rightarrow \psi'$, for all $\tau \in \mathcal{T}$. Using invariants we now define adequacy.

Definition 2.4.2 (Adequate). *A protocol is adequate if the number of resources allocated in every site A never exceeds the total number of initially available resources T_A.*

Adequacy is a fundamental property, required in every reasonable protocol, since no more resources than available can possibly be granted. Figure 2.5 shows ADEQUATE-P, a simple adequate protocol. This protocol is a simple generalization of a counting semaphore for the variable t_A that keeps count of the available resources. The entry section of ADEQUATE-P blocks further progress until the guard expression $0 < t_A$ evaluates to true, that is, until there is at least one resource available. The adequacy of ADEQUATE-P is a consequence of the following invariants:

$$\psi_1 : \textit{ for all sites } A.\ t_A \geq 0$$
$$\psi_2 : \textit{ for all sites } A.\ T_A = t_A + at\_\ell_{1,A} + at\_\ell_{2,A}$$

$$n :: \left[\begin{array}{ll} \ell_0 : & \left[\begin{array}{c}\textbf{when } 0 < t_A \textbf{ do} \\ t_A\texttt{--}\end{array}\right] \\ \ell_1 : & n.run() \\ \ell_2 : & t_A\texttt{++} \\ \ell_3 : & \end{array}\right]$$

Figure 2.5: The protocol ADEQUATE-P

where the symbol $at\_\ell_{i,A}$ represents, for a given global state, the total number of processes running in site A that are labeled ℓ_i. In particular, $at\_\ell_{1,A} + at\_\ell_{2,A}$ denotes the total number of active processes in site A. It is easy to show that ψ_1 and ψ_2 are inductive, since they hold initially and every transition preserves them, provided that ADEQUATE-P is used as allocation protocol.

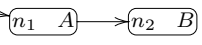
On the other hand, the protocol EMPTY-P (Figure 2.4) does not guarantee the adequacy property. One can easily build a run that violates adequacy as follows. First, pick any call graph (let the root node be $n{:}A$), and spawn $T_A{+}1$ processes to follow this call graph. Then, activate all processes by scheduling the appropriate **process initialization** and **method entry** transitions, after which more than T_A resources are in use in site A.

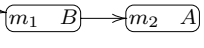
The next property that we formally define is deadlock, which is the central notion studied in this dissertation. Our goal is to build protocols that prevent any deadlock from happening.

Definition 2.4.3 (Deadlock). *A state σ is a deadlock if some process is waiting, but only non-progressing transitions are enabled.*

If a deadlock is reached, the processes involved cannot progress. Intuitively, each of the processes has locked some resources that are necessary for other processes to complete, but none of them has enough resources to terminate. The following example shows that ADEQUATE-P does not guarantee absence of deadlock.

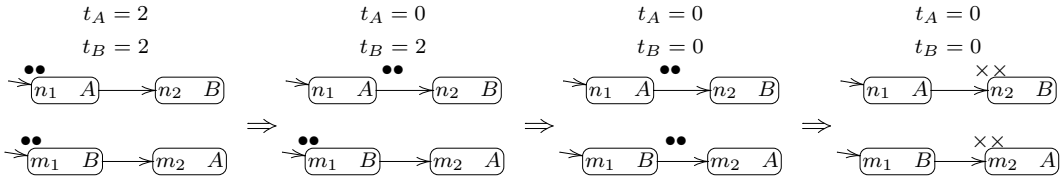
Example 2.4.4. Consider the system $\mathcal{S} : \langle \mathcal{R}, M, \{G_1, G_2\}\rangle$ with two sites $\mathcal{R} : \{A, B\}$, four methods $M = \{n_1, n_2, m_1, m_2\}$, and the following two call graphs

G_1 : 

G_2 : 

Assume that both sites have two resources initially available ($T_A = T_B = 2$), and that

ADEQUATE-P is used to control resource allocation for all methods. Let the global state $\sigma : \langle\{P_1, P_2, Q_1, Q_2\}, t_A = 0, t_B = 0\rangle$ consist of four processes: P_1 and P_2, instances of G_1 both with $\gamma(n_1) = \ell_1$ and $\gamma(n_2) = \ell_0$, and Q_1 and Q_2 instances of G_2 with $\gamma(m_1) = \ell_1$ and $\gamma(m_2) = \ell_0$. It is easy to see that σ is a deadlock: no progressing transition is enabled. Furthermore, σ is reachable from an initial state and hence appears in some run. A run from the initial state $\langle\varnothing, t_A = 2, t_B = 2\rangle$ to the deadlock state σ is depicted below:



In this diagram a • represents an existing process that tries to acquire a resource at a method (if • precedes the method) or has just been granted the resource (if • appears after the method). A × represents that the enabling condition is not satisfied so the method entry transition cannot be executed. ⌟

The following result is a fundamental lemma that holds independently of the protocol used. This lemma states that if a process P is present in a deadlock while holding a resource, then P must be blocked waiting for a remote call to return.

Lemma 2.4.5. In a deadlock state, every active process has a waiting descendant.

Proof. Let σ be a deadlock state and P an active process running method n. Since P is active, either $\gamma(P) = \ell_1$ or $\gamma(P) = \ell_2$. In the latter case, transition τ_6 is enabled, contradicting deadlock, so the labeling of P must be ℓ_1. We prove that P has at least one waiting descendant by induction on the minimum distance in the call graph from n to a leaf. For the base case, let n be a leaf. In this case the **method execution** transition τ_5 is enabled for P, contradicting deadlock. Hence, a leaf method cannot be active in a deadlock state. For the inductive case, let $Q_1, \ldots, Q_n$ be the descendants of P. If some Q_i is waiting the result follows. If some Q_i is active, by the inductive hypothesis it has a waiting descendant, and hence P also has a waiting descendant. The last case is, for all Q_i, $\gamma(Q_i) = \perp$ or $\gamma(Q_i) = \ell_3$. But then τ_5 is enabled for P, contradicting deadlock. □

Another desirable property of resource allocation protocols is absence of starvation, that is, every individual process eventually progresses.

Definition 2.4.6 (Starvation). *A process P starves in a run of a system if, after some prefix of the run, the labeling of P never changes thereafter. A system prevents starvation if no process starves in any of its runs.*

In the model of computation introduced above, every process terminates after a finite number of transitions, so if a process does not starve, it eventually terminates. Deadlock implies starvation because every process present in a deadlock state starves. The converse does not hold, as will be shown in Chapter 5.

2.5 Allocation Sequences

We finish our discussion of the model of computation by presenting allocation sequences. It is clear from the definition of the state transition system that the only transitions that modify the values of the protocol variables are **method entry** and **method exit**. We abstract here a system's run into the sequence of allocations and deallocations that are performed, and define some natural requirements for the protocols based on allocation sequences.

Given a system $\mathcal{S}$ and its global call graph $(M, \rightarrow, I)$ let the set $\overline{M}$ contain a symbol $\overline{n}$ for every method n in M. The allocation alphabet Σ is defined as the disjoint union of M and $\overline{M}$. Symbols in M are called allocation symbols, while symbols in $\overline{M}$ are referred to as deallocation symbols. Every prefix of a system run can be abstracted into an allocation sequence by considering only the **method entry** transition τ_4 and the **method exit** transition τ_6.

Given a string s in Σ^* and an allocation symbol n in Σ we use s_n for the number of occurrences of n in s (similarly $s_{\overline{n}}$ represents the number of occurrences of the deallocation symbol $\overline{n}$), and use $|s|_n$ to stand for $s_n - s_{\overline{n}}$. A *well-formed* allocation string s is one for which every deallocation occurs after a matching allocation, that is, for every prefix p of s, $|p|_n \geq 0$. An *admissible* allocation sequence is one that corresponds to a prefix run of the system. This requires (1) that the string is well-formed, (2) that every allocation of a non-root method is preceded by a matching allocation of its parent method, and (3) that every deallocation of a method is preceded by a corresponding deallocation of its children methods. Formally,

Definition 2.5.1 (Admissible Strings). *A well-formed allocation string s is called admissible if for every prefix p of s, and every remote call $n \rightarrow m$: $|p|_n \geq |p|_m$.*

Admissible strings ensure that the number of children processes (callees) is not higher than the number of parent (caller) processes, so that there is a possible match, and the string corresponds to a feasible run. For brevity, we simply use *string* to refer to an admissible string.

Given a state σ and a protocol P, if the enabling condition of P for a node n is satisfied at σ we write $En_n^P(\sigma)$. For convenience, we introduce a new state $\perp$ to capture sequences that a protocol forbids, and require that no enabling condition is satisfied in state $\perp$. We denote by $P(s)$ the[1] state reached by P after exercising the allocation string s, defined inductively as $P(\epsilon) = \Theta$ and:

$$P(s\ n) = \begin{cases} In_n^P(P(s)) & \text{if } En_n^P(P(s)) \\ \perp & \text{otherwise} \end{cases} \qquad P(s\ \overline{n}) = \begin{cases} Out_n^P(P(s)) & \text{if } P(s) \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

We say that a string s is accepted by a protocol P if $P(s) \neq \perp$. In other words, $P(s) \neq \perp$ for all strings that correspond to prefixes of runs when P is used as allocation manager. The set of strings accepted by P is denoted by $\mathcal{L}(P)$.

Example 2.5.2. Reconsider the system in Example 2.4.4. The allocation sequence that leads to a deadlock is $s : n_1 n_1 m_1 m_1$, which is in the language of ADEQUATE-P. The reached state is ADEQUATE-P$(n_1 n_1 m_1 m_1) = \langle \{P_1, P_2, Q_1, Q_2\}, t_A = 0, t_B = 0 \rangle$, a deadlock state. ⊿

Intuitively, allocations imply that the resource availability is reduced, and deallocations that it is increased. We are interested in exploring protocols that follow this intuition. We say that a protocol is *monotone* if an allocation by a process cannot help some other waiting process to gain a resource, and a deallocation cannot turn a different process from enabled into waiting. Formally,

Definition 2.5.3 (Monotone Protocol). *A protocol P is called monotone if for all strings s_1 and s_2, method m and allocation symbol n,*

1. *if $En_m^P(P(s_1 s_2))$ is false, then so is $En_m^P(s_1 n s_2)$, and*

2. *if $En_m^P(P(s_1 s_2))$ is true, then so is $En_m^P(s_1 \overline{n} s_2)$.*

[1] all the protocols studied in this dissertation are deterministic but the results can be easily adapted for non-deterministic protocols.

Another characteristic of the resources we model is that they are reusable. After a process finishes using a thread and returns it, the thread is intact and ready for another process. This is captured by the notion of a reversible protocol. Intuitively, a protocol is reversible if a deallocation undoes all the effects of its matching allocation. Formally,

Definition 2.5.4 (Reversible Protocol). *A protocol P is called reversible if for all strings s_1 and s_2, and allocation symbol m, whenever $P(s_1 m s_2 \overline{m}) \neq \perp$, then the states $P(s_1 m s_2 \overline{m})$ and $P(s_1 s_2)$ are identical.*

This includes the fact that $P(sm\overline{m})$ is equivalent to $P(s)$, whenever $P(sm\overline{m}) \neq \perp$. We will focus our attention on local, monotone and reversible protocols, and call them simply *protocols* in the rest of this thesis.

2.6 Summary

In this chapter we have introduced a new model of computation. DRE systems are modeled as a finite set of sites that are able to execute a predetermined finite collection of methods, and a set of acyclic call graphs that capture the possible sequences of remote method invocations. Runtime protocols implement the resource allocation policy, controlling each resource allocation and deallocation. We are interested in studying protocols that are local (only local variables are inspected and manipulated), monotone (allocations cannot help waiting processes, and deallocations cannot obstruct enabled requests), and reversible (the effects of allocations and deallocations cancel each other). Given a system and a protocol for every method, the dynamics are described by a state transition system, from which we derive formally the notion of a deadlock state.

The semantics of remote invocation defined here correspond to asynchronous calls that must collect results before terminating. Since synchronous calls are a special case, captured by a subset of the runs, protocols that prevent deadlocks in our model will also prevent deadlocks for synchronous call semantics. Fully asynchronous invocations without result collection can also be described, by removing the edge that corresponds to a fully asynchronous call that does not require collection. Such an invocation will be modeled by a new process being spawned to perform the callee. Other semantics of remote invocation (at-most-once, at-least-once, etc) can also be incorporated in this model easily, but these adaptations are out of the scope of this dissertation.

Chapter 3

Basic Solutions

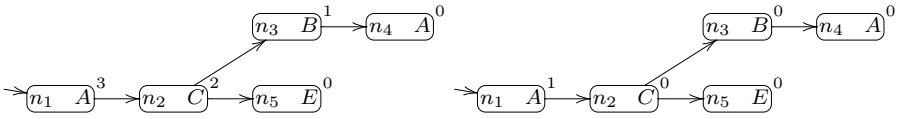
This chapter contains the first protocols that guarantee deadlock free resource allocation. These protocols are parameterized by annotations of the call graph methods. The annotation is computed before the system is deployed, and all sites agree on the same annotation. We begin by presenting some examples of annotations that ensure deadlock freedom in scenarios with no concurrency. Then, we describe a protocol that provides deadlock avoidance with arbitrary concurrency. Finally, we capture the precise property that annotations must satisfy to guarantee absence of deadlock.

3.1 An Introduction to Annotations

We first introduce the notion of annotation. Consider a system with a global call graph $(M, \rightarrow, I)$. An annotation is a map from methods to the natural numbers $\alpha : M \mapsto \mathbb{N}$. Intuitively, an annotation provides a measure of the resources that need to be available in the local site to complete the task described by the method. We now define two simple annotations: *height* and *local height*. The height of a method in a call graph is the usual height of a node in a tree.

Definition 3.1.1 (Height). *Given a call graph $(M, \rightarrow, I)$, the height of a method n, denoted by $h(n)$, is defined inductively as*

$$h(n) = \begin{cases} 0 & \textit{if } n \textit{ is a leaf} \\ 1 + max\{h(m) \mid n \rightarrow m\} & \textit{otherwise.} \end{cases}$$



(a) A height annotation (b) A local height annotation

Figure 3.1: The height and local-height annotations of a call graph

Local height is defined similarly, but only taking into account methods that reside in the same site. The local height of a method n is the maximum number of methods in the same site that are traversed in any outgoing path from n:

Definition 3.1.2 (Local Height). *Given a call graph $(M, \rightarrow, I)$, the local height of a method n, denoted by $lh(n)$ is defined inductively as*

$$lh(n) = \begin{cases} 0 & \text{if } n \text{ is a leaf, and} \\ 1 + max\{lh(m) \mid n \rightarrow^+ m \text{ and } n \equiv_{\mathcal{R}} m\} & \text{otherwise.} \end{cases}$$

where $\rightarrow^+$ is the transitive closure of $\rightarrow$, i.e., $n \rightarrow^+ m$ represents that there is a sequence of remote invocations from n to m.

Example 3.1.3. Figure 3.1 shows the call graph of Example 2.1.3 annotated with height, in Figure 3.1(a), and local-height, in Figure 3.1(b). For example, $n_1 : A$ has local height 1 because n_1 may indirectly call n_4 in the same site through a nested upcall. ⊣

The previous definitions are sound because the call graphs are acyclic. Annotations play a special role in the deadlock avoidance protocols introduced later in this chapter. Chapter 4 contains several results about annotations including effective procedures to compute optimal ones.

3.2 Single Agent

For simplicity, let us begin by considering scenarios with a single agent sequentially activating processes, first studied in [SG04]. These scenarios correspond to environments that cannot perform concurrent process spawning, and in which all remote invocations follow synchronous call semantics. In terms of the formal model introduced in the previous chapter, this implies that the number of processes in any state is at most 1, that is, $|I| \leq 1$

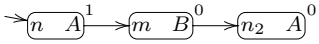
is an invariant. The following theorem establishes a necessary and sufficient condition to guarantee absence of deadlock in the single agent case:

Theorem 3.2.1 (from [SG04]). A method $n{:}A$ with local-height k can be executed with absence of deadlock, if more than k resources are available in A.

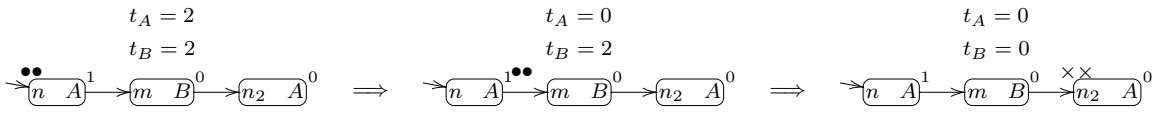
Theorem 3.2.1 provides a simple design-time technique to compute the minimum number of resources needed in each site to guarantee deadlock free operation: the value of T_A must be at least the maximum local height for any method residing in A in any call graph. The condition is necessary, because if it is violated a deadlock will occur, independently of the protocol used. If this condition is violated, a run that follows a path that contains the method with maximum local height leads to a deadlock. The condition is sufficient because if it is met no deadlock will occur. Thus, in the single agent case the protocol EMPTY-P will guarantee absence of deadlock, provided all sites manage initially at least the required number of threads. Trivially, ADEQUATE-P also guarantees absence of deadlock, while unnecessarily testing for resource availability.

In practice, this scenario faithfully represents an environment that is restricted to perform no concurrent calls. In cases where multiple processes can run in parallel, or can perform asynchronous remote invocations, the condition expressed in Theorem 3.2.1 is necessary but not sufficient to guarantee the absence of a deadlock—using EMPTY-P or ADEQUATE-P—as the following example shows.

Example 3.2.2. Consider a system with two sites, A and B, with $T_A = T_B = 2$ and the call graph



where the local-height annotation is depicted. This system satisfies the condition of Theorem 3.2.1. If two processes are spawned and then both of them gain access to n, the resources in A are exhausted. The system will arrive at a deadlock as soon as the processes try to execute n_2. These nested calls will be blocked in the entry section of n_2 forever:



This example can be scaled up to any number of resources in site A, simply by spawning the same number of processes and scheduling each of them to acquire a thread to run $n{:}A$, which is a possible prefix run. ⌟

$$n :: \left[\begin{array}{ll} \ell_0\colon & \left[\begin{array}{c}\textbf{when } \alpha(n) < t_A \textbf{ do} \\ t_A\texttt{--}\end{array}\right] \\ \ell_1\colon & n.run() \\ \ell_2\colon & t_A\texttt{++} \\ \ell_3\colon & \end{array}\right]$$

Figure 3.2: The protocol BASIC-P

3.3 A Basic Solution for Multiple Agents

Examples 3.2.2 and 2.4.4 show that neither EMPTY-P nor ADEQUATE-P guarantee absence of a deadlock in a general execution environment. Indeed, it is easy to see that in these examples no number of pre-allocated resources can make deadlock unreachable, in the presence of an unbounded number of concurrent invocations. Thus, more sophisticated protocols are needed to control resource allocation.

In this section we introduce the first and simplest deadlock avoidance protocol, called BASIC-P , which is parameterized by an annotation α, and show properties that hold for every annotation. Then, in the next sections we prove that this protocol provides deadlock avoidance for some specific annotations, including height.

The protocol BASIC-P is shown in Figure 3.2 for call graph method $n\colon A$. The variable t_A, as usual, keeps track of the threads currently available in site A. In the entry section access is granted to run method n only if the number of resources indicated by the annotation $\alpha(n)$ is strictly less than the number of resources available. When access is granted, t_A is decremented by just one unit, reflecting that only one thread has been allocated. Note that not all resources provisioned are locked. The protocol BASIC-P is local since the annotation of every method $n\colon A$ is provided statically and the only variable used is t_A. It is monotone because the comparison $<$ used in the enabling condition is monotone. It is reversible because the effects of decrement (t_A--) and increment (t_A++) cancel each other.

We now establish some properties that hold for BASIC-P, for any annotation α. We first introduce some notation and abbreviations. Recall that the symbol $at\_\ell_{i,A}$ represents the number of processes in site A that are at location ℓ_i. This way, the number of active processes in A is given by $Act_A \stackrel{\text{def}}{=} at\_\ell_{1,A} + at\_\ell_{2,A}$. We use $act_A[k]$ to denote the number of active processes running methods with annotation k, and $Act_A[k]$ for the number of active processes in A with annotation greater than or equal k. That is, $Act_A[k] = \sum_{j \geq k} act_A[j]$

and, in particular, $Act_A[0] = Act_A$. With initial condition $\Theta_A : \ T_A = t_A$, it is easy to verify that the following are inductive invariants for all sites, when BASIC-P is used as the allocation protocol:

$$\begin{aligned} \varphi_1 &: \ t_A \geq 0 \\ \varphi_2 &: \ T_A = t_A + Act_A \end{aligned}$$

The following lemmas apply to all sites A.

Lemma 3.3.1. If $t_A = 0$ then there exists at least one active process with annotation 0 in A, that is,

$$\varphi_3 : \ t_A = 0 \rightarrow act_A[0] \geq 1$$

is an invariant.

Proof. The predicate φ_3 is inductive, so it is an invariant. □

Lemma 3.3.1 states that when resources are exhausted there is at least one active process with the lowest annotation. We now prove some results that connect resource availability with guarantees on the existence of active processes running methods with a small enough annotation value.

Lemma 3.3.2. The number of active processes in A running methods with annotation k or higher is less than or equal to $T_A - k$. That is, for all sites A and annotations k

$$Act_A[k] \leq T_A - k$$

is an invariant. We call this invariant φ.

Proof. Clearly, φ holds initially. To prove that φ is an invariant, it suffices to show that in a state where $Act_A[k] = T_A - k$, a waiting process Q attempting to execute a method $m{:}A$ with annotation value $\alpha(m) \geq k$ cannot proceed. Note that, by definition, $Act_A \geq Act_A[k]$ for every site A and annotation value k. Then, by φ_2,

$$\begin{aligned} T_A &= t_A + Act_A \\ &\geq t_A + Act_A[k] \\ &= t_A + (T_A - k). \end{aligned}$$

Therefore, $k \geq t_A$. Then, the enabling condition of m is not satisfied and process Q must wait, as desired. □

Global states that satisfy the invariant φ are called φ-states. The fact that BASIC-P preserves φ is the central keystone in the proof of deadlock avoidance. This invariant will be revisited in Chapter 5 and refined into a protocol that provides liveness for individual processes. We use φ_A for the clauses that involve site A, and $\varphi_A[k]$ for the clause for k: $Act_A[k] \leq T_A - k$. Some instances of φ for particular values of k are:

- $\varphi_A[0]$: $Act_A[0] \leq T_A$. This implies that there can be a maximum of T_A active processes in A, which is equivalent to the adequacy condition.
- $\varphi_A[1]$: $Act_A[1] \leq T_A - 1$. Processes running methods with annotation 1 or higher cannot exhaust all resources. There will always be one resource reserved for processes with the lowest annotation.
- $\varphi_A[T_A - 1]$: $Act_A[T_A - 1] \leq 1$. There can be at most one active process running a method with the maximum annotation $T_A - 1$.

Lemma 3.3.3. If a process is waiting in ℓ_0 to run a method $n{:}A$, and its enabling condition is not satisfied, then there is an active process running a method $m : A$ with annotation $\alpha(m) \leq \alpha(n)$.

Proof. Let $k = \alpha(n)$. Since n is not enabled, $k \geq t_A$. By the invariant φ_A for $k+1$, $Act_A[k+1] \leq T_A - (k+1)$. Hence,

$$\begin{aligned} Act_A[k+1] &\leq T_A - (k+1) \\ &\leq T_A - (t_A + 1) \\ &< T_A - t_A \\ &= Act_A \end{aligned}$$

Consequently, since the number of active processes with annotation at most k is given by $Act_A - Act_A[k+1]$, there must be one such process. □

Lemma 3.3.3 establishes that if there is a waiting process, then there is some active process running a method with annotation at most that of the waiting process. This is the

key element in the proof of deadlock avoidance, that can be constructed now by contradicting the existence of a minimal annotation of a waiting process in a deadlock state.

3.4 Deadlock Avoidance with the Height Annotation

In the previous section we introduced the protocol Basic-P and studied some of its properties for a generic annotation. We study now two particular annotations: height and local height. Local height requires the least resources and is necessary to prevent deadlocks: every annotation that prevents deadlock must map each method to at least its local height. Unfortunately, local height does not guarantee freedom from deadlock in all scenarios. A simple counterexample was provided by Example 3.2.2.

We prove here that Basic-P , when instantiated with height, does guarantee absence of deadlock. However, for many designs, height is too restrictive in its requirements and utilization of resources. To mitigate this problem, in Section 3.7 we study a sufficient condition that annotations must satisfy to guarantee deadlock freedom.

We assume that for every site A the number of resources T_A is at least the highest annotation of any method that runs in A in any call graph. We first prove one more auxiliary lemma.

Lemma 3.4.1. When using Basic-P with the height annotation, for every process P running a method with annotation 0 there is a continuation of the run in which P completes.

Proof. Let P be a process running a method n with annotation 0. Note that P can always progress when it is active, since n is a leaf node and therefore P performs no remote invocations. Thus, it is sufficient to show that P can eventually progress when it is waiting at ℓ_0. If $t_A > 0$, the enabling condition is satisfied and P can progress immediately. If $t_A = 0$, by Lemma 3.3.1, there exists an active process running a method $m : A$ with annotation $h(m) = 0$. This process, being active, can be scheduled to terminate, thereby incrementing t_A and unblocking P, which can then be scheduled to proceed. □

We are now ready to prove deadlock avoidance for Basic-P when used with the height annotation.

Theorem 3.4.2. Basic-P with height annotation guarantees absence of deadlock.

Proof. By contradiction, suppose that σ is a reachable deadlock state. Let P be a process in σ, running a method $n:A$ such that $h(n)$ is minimal, among all the processes present in σ. Consider the two possible cases:

1. $h(n) = 0$. By Lemma 3.4.1, P can eventually progress, contradicting deadlock.

2. $h(n) > 1$. If P is active, then by Lemma 2.4.5 it must have a waiting descendant, contradicting that P has minimal height. If P is waiting, then by Lemma 3.3.3 there exists an active process Q running a method $m:A$ with $h(m) \leq h(n)$. Again, $h(m) < h(n)$ contradicts the minimality of n. If $h(m) = h(n)$ then Q—being active—must have a waiting descendant, by Lemma 2.4.5. In this case, the the minimality of n is also contradicted. □

Theorem 3.4.2 provides a design methodology to guarantee deadlock free operation in all scenarios. First, annotate every call graph method with its height and provide every site A with at least as many threads as the maximum height of a method that resides in A. Then, at runtime, use BASIC-P as the local resource allocation protocol in every site. The disadvantage of using height as an annotation is that the number of resources needed in each site can be much larger than is strictly necessary. Using height implies not only an underutilization of resources, but also precludes the use of some useful idioms, as illustrated by the following example.

Example 3.4.3. Consider a system $\mathcal{S} : \langle \mathcal{R}, M, \{G_1, \ldots, G_k\} \rangle$. A simple idiom to force invocations of $G_1, \ldots, G_k$ to be performed sequentially is to introduce a new site S— called the *serializer*—and create a wrapper initial method $m:S$. The method m has as descendants the initial methods of all G_i. When using BASIC-P with the height annotation, the annotation of the new initial method is $h(m) = max\{h(G_1), \ldots, h(G_k)\} + 1$, which may be a large number. Moreover, S needs to be provided with exactly $T_S = h(m) + 1$ resources, even though annotating m with 0 and providing $T_S = 1$ would have sufficed. ⌟

Clearly, using height may be wasteful of resources. Fortunately, more efficient annotations can be used in most cases.

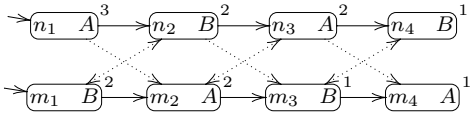


Figure 3.3: An annotated global call graph

3.5 Efficient Annotations

Intuitively, deadlock is caused by cyclic dependencies. Using Basic-P with an annotation that breaks all cyclic dependencies between waiting processes would prevent deadlock. Example 3.2.2 showed that the deadlock reached when local height is used as annotation is caused by the interaction of processes executing different call graphs. Thus, a check for cyclic dependencies must consider all call graphs at once. To capture this intuition we introduce annotated global call graphs.

Definition 3.5.1 (Annotated Global Call Graph). *Given a system $\mathcal{S} : \langle \mathcal{R}, M, \mathcal{G} \rangle$ and an annotation α, the annotated global call graph $G_{\mathcal{S},\alpha} : (M, \rightarrow, \dashrightarrow)$ has M as the set of vertices and contains two kinds of edges:*

- *$\rightarrow$ is the union of the remote invocation relations of all the call graphs.*
- *there is an edge $n \dashrightarrow m$ whenever n and m reside in the same site and $\alpha(n) \geq \alpha(m)$.*

Note that two methods can be related by $\dashrightarrow$ even if they belong to different call graphs.

Definition 3.5.2 (Dependency Relation). *Given a global call graph $G_{\mathcal{S},\alpha} : (M, \rightarrow, \dashrightarrow)$ we say that method n depends on method m, written $n \succ m$, if there is a path from n to m in $G_{\mathcal{S},\alpha}$ containing at least one $\rightarrow$ edge.*

An annotated global call graph has a *cyclic dependency* if some method n depends on itself, i.e., if for some n, $n \succ n$. Given a system $\mathcal{S}$, if an annotation α does not create cyclic dependencies in the annotated global call graph, we say that α is an acyclic annotation of $\mathcal{S}$.

Example 3.5.3. Figure 3.3 shows an annotated call graph, where transitive edges are not depicted, for clarity. Method m_1 depends on n_3 because $m_1 \dashrightarrow n_2 \rightarrow n_3$. However, m_1 does not depend on n_2 because the only path is $m_1 \dashrightarrow n_2$. This annotation is acyclic. ⌟

The following theorem provides a general sufficient condition on annotations for which BASIC-P guarantees deadlock free operation.

Theorem 3.5.4 (Annotation Theorem). The use of BASIC-P with an acyclic annotation guarantees absence of deadlock.

Proof. This proof follows closely that of Theorem 3.4.2. We first observe that, in the absence of cyclic dependencies, the dependency relation $\succ$ is a partial order on the set of methods M. By contradiction, assume that σ is a reachable deadlock state, and let P be a process present in σ running a method $n:A$ that is minimal with respect to $\succ$ (among all existing processes in σ). Consider the three possible cases:

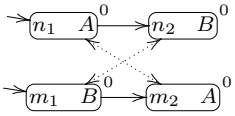
1. P is active. Then, by Lemma 2.4.5, P must have a waiting descendant Q running method m, but then $n \succ m$, contradicting the minimality of n.

2. P is waiting and $\alpha(n) = 0$. Then $t_A = 0$ (otherwise P could proceed, contradicting deadlock), and by Lemma 3.3.1, there exists an active process Q running $m:A$ with annotation 0. Therefore, Q can proceed, contradicting deadlock.

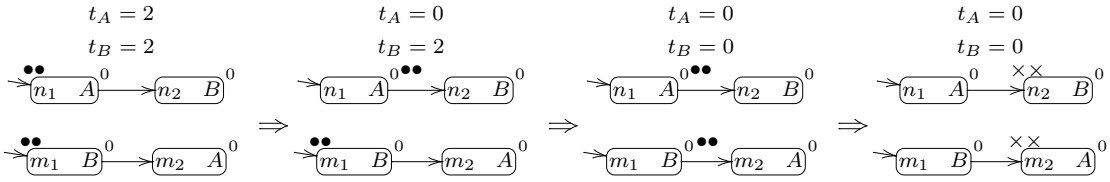
3. P is waiting and $\alpha(n) > 1$. By Lemma 3.3.3, there exists an active process Q running method $m:A$, so $\alpha(m) \leq \alpha(n)$. If Q is active and present in a deadlock, it must have a waiting descendant, by Lemma 2.4.5. Let this waiting descendant be running method m_2. It follows that $n \succ m_2$, which, again, contradicts the minimality of n. □

The condition of α being an acyclic annotation is known as the *annotation condition*. It is easy to see that the annotation condition implies that the annotation subsumes local height: if the annotation value of some method is smaller than its local height there is immediately a dependency cycle. On the other hand, height clearly satisfies the annotation condition. For many systems, however, there are acyclic annotations that are significantly smaller than height. For example, in Example 3.4.3, the serializer node can safely be given annotation 0, instead of 1 plus the maximum height of all call graph methods. The next chapter studies properties of acyclic annotations, including an algorithm that computes minimal acyclic annotations efficiently. The following example illustrates why acyclic annotations guarantee deadlock freedom.

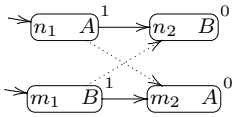
Example 3.5.5. Let us revisit the system of Example 2.4.4. Consider the protocol BASIC-P, together with local height annotation:



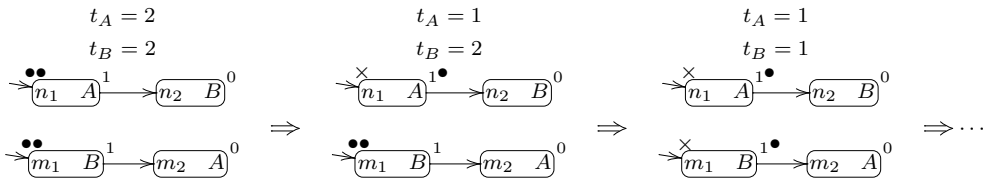
In this case, the annotation condition is not satisfied: the path $n_1 \to n_2 \dashrightarrow m_1 \to m_2 \dashrightarrow n_1$ is a dependency cycle. Therefore deadlock freedom is not guaranteed by the Annotation Theorem. Indeed, with all nodes annotated 0, BASIC-P is equivalent to ADEQUATE-P , and the same run that exhibits a deadlock in Example 2.4.4 is admitted here:



If the height annotation is used instead:

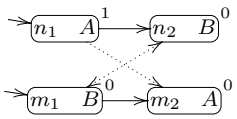


there is no dependency cycle, and the Annotation Theorem guarantees deadlock free operation. The run to deadlock is not accepted, since after the first process advances to execute n_1 the second process is not allowed to enter the method section of m_1. In other words, the last resource in A is reserved to execute m_2. Similarly, the last resource in B is reserved to run n_2. Pictorially, the run that previously led to deadlock becomes:



The height annotation is not, in this case, a minimal acyclic annotation. It is enough if one method among n_1 and m_1 takes the "pessimistic" approach of reserving the last resource

for the corresponding leaf node:

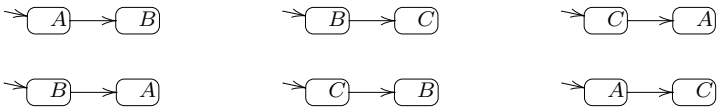


Even though $m_1 \dashrightarrow n_2 \dashrightarrow m_1$ is a cycle, it is not a *dependency* cycle, and it does not correspond to a potential blocking chain. This annotation is also acyclic, and BASIC-P guarantees deadlock avoidance, as ensured by the Annotation Theorem. ⌟

3.6 A Solution to the Dynamic Dining Philosophers

The dynamic dining philosophers was introduced in Section 1.1.3 to illustrate the kind of deadlock scenarios under study. Dynamic dyning philosophers generalizes the classical dining philosophers with several philosophers sharing a place at the table, and joining and leaving dynamically. Distributed solutions also restric the algorithms such that philosophers can only see the pile of chopsticks they are picking from. The number of philosophers present and the other piles are hidden.

Figure 3.4 shows an instance of the problem with three eating places and two chopsticks per pile. Figure 3.5 depicts a system description that captures this instance of dynamic dining philosophers. Piles are modeled as sites, all of them with initial size 2. Possible philosopher behaviors are modeled in the following call graph, where method names are omitted:



The call graphs in the left correspond to the two possible behaviors of philosophers sitting between piles A and B: either they pick a chopstick from A and then from B or the other way

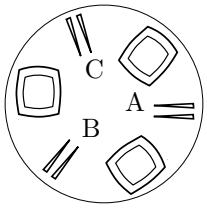


Figure 3.4: An instance of the dynamic dining philosophers

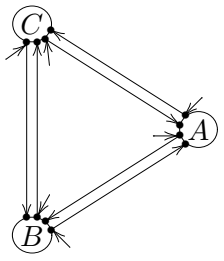
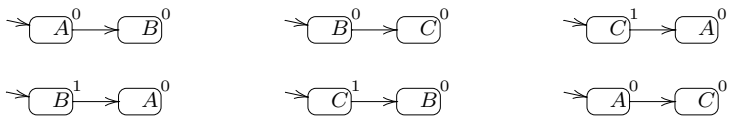


Figure 3.5: The corresponding system for the dynamic dining philosophers instance

around. Similarly, the middle graphs describe the behaviors of philosophers sitting between B and C, and the graphs on the right the behaviors of philosophers that sit between C and A. One acyclic annotation of this call graph is:



With this annotation, the annotation theorem guarantees absence of deadlock if Basic-P is used as allocation manager. This corresponds to the solution sketched in Section 1.1.3: it is enough if philosophers following a decreasing alphabetical order do not take the last chopstick when they pick from the first pile.

3.7 A More Efficient Protocol

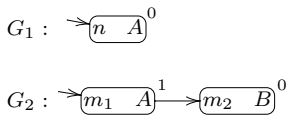
The protocol Basic-P can be refined to allow more concurrency while still preventing deadlock. The protocol Efficient-P , shown in Figure 3.6 exploits the observation that with acyclic annotations, every process running a method that requires just one resource can always terminate, independently of other parallel executions. The protocol Efficient-P maintains two local variables, t_A and p_A. The variable t_A, as in Basic-P, keeps track of the number of threads currently available, and p_A tracks the threads that are *potentially* available. The difference with Basic-P is that the number of potentially available resources is not decremented when a resource is granted to a process that runs a method with annotation 0, because these resources will always be returned. With Efficient-P fewer processes are blocked, thus increasing potential concurrency and improving resource utilization.

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

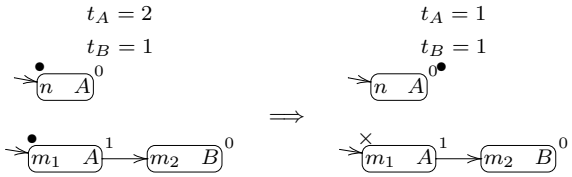
If $\alpha(n) = 0$ If $\alpha(n) > 0$

Figure 3.6: The protocol EFFICIENT-P

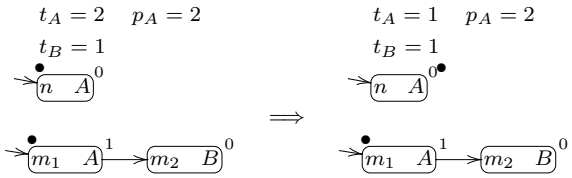
Example 3.7.1. Consider the system $\mathcal{S}: \langle\{A, B\}, \{n, m_1, m_2\}, \{G_1, G_2\}\rangle$ with $T_A = 2$ and $T_B = 1$ and the following call graphs, where the height annotation is depicted



Assume the following arrival of processes. First, P is spawned as an instance of G_1. Then, Q is created to run G_2. With BASIC-P, Q is blocked until P finishes and has released the resource in A:



With EFFICIENT-P, Q can run concurrently with P:



This illustrates how EFFICIENT-P allows more concurrent executions than BASIC-P. ⊿

The proof of deadlock avoidance of EFFICIENT-P follows similar steps as the one for BASIC-P , except that it uses specific invariants that make use of combined properties of p_A and t_A. Chapter 6 presents a family of allocation procotocols based on the invariant φ, and proves the correctness of all instances. The protocols BASIC-P and EFFICIENT-P are shown to be members of this family.

3.8 Summary

In this chapter we have introduced the first deadlock avoidance algorithms, implemented by protocols that are parameterized by annotations of the call graph methods. The protocol Basic-P guarantees freedom from deadlock by comparing the annotation value of the incoming request with the local resource availability. Upon a successful request, only one thread is locked. Some requests are denied depending on the resource utilization—even if there is a resource available—which prevents unobserved remote processes from creating a cyclic blocking chain. The correctness of Basic-P is captured by the Annotation Theorem, which establishes a sufficient condition on annotations to prevent deadlocks. The condition, known as the annotation condition, states that the annotation must be acyclic. Finally, we have presented the protocol Efficient-P, that increases concurrency while preserving deadlock freedom, by accounting separately for processes with the minimal annotation value, that are guaranteed to terminate in spite of unobserved remote processes.

Chapter 4

Annotations

The previous chapter has presented protocols that guarantee deadlock free operation. These protocols are parameterized by annotations of the methods in the form of natural numbers. The Annotation Theorem guarantees that if the annotations are acyclic then no deadlock is reachable. This chapter investigates the following questions: (1) how to efficiently compute minimal acyclic annotations; (2) whether deadlock freedom is compromised if an annotation is cyclic, (3) how to decide if there is a minimal annotation given constraints on the initial resources managed by the sites.

4.1 Problem Taxonomy

Let us first introduce some notation to refer to the problems that we study in this chapter. Given a system specification $\mathcal{S} : \langle \mathcal{R}, M, \mathcal{G} \rangle$ we use $\langle \mathcal{S}, \alpha, \mathbf{T} \rangle$ to denote whether the system $\mathcal{S}$ along with an annotation α and initial configuration of resources $\mathbf{T} : \{T_A = k_A\}_{A \in \mathcal{R}}$ has reachable deadlocks. In the next sections we study:

1. $\langle \mathcal{S}, ?, ? \rangle$: For a given system $\mathcal{S}$ with no constraints on the initial configuration of resources, we show how to compute a minimal acyclic annotation and determine the initial resources required.
2. $\langle \mathcal{S}, \alpha, \mathbf{T} \rangle$: Given $\mathcal{S}$ with cyclic annotation α and initial resources $\mathbf{T}$, we prove that deciding whether Basic-P guarantees absence of deadlock in all runs is an NP-complete problem.
3. $\langle \mathcal{S}, ?, \{T_A = k_A\}_{A \in X} \rangle$: Given $\mathcal{S}$ and a set of constraints on the number of resources for some of the sites $X \subseteq \mathcal{R}$, this is the decision problem of determining whether there

exists an acyclic annotation compatible with the restrictions. We study two sub-cases: first we show that the general case of arbitrary restrictions is NP-hard, and then we consider mutual exclusion as the only restriction, to conclude that this simpler case is in P.

4.2 Generating Minimal Annotations

In this section we study the problem $\langle \mathcal{S}, ?, ? \rangle$ of generating a minimal acyclic annotation for a given system $\mathcal{S}$. An acyclic annotation is minimal if no annotation value of any method in the call graph can be reduced without creating cycles. It follows that, in a minimal annotation, reducing more than one value also creates cycles. We begin by presenting an algorithm, called CalcMin, that computes minimal acyclic annotations. Then we prove that the algorithm is complete in the sense that it can generate every acyclic annotation.

CalcMin takes as input a call graph and a reverse topological order of its methods. This order is followed to calculate the annotation values; in other words, callee methods are visited before caller methods. Then, when calculating $\alpha(n)$, the annotations of all descendants of n have been computed. Figure 4.1 shows the pseudo-code of CalcMin.

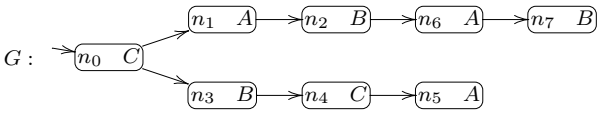
The algorithm computes, at the iteration for method n, the minimum possible value for $\alpha(n)$ such that no dependency cycle is created containing only n and the methods already visited. Lines 5-8 compute the set of methods that can be reached from n. These are

```
1: {Order M in reverse topological order}
2: {Let Reach_X = {m | x(→ ∪ ⇢)*m, x ∈ X}}
3: {Let Site_n = {m | n ≡_R m} }
4: for n = n_1 to n_|M| do
5:     R = {n}
6:     repeat
7:         R ← Reach_R
8:     until fix-point
9:     if R ∩ Site_n is empty then
10:        α(n) = 0
11:    else
12:        α(n) = 1 + max{α(m) | m ∈ R ∩ Site_n}
13:    end if
14: end for
```

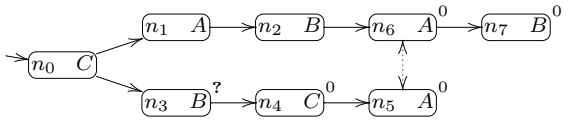
Figure 4.1: CalcMin: An algorithm for computing minimal acyclic annotations

methods m whose annotation has been computed in previous iterations, and for which $n \succ m$, independently of the value of $\alpha(n)$. Line 9 determines the methods that are candidates to precede n in a potential cycle. Finally, lines 10 or 12 assign to $\alpha(n)$ the minimum value that creates no cycle. The following example illustrates how CalcMin computes an annotation.

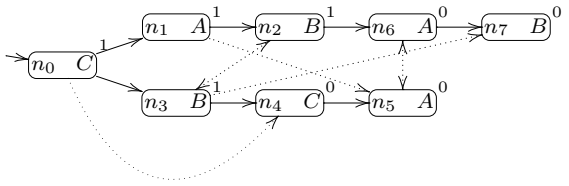
Example 4.2.1. Consider the system $\mathcal{S} : \langle \mathcal{R}, M, \{G\} \rangle$ with three sites $\mathcal{R} : \{A, B, C\}$, eight methods $M : \{n_0, \ldots, n_7\}$ and call graph G:



Assume that the topological order is $n_i < n_j$ whenever $i < j$. We show how CalcMin generates the annotation value for n_3; the values of α for n_4, n_5, n_6 and n_7 have already been computed in previous iterations:



In lines 5-8, the set R of methods reachable from n_3 is first approximated as $\{n_4, n_5\}$ because these are the descendants of n_3. Then, n_6 is added because it is reachable from n_5. Finally, $R = \{n_4, n_5, n_6, n_7\}$ considering also the descendants of n_6. This set is the fix-point. The only method in R that resides in the same site as n_3 is n_7, so $\alpha(n_3)$ is set to $1 + \alpha(n_7) = 1 + 0 = 1$. Continuing with the iterations for n_2, n_1 and n_0, the whole graph is annotated as follows:



⌟

Theorem 4.2.2. The algorithm CalcMin computes a minimal acyclic annotation.

Proof. We first show that CalcMin always computes an acyclic annotation. By contradiction, assume that the annotation generated is not acyclic. Given a dependency cycle, the method that is highest, according to $<$, is called the top of the cycle. Consider the cycle $\mathcal{C}$

that has a minimum top, n. Because of this choice of $\mathcal{C}$, before computing the annotation value of n there are no cycles, and the value computed for n creates a cycle. Moreover, the cycle is of the form: $n \to n_2 \cdots m \dashrightarrow n$. Consequently, $\alpha(n) \leq \alpha(m)$. But m is in $R \cap \text{Site}_n$ in line 12 so, $\alpha(n) > \alpha(m)$, a contradiction. Essentially, the algorithm guarantees that, after each step, no new cycle is created.

We now prove minimality. For each method n, every acyclic annotation satisfies:

$$\alpha(n) \geq 1 + max\{\alpha(m) \mid n \succ m \text{ and } n \equiv_{\mathcal{R}} m\}$$

Otherwise, a cycle $n \succ m \dashrightarrow n$ could be formed. In line 12, $R \cap \text{Site}_n$ is a subset of the methods that n finally depends on, so

$$\begin{aligned}\alpha(n) &= 1 + max\{\alpha(m) \mid m \in R \cap \text{Site}_n\} \\ &\leq 1 + max\{\alpha(m) \mid n \succ m \text{ and } n \equiv_{\mathcal{R}} m\}.\end{aligned}$$

This shows that α is a minimal annotation. □

We prove now that the algorithm CalcMin can generate every minimal acyclic annotation, simply by providing the right order. This is shown by, given a minimal acyclic annotation α, constructing an appropriate reverse topological order $<_\alpha$ from which CalcMin generates α.

Lemma 4.2.3. Every minimal acyclic annotation can be produced by CalcMin.

Proof. Given acyclic α, let $<_\alpha$ be an order compatible with $\to$, $\dashrightarrow$ and $\succ$. The existence of such an order (called preference order) follows directly from the acyclicity of α and is proved in Corollary 6.3.3. We show by complete induction on $<_\alpha$ that CalcMin($<_\alpha$) generates an acyclic minimal annotation β with $\beta(n) = \alpha(n)$ for every method n. Let n be an arbitrary method. By our choice of order, all methods m with $n \succ m$ have been visited before, and by the inductive hypothesis $\beta(m) = \alpha(m)$. The value $\alpha(n)$ creates no cycle, so in line 12, $\beta(n) \leq \alpha(n)$. Assume, by contradiction, that $\beta(n) < \alpha(n)$. In this case replacing $\alpha(n)$ by $\beta(n)$ makes α still an acyclic annotation which contradicts the minimality of α. Therefore, $\beta(n) = \alpha(n)$. □

4.3 Cyclic Annotations

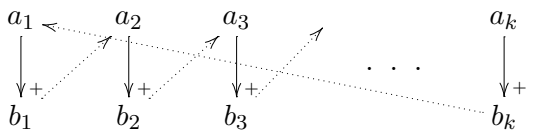
The previous section presented how to compute minimal acyclic annotations. An immediate question is what happens if a smaller annotation, which is necessarily a cyclic annotation, is used. In this case, the Annotation Theorem does not guarantee absence of deadlock, and Example 3.5.5 shows a case where deadlocks are indeed reachable. This section studies whether deadlocks are always reachable if the annotation is not acyclic. The main result is that, even though for some values of the initial resources deadlocks may not be present, deadlocks are always reachable if initial resources are increased sufficiently. This result exhibits a rather surprising anomaly: in the presence of cyclic dependencies, increasing the number of threads may introduce the possibility of deadlock in an originally deadlock free system. This anomaly resembles the Belady anomaly [BNS70] that shows, in the context of operating systems, that for some scheduling algorithms assigning more resources can degrade performance. This anomaly is clearly undesirable from an engineering perspective, since it is common practice to "over-provision", 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.

We first refine the notion of cyclic dependencies and introduce *simple dependency cycles* and *unavoidable deadlocks*. A cyclic dependency occurs when there is a sequence of methods $v_1, \ldots, v_k$, with $v_1 = v_k$ such that:

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

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

Definition 4.3.1 (Dependency Cycle). *A dependency cycle consists of two sequences of methods $\langle [a_1, \ldots, a_k], [b_1, \ldots, b_k] \rangle$, of the same length, such that for all i, there is an edge $a_i \rightarrow^+ b_i$, and an edge $b_i \dashrightarrow a_{i \oplus 1}$ (here $\oplus$ stands for addition modulo $k+1$):*

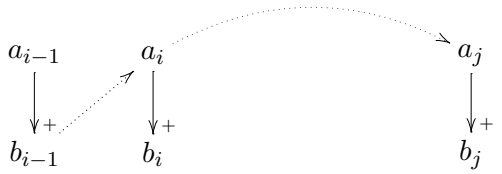


The methods a_i are called "above" or a-methods, and the b_i's are called "below" or b-methods. We say that a dependency cycle is *simple* if all a-methods reside in different sites.

Consequently, no two b-methods reside in the same site either.

Lemma 4.3.2. If an annotated global call graph $G_{\mathcal{S},\alpha}$ has a dependency cycle, it also has a simple dependency cycle.

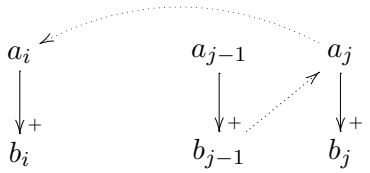
Proof. By contradiction, assume that there is a dependency cycle but no simple dependency cycle. Let $\mathcal{C} = \langle [a_1, \ldots, a_k], [b_1, \ldots, b_k] \rangle$ be a dependency cycle with minimum number of pairs of a-methods that reside in the same site. In $\mathcal{C}$ there exist a_i and a_j that reside in the same site (w.l.o.g. we assume $j > i$). Then, either $\alpha(a_i) \geq \alpha(a_j)$ or $\alpha(a_i) < \alpha(a_j)$. In the first case:



by transitivity of $\dashrightarrow$, $b_{i-1} \dashrightarrow a_j$ and therefore

$$\langle [a_1, \ldots, a_{i-1}, a_j, \ldots, a_k], \\ [b_1, \ldots, b_{i-1}, b_j, \ldots, b_k] \rangle$$

is a dependency cycle with strictly fewer pairs of a-methods residing in the same site. Similarly, if $\alpha(a_i) < \alpha(a_j)$, as shown in:



By transitivity $b_{j-1} \dashrightarrow a_i$, and then the sub-graph

$$\langle [a_i, \ldots, a_{j-1}], \\ [b_i, \ldots, b_{j-1}] \rangle$$

is a dependency cycle with fewer coincidences. In both cases the minimality of $\mathcal{C}$ is contradicted. Hence, there is a simple dependency cycle, as desired. □

The following definition models the sequence of remote invocations that a process must go through to execute a particular method:

Definition 4.3.3 (Path). *A path is a sequence of methods, starting from an initial method, that follows the descendant relation* $\rightarrow$*. The path* leading *to a method* n *is the ordered sequence of its ancestors. The methods* $n_1, \ldots, n_{k-1}$ *are called* internal *methods of the path* $\pi : (n_1, \ldots, n_{k-1}, n_k)$.

We introduce the notion of "unavoidable deadlock" to aid in reasoning about system states that will inevitably reach a deadlock. An unavoidable deadlock state will reach a deadlock if the processes involved are scheduled to execute. Hence, in every continuation of the run either the processes are not scheduled and starve, or the system reaches a deadlock. In either case the processes involved in an unaviodable deadlock cannot progress to termination. Unavoidable deadlocks are easier to create in proofs than are deadlocks.

Definition 4.3.4 (Unavoidable Deadlock). *A global state* σ *is an unavoidable deadlock state if no process present in* σ *terminates in any state reachable from* σ*.*

An alternative characterization is given by:

Lemma 4.3.5. If in state σ no process can individually proceed to completion even if continuously scheduled, then σ is an unavoidable deadlock state.

Proof. We show that if σ is not an unavoidable deadlock state, then there is a process that proceeds to termination if continuously scheduled. Consider a shortest run ζ extending σ for which an existing process P terminates. Clearly, there is no creation of new processes in ζ because, by monotonicity of protocols, one could remove all the **creation** transitions to produce a strictly shorter run in which P also terminates. If all transitions in ζ are related to P, the result is shown. 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 ζ after τ, would also have been enabled had τ not been taken, again by monotonicity of the protocols. Therefore, we can produce a shorter run by removing τ, which contradicts the minimality of ζ. □

4.3.1 Deadlocks with Cyclic Annotations

We show now that, in the presence of dependency cycles, an unavoidable deadlock can be reached if enough resources are initially present.

The method presented in this section calculates, given a system together with a cyclic annotation, some initial resources for which some run leads to deadlock. We calculate

these resources by generating a symbolic run in which when all the a-methods in a simple dependency cycle are executed, no more processes can visit any a-method, and consequently no b-method can be executed either. In effect, the processes reach an unavoidable deadlock state.

We construct the symbolic run as follows. Let $\langle [a_1, \ldots, a_m], [b_1, \ldots, b_m] \rangle$ be a simple dependency cycle for (distinct) sites $A_1, \ldots, A_m$. Let $\pi_1, \ldots, \pi_m$ be the paths leading to the a-methods $a_1, \ldots, a_m$. We build an execution by spawning k_i processes for each path π_i and scheduling the k_i processes that gain access to each of the methods in π_i simultaneously.

For every step in the symbolic execution we generate a constraint—on the possible values of k_i and the total number of threads $\{T_A\}$—that determines that the step is legal. Then, another constraint determines that when all the k_i processes execute their target method a_i the threads are exhausted. A global predicate consisting of the conjunction of all these intermediate constraints captures for which values the symbolic run has a concrete instance. Finally, we prove that this global constraint is always satisfiable. Each solution corresponds to an execution of the system that reaches an unavoidable deadlock.

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

$$\bigwedge_{A} T_A \geq 1 \qquad\qquad \bigwedge_{\pi_i} k_i \geq 1 \tag{4.1}$$

A (macro) step in the symbolic execution consists of all the k_i processes entering the method section of a method n in π_i. In terms of the computational model described in Chapter 2, this corresponds to k_i consecutive executions of the **method entry** transition. We use (n, k_i) to denote that all the k_i processes gain access to execute n, and say that the k_i processes "visit" method n. The symbol H represents the set of all visits that occur during a symbolic execution:

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

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

The steps of the different paths can be interleaved in many ways, each of which leads

to a different run and corresponds to a different set of constraints. We consider any total order $<$ on the set of visits 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 restrict our attention to *acceptable* total orders in which every a-method is the last method visited residing in its site (i.e., if (n, k_j) resides in A_i then $(n, k_j) < (a_i, k_i)$). Finally, we define $(H_A, <_A)$ to be the projection of $(H, <)$ for methods that reside in site A:

$$H_A \stackrel{\text{def}}{=} \{(n, k_i) \in H \mid site(n) = A\}.$$

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

Once an acceptable order is picked, the symbolic run is completely determined. The set of constraints created in the symbolic run that leads to an unavoidable deadlock is:

- **Resources are exhausted**: For all sites A_i, all threads are exhausted after the k_i processes visit a_i, such that no further executions of a_i are possible:

$$\psi_i \ : \ T_{A_i} - \sum_{(n,k_j) \in H_{A_i}} k_j = \alpha(a_i). \tag{C1}$$

  Note that $\sum_{(n,k_j) \in H_{A_i}} k_j$ corresponds to the total number of resources granted in site A_i in the whole execution. This constraint forces the remaining threads in A_i to be $\alpha(a_i)$, which is insufficient for any subsequent visit to a_i, according to the enabling condition for Basic-P. Consequently, no subsequent visit to $b_{i \ominus 1}$ is allowed either.

- **The run is feasible**: For all intermediate methods $(n, k_i) \in H$ (i.e., $n \neq a_i$) the protocol grants the resource allocation to all k_i processes. Assuming that method n resides in A, this is expressed by

$$\phi_{(n,k_i)} \ : \ T_A - \sum_{(m,k_j) \leq_A (n,k_i)} k_j \geq \alpha(n). \tag{C2}$$

  The term $\sum_{(m,k_j) \leq_A (n,k_i)} k_j$ accounts not only for the resources allocated by the k_i processes visiting n, but also for all the previous visits to methods in A by processes following any path.

Finally, to prove that an unavoidable deadlock is reached it is sufficient to show that

the following global constraint—together with (4.1)—is satisfiable:

$$\Phi \ : \ (\bigwedge_{\nu \in H} \phi_\nu) \wedge (\bigwedge_{a_i} \psi_i)$$

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

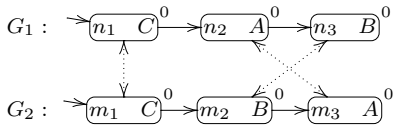
$$T_{A_i} = \sum_{(n,k_j) \in H_{A_i}} k_j + \alpha(a_i). \tag{C1'}$$

This equation gives a means to compute the value of T_{A_i} once all the k_i values are determined. Since $\alpha(a_i) \geq 0$ and $k_i \geq 1$, then $T_{A_i} \geq 1$ and this equation is consistent with (4.1). Using (C1') we simplify the constraint (C2) corresponding to $\phi_{(n,k_i)}$ to the following form:

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

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

Example 4.3.6. Consider a scenario with the following annotated global call graph:



This annotated graph has a simple dependency cycle $\langle [n_2, m_2], [n_3, m_3] \rangle$. If the initial resources allocated are $T_A = 1$, $T_B = 1$ and $T_C = 1$ no deadlock is reachable. To see this, observe that only one process can be granted access to either method n_1 or m_1, so the fact that $T_C = 1$ and that the root methods have annotation 0 serializes the access to the rest of the methods in both call graphs. The serialization breaks the cyclic contention that the annotation condition intends to capture. However, using the technique outlined above we show that allocating more resources (by increasing T_C) could lead to a deadlock. The two paths leading to a-methods are $\pi_1 = (n_1, n_2)$ and $\pi_2 = (m_1, m_2)$. Let k_1 denote the number of processes following π_1 and let k_2 follow π_2. The set of visits is:

$$H = \{(n_1, k_1), (n_1, k_1), (m_1, k_2), (m_2, k_2)\}.$$

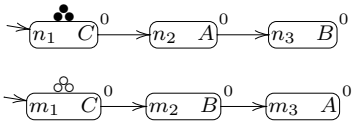
We pick the following acceptable total order $<$, that respects the topological order of π_1 and π_2 and for which the a-methods are the last visits of their sites: $(n_1, k_1) < (m_1, k_2) < (n_2, k_1) < (m_2, k_2)$. The set of constraints is:

$$\begin{aligned} T_C - k_1 &\geq \alpha(n_1) \\ T_C - k_1 - k_2 &\geq \alpha(m_1) \\ T_A - k_1 &= \alpha(n_2) \\ T_B - k_2 &= \alpha(m_2) \end{aligned}$$

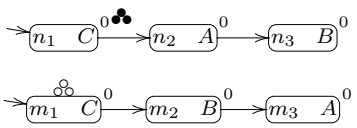
Simplifying with the numerical values of the annotation α and using the substitutions (C1') and (C2'):

$$\begin{aligned} T_C - k_1 &\geq 0 & T_A &= k_1 \\ T_C - k_1 - k_2 &\geq 0 & T_B &= k_2 \end{aligned}$$

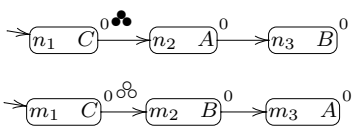
This system is clearly satisfiable, as shown by picking $k_1 = k_2 = 1$, and $T_A = T_B = 1$ and $T_C = 2$. In other words, the following sequence leads to an unavoidable deadlock. In the diagrams, ♣ represents the set of k_1 processes traversing path π_1, and ♧ symbolizes the set of k_2 processes traversing π_2. First, k_1 and k_2 processes are spawned:



Then, according to $<$, the k_1 processes gain access to n_1:

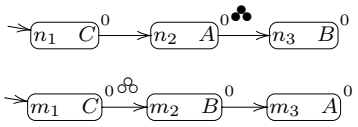


after which, the k_2 processes visit m_1:

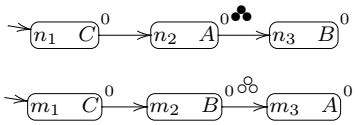


At this point, processes start visiting the a-nodes of the dependency cycle. First, the k_1

processes visit n_2:



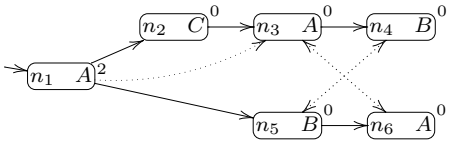
and after them, the k_2 processes visit m_2:



At this point $t_A = t_B = 0$, and consequently none of the processes can proceed independently to completion, so the deadlock is unavoidable. ⌟

The previous discussion shows that if we violate the annotation condition, even if we come up with a set of resources ($T_A = 1$, $T_B = 1$ and $T_C = 1$) that avoids deadlock, by allocating more resources ($T_A = 1$, $T_B = 1$ and $T_C = 2$) there is a possibility of deadlock. The following example shows a more sophisticated scenario, were paths leading to the methods causing the deadlock have common ancestors:

Example 4.3.7. Consider a scenario with the following annotated global call graph

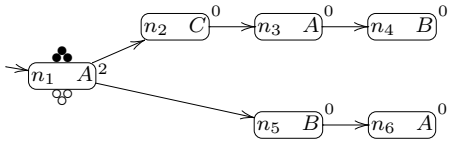


It is easy to see that, even though there is a simple dependency cycle $\mathcal{C} : \langle [n_3, n_5], [n_4, n_6] \rangle$, with initial resources $T_A = 3$, $T_B = T_C = 1$ the system cannot reach a deadlock. The reason, again, is that n_1 serializes accesses to the rest of the call graph. After a process executes n_1 no other process can become active in n_1. However, there are still 2 free resources in A, which breaks the potential cyclic conflict. Again, we show how to compute minimal initial resources that exercise the dependency cycle.

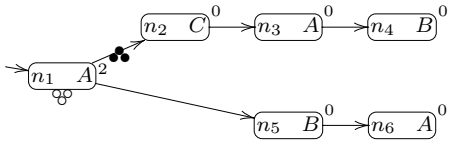
The cycle $\mathcal{C}$ generates the paths $\pi_1 : (n_1, n_2, n_3)$ and $\pi_2 : (n_1, n_5)$. Path π_1 represents the sequence of methods that are visited prior to n_3, while path π_2 contains the sequence ending in n_5. Observe that method n_1 is shared among the two paths. Let k_1 processes follow path π_1 and k_2 processes follow π_2. The set of visits for these two paths is

$$H = \{(n_1, k_1), (n_2, k_1), (n_3, k_1), (n_1, k_2), (n_5, k_2)\}.$$

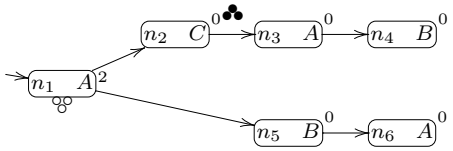
One acceptable total order is $(n_1, k_1) < (n_2, k_1) < (n_1, k_2) < (n_5, k_2) < (n_3, k_1)$. The symbolic run starts by spawning all the k_1 and k_2 processes



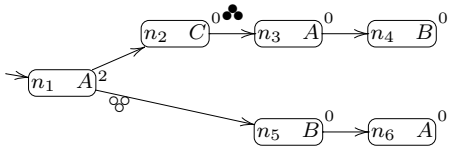
First, the k_1 processes gain access to n_1,



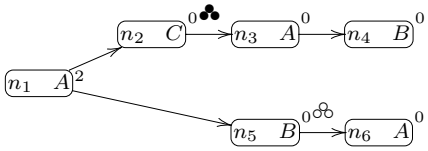
and then execute n_2:



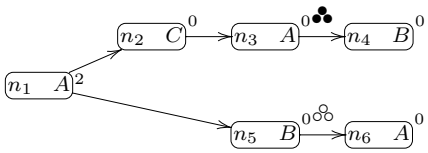
After which, the k_2 processes gain access to n_1.



At this point, all intermediate methods have been visited. Now, the k_2 processes visit the a-method n_5:



and, finally, the k_1 processes visit n_3:



The set of constraints generated is:

$$\begin{aligned} T_A - k_1 &\geq \alpha(n_1) \\ T_C - k_1 &\geq \alpha(n_2) \\ T_A - k_1 - k_2 &\geq \alpha(n_1) \\ T_B - k_2 &= \alpha(n_5) \\ T_A - (k_1 + k_2) - k_1 &= \alpha(n_3) \end{aligned}$$

which rewritten, according to (C1') and (C2'):

$$\begin{aligned} k_2 + k_1 &\geq 2 & \qquad T_B &= k_2 \\ T_C - k_1 &\geq 0 & \qquad T_A &= k_1 + k_2 + k_1 \\ k_1 &\geq 2 \end{aligned}$$

This system of equations is clearly satisfiable. One possible solution is $k_1 = 2$, $k_2 = 1$, and $T_A = 5$, $T_B = 1$, $T_C = 2$. In other words, if the sites A, B and C have initially available 5, 1 and 2 threads (resp.), then 2 processes can be spawned to follow the path π_1, and 1 to follow π_2 causing a deadlock if properly scheduled. ⌟

We now show the general result. Every scenario that violates the annotation condition can potentially reach a deadlock:

Theorem 4.3.8. 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 annotated global call graph $G_{\mathcal{S},\alpha}$ has dependency cycles. Consider a simple cycle $\mathcal{C}$ which, by Lemma 4.3.2, always exists. There is an acceptable order for visiting the methods in $\mathcal{C}$: the order $<$ where first all internal methods of every path π_i are visited in any topological order, and then all a_i are visited. Using any admissible order the generated set of constraints (C1') and (C2') is satisfiable. The following values of k_i and T_A satisfy all the constraints:

(1) if a site A does not appear in any constraint then assign $T_A = 1$.

(2) Take k_i to be the largest value of the right hand side of any formula $\phi_{(n,k_i)}$ where k_i appears. This way, all (C2') are satisfied.

(3) Compute the values of T_{A_i} using (C1').

(4) Finally, if some site A is visited in some intermediate method but no a_i resides in A, simply pick T_A to be the addition of all other elements appearing in all constraints involving T_A. 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 $T_A > c$ for constants c. Therefore, given any set of resources, a scenario with reachable deadlocks can always be built for an annotation with dependency cycles by allocating more resources.

4.4 Deciding Deadlock Reachability

In this section we study the decision problem $\langle \mathcal{S}, \alpha, \mathbf{T} \rangle$: we show that checking deadlock reachability for a fixed number of resources when the annotation is cyclic is computationally hard. These results indicate that, in practice, the annotation condition must be fulfilled. We first present a non-deterministic algorithm that decides in polynomial time whether a system has reachable deadlocks. Then, we introduce a reduction from 3-*CNF* to deadlock reachability that proves that the decision problem $\langle \mathcal{S}, \alpha, \mathbf{T} \rangle$ is NP-hard.

Lemma 4.4.1 (Finding Deadlocks). Given a system $\mathcal{S}$, cyclic annotation α, and assignment of resources $\mathbf{T} : \{T_A = k_A\}_{A \in \mathcal{R}}$, there exists a non-deterministic algorithm that decides in polynomial time whether a deadlock is reachable.

Proof. First, we say that a proper process is "relevant" in an execution if it is granted some resource. It is clear that if there is a run to a deadlock, then there is a run where only relevant processes exist, because the run obtained by removing irrelevant processes also reaches a deadlock.

Corollary 6.3.3 and Theorem 6.3.5 show that every reachable state can be reached by Basic-P performing only allocations, and following some topological order. We show now a non-deterministic algorithm that first guesses a deadlock state σ, and then guesses an order $<$ of allocations such that Basic-P reaches σ following $<$. The run is constructed by merging all allocations of a single method n into a macro step in which more than one process simultaneously acquire a resource in n.

Given a system specification $\langle \mathcal{S}, \alpha, \mathbf{T} \rangle$ a state of the algorithm is a vector $\langle p_1, \ldots, p_{|M|} \rangle$, where entry p_n represents the number of active processes running method n. A macro step

is represented by $\vdash_n^k$, corresponding to k processes gaining access to method n:

$$\langle p_1, \ldots, p_n, \ldots, p_{|M|} \rangle \vdash_n^k \langle p_1, \ldots, p_n + k, \ldots, p_{|M|} \rangle$$

where the only entry modified is p_n. A run is a sequence $\sigma_1 \vdash_{n_1}^{k_1} \sigma_2 \vdash_{n_2}^{k_2} \ldots \vdash_{n_l}^{k_l} \sigma_l$ of macro steps. A run is legal if σ_1 is $\langle 0, \ldots, 0 \rangle$, and if every state σ_{i+1} is obtained from σ_i by a legal (macro) allocation. It is easy to establish that a macro allocation $\vdash_{n_i}^{k_i}$ is legal, by checking that:

1. the enabling condition of BASIC-P for method n_i holds in σ_i for all the k_i processes,
2. the only entry modified is p_{n_i} which is increased by exactly k_i units, and
3. the parent method n of method n_i satisfies $p_n \geq p_{n_i} + k_i$, i.e., there are enough caller processes to perform all the remote invocations.

A legal run follows a total order $<$ if all the steps are carried out following it: $n_i < n_{i+1}$ for all i. This implies that the maximum length of a run that follows some total order is $|M|$. The state of the algorithm can be encoded in size linear in the size of the specification, each step can be checked in linear time, and the final state being a deadlock can also be checked in linear time. Therefore, this algorithm decides deadlock reachability in non-deterministic polynomial time. □

Lemma 4.4.2 (Deadlock Reachability). Given a system $\mathcal{S}$, cyclic annotation α and assignment of resources $\mathbf{T} : \{T_A = k_A\}_{A \in \mathcal{R}}$, deciding whether a deadlock is reachable is NP-hard.

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

$$\langle \mathcal{S}, \alpha, \{T_A = 1\}_{A \in \mathcal{R}} \rangle$$

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

- **Sites:** The set of sites $\mathcal{R}$ includes one site K_i per clause C_i and one site V_j per variable X_j:

$$\mathcal{R} \stackrel{\text{def}}{=} \{K_i\} \cup \{V_j\}.$$

- **Methods:** For each variable X_j we introduce two call graph methods, $x_j : V_j$ and $\overline{x_j} : V_j$. Similarly, for every clause C_i there are two methods, $a_i : K_i$ and $b_i : K_i$. The set of initial methods I contains all the methods corresponding to variables:

$$\begin{aligned} M &\stackrel{\text{def}}{=} \{(x_j : V_j), (\overline{x_j} : V_j) \mid \text{for every variable } X_j\} \cup \\ &\quad \{(a_i : K_i), (b_i : K_i) \mid \text{for every clause } C_i\} \\ I &\stackrel{\text{def}}{=} \{(x_j : V_j), (\overline{x_j} : V_j) \mid \text{for every variable } X_j\} \end{aligned}$$

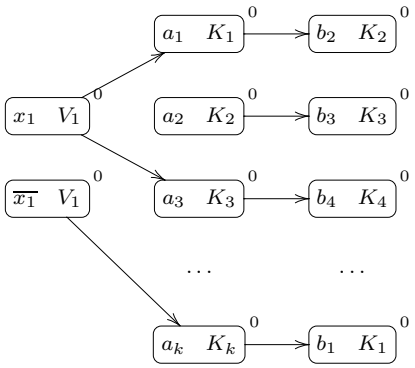
- **Resources:** The total number of resources is set to $T_{V_j} = 1$ for all variable sites and $T_{K_i} = 1$ for all clause sites.

- **Annotations:** The annotation of every method is 0, the only possible value consistent with $T_{V_j} = 1$ and $T_{K_i} = 1$. Consequently, there can be at most one active process running each method.

- **Edges:** Every method a_i is connected through a remote invocation edge to the method $b_{i \oplus 1}$ of the clause with the next index, including $a_k \to b_1$ for the last clause. Given that all annotation values are 0, this immediately creates a simple dependency cycle in the annotated graph:

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

This is the only cycle in the call graph. Finally, there is an edge from a variable x_i to all the clauses where X_i appears in positive form and one edge from $\overline{x_i}$ to all the clauses where X_i appears in negative form. For example, if X_1 appears in clauses C_1 and C_3, and $\overline{X_1}$ appears in clause C_k, the call graph will include:



Since, for all variables, the call graph methods x_i and $\overline{x_i}$ reside in the same site V_i and their annotation is 0, in any execution, at most one of them can have an active process. This corresponds to picking a valuation for the variable X_i. Then, the only clause methods that can have active processes are those with some process in a predecessor variable method: this corresponds to a clause being satisfied. Therefore, there is a run to a deadlock (exercising the only cycle in the graph), if and only if all the clauses can be satisfied. Since 3-*CNF* is NP-hard, so is the $\langle \mathcal{S}, \alpha, \mathbf{T} \rangle$ problem. □

Lemmas 4.4.1 and 4.4.2 imply:

Theorem 4.4.3. Deciding whether $\langle \mathcal{S}, \alpha, \mathbf{T} \rangle$ has reachable deadlocks, where α is cyclic, is NP-complete.

4.5 Computing Annotations with Resource Constraints

In many scenarios, there are imposed constraints on the number of resources available in certain sites. Since the algorithm presented in Section 4.2 generates all minimal acyclic annotations, it immediately provides a decision procedure for the problem $\langle \mathcal{S}, ?, \{T_A = k_A\}_{A \in X} \rangle$ of whether an acyclic annotation exists that accommodates these constraints. One can guess the order $<$, generate the annotation α with CalcMin($<$) and check whether α satisfies the constraints. Hence:

Lemma 4.5.1. Checking whether there is an acyclic annotation for $\langle \mathcal{S}, ?, \{T_A = k_A\}_{A \in X} \rangle$ is in NP.

We show now that the problem $\langle \mathcal{S}, ?, \{T_A = k_A\}_{A \in X} \rangle$ is NP-hard for arbitrary values of k_A, but it can be decided in polynomial time if all the restrictions are of the form $k_A = 1$.

4.5.1 Arbitrary number of Resources

Lemma 4.5.2. The problem $\langle \mathcal{S}, ?, \{T_A = k_A\}_{A \in X} \rangle$ is NP-hard.

Proof. We use, again, a reduction from 3-*CNF*. First, every 3-*CNF* formula θ can be transformed into an equi-satisfiable formula θ' by rewriting each clause $C_j : (V_1 \vee V_2 \vee V_3)$, where V_1 stands for a variable X_1 or its negation $\overline{X_1}$, as follows

$$C'_j : (V_1 \vee Y_j \vee Z_j) \wedge (V_2 \vee \overline{Y_j}) \wedge (V_3 \vee \overline{Z_j}).$$

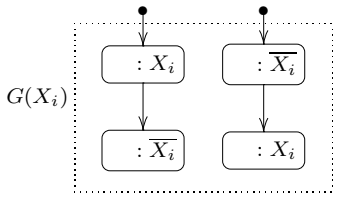
The auxiliary variables Y_j and Z_j are introduced to separate the occurrences of the different variables in the original formula. We say that θ' is in *separated* normal form (SNF). This transformation increases the number of variables by at most $2|C|$, where $|C|$ is the number of clauses in the original formula, and increases the number of clauses by a factor of 3. Consequently, the generated formula is linear in the size of the original one.

Given a formula θ in SNF we build a distributed system $\mathcal{S} : \langle \mathcal{R}, M, \mathcal{G} \rangle$—linear in the size of the formula—and a problem specification $\langle \mathcal{S}, ?, \{T_A = k_A\}_{A \in \mathcal{R}} \rangle$, such that the system admits an acyclic annotation if and only if there is a satisfying valuation of θ.

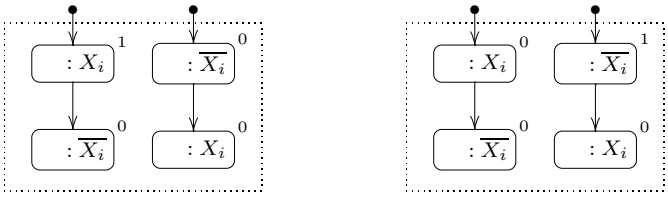
- **Resources:** The resource constraint $\mathbf{T}$ is set to $\{T_A = 2\}$ for all sites. This enforces all feasible annotations to be $\alpha(n) \leq 1$, for every method n.

- **Sites:** For each of the variables X_i in the formula θ' we introduce two sites, X_i and $\overline{X_i}$. The former represents the positive occurrence, while the latter the negative.

- **Methods and Edges:** For each of the variables X_i in the formula we introduce the following "variable gadget":

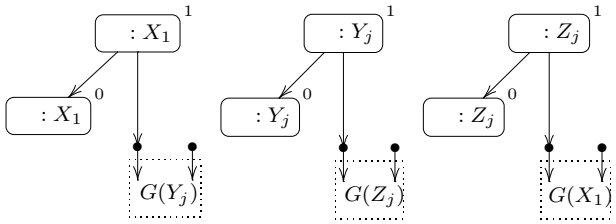


The only two possible acyclic annotations of this gadget that respect the constraints $T_{X_i} = 2$ and $T_{\overline{X_i}} = 2$ are:

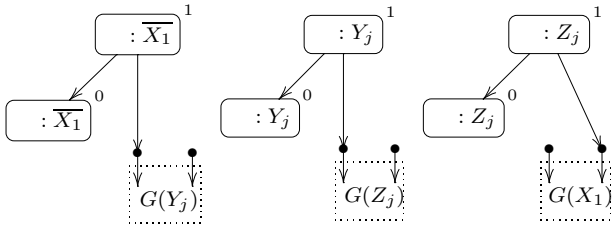


There is a one-to-one correspondence between acyclic annotations of these gadgets and valuations as follows: an annotation $\boxed{: X_i}^{\,1}$ denotes that variable X_i is false in the corresponding valuation, while $\boxed{: \overline{X_i}}^{\,1}$ denotes that X_i is true. For each variable

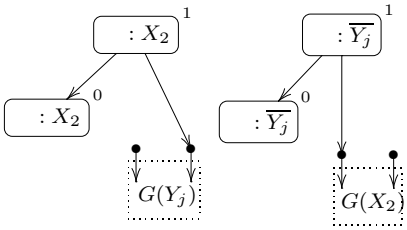
occurrence in a clause $C_j : (X_1 \vee Y_j \vee Z_j)$ we introduce the following "clause gadget", where we depict also the only possible acyclic annotation:



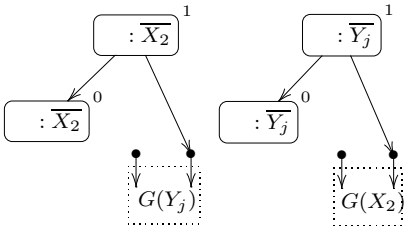
Similarly, if the clause is of the form $C_j : (\overline{X_1} \vee Y_j \vee Z_j)$:



For each of the clauses $(X_2 \vee \overline{Y_j})$ the gadget is:



Similarly, if the clause is $(\overline{X_2} \vee \overline{Y_j})$



The gadgets for the third sub-clause—either $(X_3 \vee \overline{Z_j})$ or $(\overline{X_3} \vee \overline{Z_j})$—are defined analogously.

The separation variables Y_j only occur once in positive form and once in negative form. Then the only possible cycles in the graph, once a valuation for all variables has been picked, involve all the upper methods in some clause gadget. This cycle exists if and only if not all the clauses are satisfied. Therefore, if all clauses are satisfied the induced annotation has no cycles, and if there is an annotation with no cycles the corresponding valuation is satisfying. Since 3-*CNF* is NP-hard, this reduction implies that checking whether a graph admits an acyclic annotation, with restrictions $\{T_A = k_A\}$ for $k_A \geq 2$, is also NP-hard. □

4.5.2 Mutual Exclusion Resources

We show now that the problem $\langle \mathcal{S}, ?, \{T_A = k_A\}_{A \in X} \rangle$ becomes tractable if all the constraints are of the form $k_A = 1$. That is, the only restriction is that some of the resources must be accessed in mutual exclusion, while the initial amount of all the other resources is unrestricted.

Lemma 4.5.3. The problem $\langle \mathcal{S}, ?, \{T_A = 1\}_{A \in X} \rangle$ is in P.

Proof. Consider the partially annotated graph G^α that only contains annotation values, $\alpha(n) = 0$, for all methods residing in those sites $A \in X$ that are marked as mutual exclusion resources (0 is the only possible annotation for these methods). If there are dependency cycles in G^α then there is no acyclic annotation since all fully annotated graphs extend G^α. Let $<$ be any reverse topological order that extends the dependencies $\succ$ present in G^α, which exists if G^α is acyclic.

Now, the algorithm CalcMin($<$) generates an acyclic annotation β. When $n : A$ is visited in CalcMin, no descendant of n can reach any method that resides in A, since that would imply the existence of a cycle in G^α. Therefore, $\beta(n)$ receives value 0 in line 10 of CalcMin, and β is an acyclic annotation that extends α, as desired. □

In case the technique described in Lemma 4.5.3 fails there is no protocol that can provide deadlock avoidance without communication. In practice, the alternative solution is to use deadlock prevention for some of the resources to break the cycles in the partially annotated call graph G^α. Lemma 4.5.3 also provides an efficient conservative procedure to check the feasibility of the general problem, presented in Section 4.5.1. Some of the constrained

resources can be over-restricted to be binary semaphores. Then Lemma 4.5.3 can be used to check feasibility, because every solution with binary semaphores is a solution to the general problem.

4.6 Summary

The Annotation Theorem establishes a criteria, the annotation condition, to guarantee that the protocol Basic-P provides deadlock avoidance. This chapter has studied some properties of annotations. First, we have designed the algorithm CalcMin that computes minimal acyclic annotations, and presented a proof that every minimal annotation can be generated by this algorithm. We have also shown that if the annotation condition is not satisfied, then the following anomaly occurs: even when some systems can be deadlock free for certain initial resources, every system with a cyclic annotation will have reachable deadlocks if initial resources are increased enough. In this chapter we have also proved that deciding whether a system with a given cyclic annotation and initial resources has reachable deadlocks is NP-hard. Therefore, as a design principle, to ensure that deadlock freedom is guaranteed, acyclic annotations must be used. Finally, we have shown that if some of the sites have restrictions in their amount of initial resources, computing acyclic annotations is an NP-hard problem. This problem becomes polynomially solvable if the only restrictions state that some resources must be accessed in mutual exclusion.

Chapter 5

Liveness

This chapter studies liveness: how to guarantee that every individual process eventually progresses. The protocols presented in previous chapters ensure that no deadlock can be reached, but there are still runs in which some processes starve, while other processes advance. We present here a new deadlock avoidance protocol that guarantees individual liveness, while still operating on local data. Interestingly, this algorithm allows more concurrency than all previous protocols. Finally, we discuss the implementation trade-offs, and present experimental results comparing all these protocols.

5.1 Local Schedulers

We begin by taking a closer look at how resources are managed. Deadlock avoidance protocols implement the resource allocation policy by deciding which requests are enabled. However, protocols do not control which enabled process, if there is more than one, gains access. In the model of computation introduced in Chapter 2, the environment selects which transition fires, among the enabled ones. We refine this situation here by introducing local schedulers.

At each site, resource allocation is implemented by a *controller* consisting of two cooperating components: an allocation manager and a scheduler. The allocation manager, implemented by a runtime protocol, decides which requests are safe. The scheduler arbitrates among the safe processes whenever there is a competition. Thus, when a new incoming request arrives, the controller proceeds by consulting the allocation manager about whether the request can be granted safely.

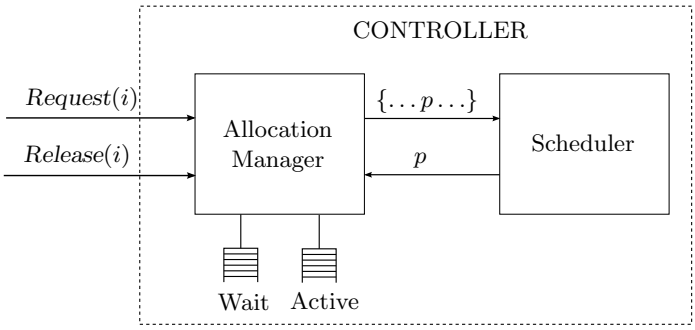


Figure 5.1: A schematic view of a resource allocation controller

- If the request is safe, a unit of resource is assigned and the protocol variables are updated accordingly.

- If the request is not safe, the process is inserted in a waiting queue.

Upon release, the controller delegates to the allocation manager the computation of the subset of processes in the waiting queue whose pending request becomes safe. Then, the controller transfers to the scheduler the job of picking one of these safe processes, which receives the resource. This interaction takes place until either the waiting queue is empty, or there are no more enabled processes. This model of the resource controller is depicted in Figure 5.1.

As for deadlock avoidance protocols, we are interested in schedulers that operate only on local data, which we call *local schedulers*. The ability of the algorithm studied in this chapter to provide individual liveness is based on the following property of schedulers.

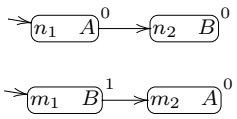
Definition 5.1.1 (Fair Scheduler). *A scheduler is called fair if no process can be offered infinitely often without being selected.*

This fairness condition essentially restricts the choices of the environment with respect to how the **method entry** transitions can be interleaved. In the context of reactive systems this notion of fairness is known as *compassion* or *strong fairness*. There are many scheduling policies that are fair in this sense: earliest deadline first, earliest creation first, etc. Section 5.4.2 below shows how to build an efficient fair scheduler.

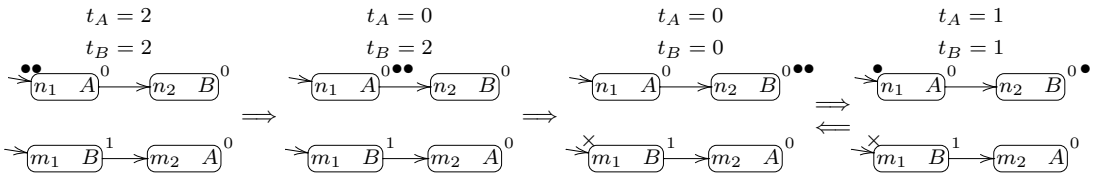
5.2 Deadlock versus Starvation

Clearly, deadlock implies starvation because no process present in a deadlock progresses. However, starvation can be present for other reasons, like processes coordinating to monopolize the resources, or schedulers that discriminate against some participant. The Annotation Theorem establishes that Basic-P guarantees that some process in the system can progress. This does not necessarily imply that every individual process can eventually progress. In fact, Basic-P allows runs with starvation, as the following example shows:

Example 5.2.1. Consider the following system, with initial resources $T_A = T_B = 2$:



The annotation is acyclic so Basic-P guarantees that no deadlock can be reached. We exhibit a run in which a process starves. The run begins with two processes starting their execution in n_1. After these processes perform the remote call to n_2 and become active, all the resources are in use. At this point, a new process P is spawned to execute m_1. As $t_B = 0$, the allocation manager—implemented by Basic-P—indicates that P must wait. Even if one of the processes executing n_2 finishes and releases its resources, P is still blocked, because the request for m_1 requires the availability of two resources in B. A new process executing n_1, however, can start, because running n_1 and n_2 only requires one resource in A and then one in B, both of which are available. Repeating this pattern results in an execution in which $t_B \leq 1$ in all future states. Hence the entry condition for P is never enabled, and P will wait forever, independently of any scheduler. This run is depicted below:



The annotation value $\alpha(m_1) = 1$ instructs Basic-P to take a pessimistic approach for requests to execute m_1. Unfortunately, this pessimism allows a malicious coalition of processes to monopolize the resources in B, and prevent m_1 from ever being executed. ⌟

5.3 A Liveness Protocol

We present in this section a protocol that provides liveness. An acyclic annotation of a call graph provides, for every method n, a measure of the number of (directly or indirectly) dependent methods that execute in the same site as n. We first revisit the invariant φ, introduced in Lemma 3.3.2, which is the essential element in the proof of deadlock avoidance of Basic-P. For every site A and annotation value k, there are never more than $T_A - k$ active processes executing methods with annotation value k or higher. Recall that the symbol φ_A denotes the clause of φ for site A. A global state that satisfies φ_A in all sites A is called a φ-state. Maintaining φ invariant is one of the properties of Basic-P, but sometimes Basic-P disables transitions that would also preserve φ. We propose here a protocol that grants precisely all requests that keep the system in a φ-state, and show that doing so guarantees individual liveness.

5.3.1 Protocol Schema

To express φ_A more formally we first introduce and review some notation. Recall that $act_A[k]$ represents the number of active processes in A executing methods with annotation value k, and $Act_A[k]$ is a shorthand for $\sum_{j \geq k} act_A[j]$. The property $\varphi_A[k]$ holds if the number of active processes executing methods with annotation k or higher does not exceed $T_A - k$, that is,

$$\varphi_A[k] \stackrel{\text{def}}{=} Act_A[k] \leq T_A - k.$$

The invariant φ that the protocol must preserve is then:

$$\varphi \stackrel{\text{def}}{=} \bigwedge_{A \in \mathcal{R}} \varphi_A \qquad\qquad \varphi_A \stackrel{\text{def}}{=} \bigwedge_{k} \varphi_A[k]$$

Here, k ranges over all annotation values of methods residing in A. Let $Act_A^{(i)}[j]$ and $act_A^{(i)}[j]$ represent the values of $Act_A[j]$ and $act_A[j]$ after a process requesting a resource to run a method with annotation i becomes active (i.e., $act_A[i]$ is incremented):

$$act_A^{(i)}[j] \stackrel{\text{def}}{=} \begin{cases} act_A[j] & \text{if } j > i \\ act_A[j] + 1 & \text{if } j = i \\ act_A[j] & \text{if } j < i \end{cases} \qquad Act_A^{(i)}[j] \stackrel{\text{def}}{=} \begin{cases} Act_A[j] & \text{if } j > i \\ Act_A[j] + 1 & \text{if } j = i \\ Act_A[j] + 1 & \text{if } j < i \end{cases}$$

$$n :: \left[\begin{array}{ll} \ell_0\colon & \left[\begin{array}{l}\textbf{when } \varphi_A^{(i)} \textbf{ do} \\ \quad act_A[i]\texttt{++}\end{array}\right] \\ \ell_1\colon & n.run() \\ \ell_2\colon & act_A[i]\texttt{--} \\ \ell_3\colon & \end{array}\right]$$

Figure 5.2: The protocol LIVE-P

Then, the condition that φ_A is preserved if a resource is granted to a process requesting access to run a method with annotation i is given by the formula $\varphi_A^{(i)}$ defined as follows:

$$\varphi_A^{(i)}[k] \stackrel{\text{def}}{=} Act_A^{(i)}[k] \leq T_A - k$$

$$\varphi_A^{(i)} \stackrel{\text{def}}{=} \bigwedge_k \varphi_A^{(i)}[k]$$

The formula $\varphi_A^{(i)}$ is the weakest precondition on a transition that preserves φ when granting a request to run a method with annotation value i.

Figure 5.2 shows the protocol schema LIVE-P that controls resource allocation for a method $n{:}A$ with annotation value $\alpha(n) = i$. LIVE-P is a schema, because the actual implementations of the test $\varphi_A^{(i)}$ and the operations $act_A[i]\texttt{++}$ and $act_A[i]\texttt{--}$ are left unspecified. Several implementations are possible, ranging from a brute force approach using tables to store $act_A[i]$ and repeated computations of $Act_A[i]$, to more efficient implementations presented later in this chapter. Any correct implementation of these operations guarantees absence of deadlock and also guarantees liveness, as we now prove.

5.3.2 Deadlock Avoidance

To show that LIVE-P guarantees absence of deadlock we first prove an auxiliary lemma.

Lemma 5.3.1. If φ_A holds and a clause $\varphi_A^{(i)}[j]$ does not hold, then there is at least one active process with annotation j.

Proof. From the fact that φ_A holds it follows that

$$\begin{aligned} Act_A[j] &\leq T_A - j \\ Act_A[j+1] &\leq T_A - (j+1) \;<\; T_A - j \end{aligned}$$

From the fact that $\varphi_A^{(i)}[j]$ does not hold, we infer

$$Act_A^{(i)}[j] = Act_A[j] + 1 > T_A - j$$

which, with $Act_A[j] \leq T_A - j$, gives

$$Act_A[j] = T_A - j$$

and thus, with $Act_A[j+1] < T - j$, we have

$$Act_A[j+1] < Act_A[j] \ .$$

Since $Act_A[j] = act_A[j] + Act_A[j+1]$ we conclude $act_A[j] > 0$, as desired. □

The following corollary holds immediately, by observing that if φ_A holds but $\varphi_A^{(i)}$ does not, there must be some offending clause for some $j \leq i$.

Corollary 5.3.2. If $\varphi_A^{(i)}$ is not satisfied, then there is some active process running a method with annotation at most i (i.e., $\sum_{j \leq i} act_A[j] \geq 1$).

We are now ready to show a version of the Annotation Theorem for LIVE-P , that establishes that this protocol provides deadlock avoidance.

Theorem 5.3.3 (Annotation Theorem for Live-P). Given a system $\mathcal{S}$ and an acyclic annotation, if every site uses LIVE-P to control allocations then all executions of $\mathcal{S}$ are deadlock-free.

Proof. We first observe that, in the absence of cyclic dependencies, the relation $\succ$ is a partial order on call graph methods. By contradiction, suppose that there is a reachable deadlock state. Let P be a process involved in the deadlock, blocked in a method $n{:}A$ that is minimal in $\succ$, and let i be the annotation value of n. We consider the two possible cases:

1. P is active. In this case a nested call to some descendant method m must be blocked, but then $n \succ m$ which contradicts the minimality of n.
2. P is waiting and $\varphi_A^{(i)}$ is false. By Corollary 5.3.2 there must be an active process running some method $n_1{:}A$ with annotation $\alpha(n_1) \leq i$. Since this process is active, it must be blocked in some subsequent remote invocation (to some method n_2). Then $n \dashrightarrow n_1 \rightarrow^+ n_2$, so $n \succ n_2$ again contradicting the minimality of n.

Hence, no deadlock is reachable. □

5.3.3 Liveness

We show now that any implementation of LIVE-P prevents starvation, provided the local schedulers are fair. In the presence of a fair scheduler, it is sufficient to show that every waiting process will eventually be enabled, that is, the entry condition of LIVE-P will eventually be satisfied. This guarantees that every process progresses and eventually receives all the resources it requests. In our model of computation this, in turn, implies that the process terminates. We first prove some auxiliary lemmas.

Lemma 5.3.4. $\varphi_A^{(k)}$ implies $\varphi_A^{(i)}$, for $k \geq i$.

Proof. First, $\varphi_A^{(k)}[j] \equiv \varphi_A^{(i)}[j]$ for all $j \geq k$ and for all $j < i$ since the formulas are syntactically identical. Now, take an arbitrary j within $i \leq j < k$. In this case,

$$\begin{array}{rcl} \varphi_A^{(k)}[j] & \equiv & \big(Act_A[j]+1 \leq T_A - j\big) \\ \varphi_A^{(i)}[j] & \equiv & \big(Act_A[j] \leq T_A - j\big), \end{array}$$

and if $\varphi_A^{(k)}[j]$ holds, so does $\varphi_A^{(i)}[j]$. □

Corollary 5.3.5 (Maximal Enabled Annotation). In every state σ there exists a value i within $0 \leq i \leq T_A$ such that all requests with annotation lower than i are enabled, and all those with annotation at least i are disabled. The annotation value i is the maximal enabled annotation in site A, and $i+1$ is the minimal illegal annotation.

The protocols BASIC-P and EFFICIENT-P also provide a notion of maximal enabled and minimal illegal annotation values. In the case of BASIC-P the minimal illegal annotation is t_A. For EFFICIENT-P, it is p_A (or 0 if $t_A = 0$). In general, these values are smaller than the one provided by LIVE-P , as the experimental results reported in Section 5.5 below confirm.

Theorem 5.3.6 (Liveness). Given a system $\mathcal{S}$ and an acyclic annotation, if LIVE-P is used as the allocation manager then in every run all waiting processes eventually become enabled.

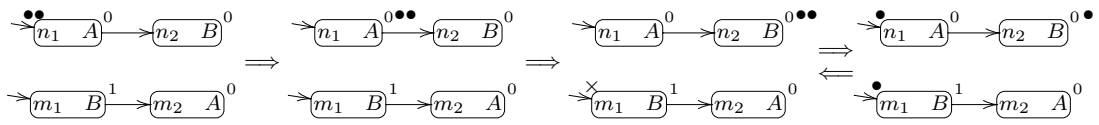
Proof. By contradiction, consider a run π with some starving process and let P starve in some method $n : A$ that is minimal in $\succ$ (among all methods with starving processes in π). Let i be $\alpha(n)$, the annotation value of n. After some prefix of π the system reaches a state σ after which P is continuously disabled, i.e., $\varphi_A^{(i)}$ does not hold. Let j be the highest annotation value of an offending clause $\varphi_A^{(i)}[j]$. Note that in any state after σ no request

for intermediate annotations k (with $j \leq k \leq i$) can be granted without P being enabled. This implies that $\varphi_A^{(i)}[j]$ remains the highest annotation value of an offending clause.

By Lemma 5.3.1, there must be some active process running a method with annotation j; all these processes terminate in the run π (by the minimality assumption on n). Let Q be the first such process to terminate. After Q releases its resources, all clauses $\varphi_A^{(i)}[k]$ for values $k \leq j$ are satisfied since the release of the resource decrements $Act_A[k]$. Therefore P becomes enabled right after Q terminates. □

The intuition behind the proof is that if P is disabled and j is the highest offending annotation, no request for an intermediate annotation value can be granted without P becoming enabled. This property is not satisfied by any of the previous protocols, as shown in Example 5.2.1.

Example 5.3.7. Let us revisit the system of Example 5.2.1, now using LIVE-P as the allocation manager. As soon as the first subprocess executing method n_2 terminates, P becomes enabled because granting the resource to P would lead to $act_A[0] = 1$ and $act_A[1] = 1$, which is a φ-state.



Consequently, either P's request is denied infinitely often, which cannot happen if the scheduler is fair, or P eventually receives its demanded resource. ⌟

5.4 Implementation

LIVE-P can be implemented in different ways to accommodate different requirements on processing time and space available. While BASIC-P and EFFICIENT-P only require simple checks and updates on one (resp. two) variables, LIVE-P must, in principle, maintain the tables $act_A[\cdot]$ and $Act_A[\cdot]$, which requires space proportional to the largest annotation value in the annotated call graph. We describe now some strategies for implementing the three components of LIVE-P: the allocation manager, the scheduler, and the controller that combines them. To justify the, rather complex, efficient implementation strategies for the allocation manager, we first present a simple implementation that only checks and updates

$$n :: \left[\begin{array}{ll} \ell_0 : & \left[\begin{array}{c}\textbf{when } 0 < t_A \ \wedge \ Act_A[i] < (T_A - i) \textbf{ do} \\ t_A\texttt{--} \ ; \ act_A[i]\texttt{++}\end{array}\right] \\ \ell_1 : & n.run() \\ \ell_2 : & t_A\texttt{++} \ ; \ act_A[i]\texttt{--} \\ \ell_3 : & \end{array}\right]$$

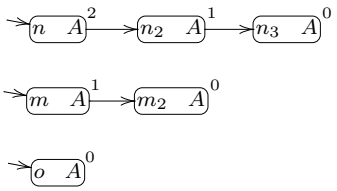
Figure 5.3: The protocol BAD-P

the clause corresponding to the annotation of the requesting process, and show that this implementation is not correct.

5.4.1 Allocation Manager

A Tempting (but Incorrect) Implementation A tempting implementation of LIVE-P consists of only checking the clause that corresponds to the annotation value of the method requested, resulting in the protocol BAD-P, shown in Figure 5.3. Unfortunately, the protocol BAD-P is not a correct implementation of LIVE-P, and it compromises deadlock freedom.

Example 5.4.1. Consider a scenario with $T_A = 3$ and the following call graph:



Let $s : omm\overline{o}n$ be an allocation sequence for site A. Table 5.1 shows the values of $act_A[\cdot]$ and $Act_A[\cdot]$ after each allocation or deallocation. All requests satisfy the conditions of the entry

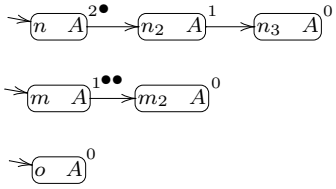
Sequence	$act_A[0]$	$act_A[1]$	$act_A[2]$	$Act_A[0]$	$Act_A[1]$	$Act_A[2]$
ϵ	0	0	0	0	0	0
o	1	0	0	1	0	0
om	1	1	0	2	1	0
omm	1	2	0	3	2	0
$omm\overline{o}$	0	2	0	2	2	0
$omm\overline{o}n$	0	2	1	3	3	1

Table 5.1: A bad sequence for BAD-P

section of protocol BAD-P, so they are immediately granted. The last row in Table 5.1, though, corresponds to a state that does not satisfy the invariant clause $\varphi_A[1]$:

$$Act_A[1] = 3 \qquad \not\leq \qquad T_A - 1 = 3 - 1 = 2 \; .$$

This illegal state is reached after a request for a method with annotation 2 (n in the call graph) is allowed. Granting this request causes a violation of $\varphi_A[1]$, but the violation is not detected by BAD-P because it is not produced in the clause checked: $\varphi_A[2]$. Moreover, all previous requests for annotation value 1 were granted rightfully. The illegal state reached is:



At this state all resources are used, $t_A = 0$, and all processes are involved in a deadlock. ⊿

This example seems to indicate that a linear number $O(T_A)$ of tests and operations are needed to perform correctly each request and release of LIVE-P. However, a more efficient implementation is possible.

An Efficient and Correct Implementation The key idea of this efficient implementation is the use of a new data-structure, called an *active tree*, that stores the number of active processes for each annotation j (denoted as $act_A[j]$ above) with efficient operations of:

1. inserting a process,
2. removing a process, and
3. obtaining the highest annotation of a request that can be safely granted without violating φ. By Corollary 5.3.5 this value is unique.

We describe here how to implement this data-structure using a binary search tree with the annotation as the key, and where each node also stores the number of active processes with that annotation, in a field named *count*. This data-structure can be maintained:

- in $O(T_A)$ space and $O(log\ T_A)$ time per insertion and removal using a complete binary tree, or

- in $O(L)$ space and $O(log\ L)$ time per insertion and removal using a balanced tree (for example a Red-Black tree), where L is the number of different annotations among those methods with some active processes. This parameter is called the *diversity load*.

When a request to run a method with annotation i is granted, if a node with key i exists in the tree, its *count* field is incremented; otherwise a new node with key i and *count* 1 is added to the tree.

In order to obtain an efficient calculation of the maximal legal annotation, the search tree is augmented with extra information in each node, based on the following observation. If the active processes were linearly ordered according to the annotation of the method they are executing, a violation of φ_A would be witnessed by a process with annotation i being located further than $T_A - i$ positions from the end of the list. Similarly, the value of the minimal illegal annotation corresponds to the process with smallest i that is precisely $T_A - i$ positions to the end of the list. We maintain enough information in each method to retrieve the smallest such offending annotation in time proportional in the height of the tree. In the following description we use *tree*(x) to denote the (sub)tree rooted at node x, and *left*(x) and *right*(x) for the left and right subtrees resp. If *foo* is a field, the instance of *foo* at node x is represented by $x.foo$. Each node in the tree stores the following fields:

1. *key*: the annotation of the processes that the node describes.
2. *count*: the number of active processes with that annotation.
3. *size*: the total number of processes in *tree*(x), including all the $x.count$.
4. *larger*: the maximum number of processes with annotation larger than the largest key in *tree*(x), that could be added (or that exist in the super-tree containing *tree*(x)) without causing a φ violation with respect to any of the nodes in *tree*(x).
5. *larger_me*: the maximum number of processes with annotation larger than the largest key in *tree*(x), that could be added (or that exist in the super-tree containing *tree*(x)) without causing a φ violation with respect to the node x itself.
6. *larger_left*: the maximum number of processes with annotation larger than the largest key in *tree*(x), that could be added (or that exist in the super-tree containing *tree*(x)) without causing a φ violation in *left*(x). Note that $x.count$ and all the processes

described in $right(x)$ are already present and higher than any annotation stored in $left(x)$.

7. *larger_right*: the maximum number of processes with annotation larger than or equal to the largest key in $tree(x)$, that could be added (or that exist in the super-tree containing $tree(x)$) without causing a φ violation with respect to $right(x)$.

It is well-known (see for example [CLRS01], Theorem 15.1) that an augmented Red-Black tree can be maintained, with the regular operations of insertion and removal still in $O(log\ n)$, if all fields can be computed from simpler fields of the node and all the fields of the children nodes. This augmentation result obviously holds for complete binary trees as well. Our augmentations satisfy this property, since:

1. *key* and *count* are primitive fields, not depending on other fields in any node in the tree.

2. *size* can be computed from the primitive *count* field and the *size* fields of the children:

$$x.size = left(x).size + right(x).size + x.count.$$

3. *larger* is just the minimum of three other augmentation fields:

$$x.larger = min(x.larger\_me, x.larger\_left, x.larger\_right).$$

4. *larger_me* can be computed using

$$x.larger\_me = T_A - x.key - (x.count + right(x).size).$$

This holds because if there are $x.larger\_me + right(x).size$ active processes with annotation higher than $x.key$, then the total number of processes with annotation $x.key$ or higher is

$$Act_A[x.key] = x.count + x.larger\_me + right(x).size,$$

and then $Act_A[x.key]$ would be $T_A - x.key$. This is the largest value that satisfies φ.

5. *larger_right* is directly the largest value of the right subtree:

$$x.larger\_right = right(x).larger.$$

```
1: MaxLegal(x, extra) :
2: if (x.larger_left − extra = 0) then
3:     return MaxLegal(left(x), extra + right(x).size + x.count)
4: else if (x.larger_me − extra = 0) then
5:     return x.key − 1
6: else if (x.larger_right − extra = 0) then
7:     return MaxLegal(right(x), extra)
8: else
9:     return T_A − 1
10: end if
```

Figure 5.4: MaxLegal: computes the maximal legal annotation

6. Finally, *larger_left* can be computed from the left subtree by considering the processes already present in the root and right subtrees:

$$x.\mathit{larger\_left} = \mathit{left}(x).\mathit{larger} - (\mathit{right}(x).\mathit{size} + x.\mathit{count}).$$

In all the definitions above, if the left (resp. right) subtree is missing, then $\mathit{left}(x).\mathit{size}$ is 0, and $\mathit{left}(x).\mathit{larger} = \infty$. A tree stores a legal configuration of active processes if the value of $\mathit{root.larger}$ is non-negative.

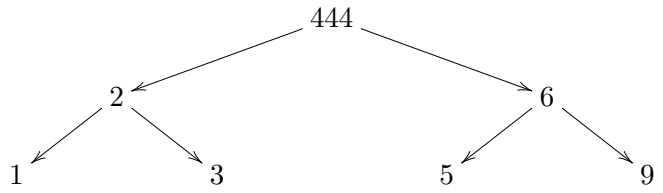
Finally, Figure 5.4 shows the algorithm that calculates the maximum value of a legal insertion. The initial call is MaxLegal$(\mathit{root}, 0)$. The algorithm traverses the tree seeking for the leftmost occurrence of a method x satisfying the following condition:

$$(x.\mathit{larger\_me} - \mathit{extra}) = 0 \tag{5.1}$$

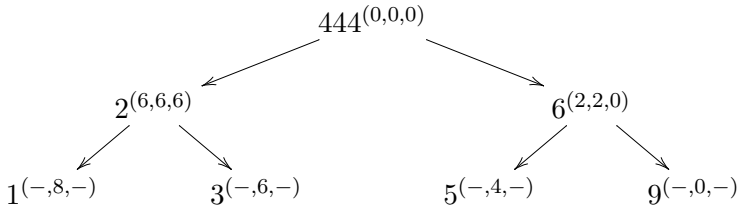
Since the parameter *extra* passes the number of active processes with annotation actually larger than $x.\mathit{key}$ in the whole tree, condition (5.1) captures precisely whether a new insertion of a value larger than or equal to $x.\mathit{key}$ would cause a $\varphi_A[x.\mathit{key}]$ violation. This search can be performed in a number of steps proportional to the height of the tree, which gives a complexity of $O(\mathit{log}\ L)$ where L is the size of the tree (the diversity load) with the use of balanced trees, and $O(\mathit{log}\ T_A)$ with the complete tree.

Example 5.4.2. Consider a site A with $T_A = 10$ resources, and the following tree, which

is a possible active tree representing the set of active processes $\{1, 2, 3, 4, 4, 4, 5, 6, 9\}$:



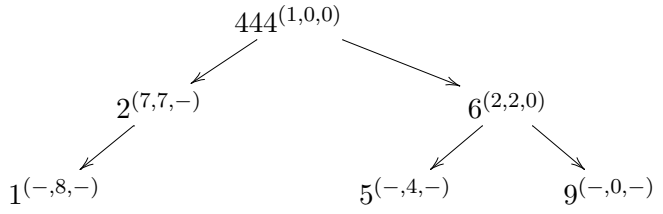
The values of $(\mathit{larger\_left}, \mathit{larger\_me}, \mathit{larger\_right})$ for each of the nodes are:



MaxLegal$(\mathit{root}, 0)$ returns 0, after performing the sequence of calls:

$$\begin{aligned}\textsc{MaxLegal}(444^{(0,0,0)}, 0) &\mapsto \textsc{MaxLegal}(2^{(6,6,6)}, 6)\\ &\mapsto \textsc{MaxLegal}(1^{(-,8,-)}, 8)\\ &\mapsto 1 - 1 = 0\end{aligned}$$

The maximum annotation of a method with an enabled entry section is 0 since it can be legally inserted, and any insertion of 1 or higher would cause a violation in the node $1^{(-,8,-)}$. Suppose that the process with annotation 3 releases its resource, and that the resulting tree is:



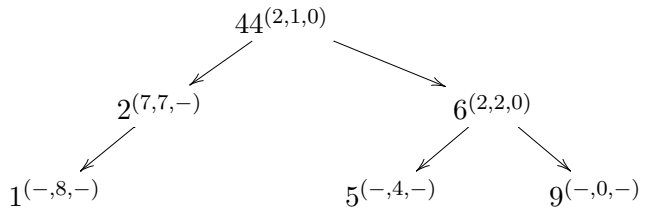
In this case the maximal legal annotation is 3 since:

$$\textsc{MaxLegal}(444^{(1,0,0)}, 0) \mapsto 4 - 1 = 3$$

data-structure	*time*	*space*
Array	$O(T_A)$	$O(T_A)$
CompleteBinaryTree	$O(log\ T_A)$	$O(T_A)$
Red-Black Tree	$O(log\ L)$	$O(L)$

Table 5.2: Asymptotic running times of implementations of LIVE-P

Finally, if one of the processes with annotation 4 releases its resource, the resulting tree is:



The maximal annotation is 8 as indicated by:

$$\begin{aligned}\textsc{MaxLegal}(44^{(2,1,0)}, 0) &\mapsto \textsc{MaxLegal}(6^{(2,2,0)}, 0)\\ &\mapsto \textsc{MaxLegal}(9^{(-,0,-)}, 0)\\ &\mapsto 9 - 1 = 8\end{aligned}$$

⌟

The asymptotic running times of the three methods presented to implement LIVE-P is summarized in Table 5.2. Section 5.5 reports experimental results comparing the running times of these implementations.

5.4.2 Implementation of a Fair Scheduler

We sketch how to implement a local fair scheduler based on an *oldest process first* policy. An earliest deadline first (EDF) policy, which is also strongly fair, could be accomplished similarly. The implementation is based on a data structure, called a *waiting tree*, that can perform three operations:

1. insert a process,
2. remove a process,
3. obtain the oldest process with a certain annotation or smaller.

Similarly to the discussion of the previous section, an efficient waiting tree can be implemented using a binary search tree, with the annotation as the key but this time including a priority queue to store the waiting processes running methods with the same annotation. In a waiting tree each node is also augmented with the oldest process present in the left and right subtrees. These augmentations only depend on the values of the corresponding children nodes, so its maintenance is efficient. Using a Red-Black tree, this data-type can be maintained in $O(log\ w + log\ m)$, where w is the number of different annotations with some waiting process, and m is the maximum size of any priority queue (maximum number of waiting processes for the worst annotation). Using a complete tree a running time of $O(log\ T_A + log\ m)$ is obtained.

5.4.3 Implementation of the Controller

Finally, the controller can be built by combining the active tree that implements the deadlock avoidance algorithm with the waiting tree that implements the scheduler, as follows:

- **allocation request**: check whether the annotation of the requesting process is at most MaxLegal($root, 0$).
  - If the check succeeds, grant the resource and insert the process in the active tree.
  - If the check fails, insert the process in the waiting tree.
- **resource release**: remove the process from the active tree, and recalculate $k =$ MaxLegal($root, 0$). Obtain the oldest process P with annotation k or smaller from the waiting tree (if any); extract it, and perform an allocation request. This allocation is guaranteed to be successful.

5.5 Experimental Results

The experimental results reported in this section were obtained using a direct implementation of the data structures in C++, executed on a 1GHz Pentium III Xeon with 1GB of RAM. The experiments consist of a sequence of allocations in a single site, for annotation values generated uniformly at random.

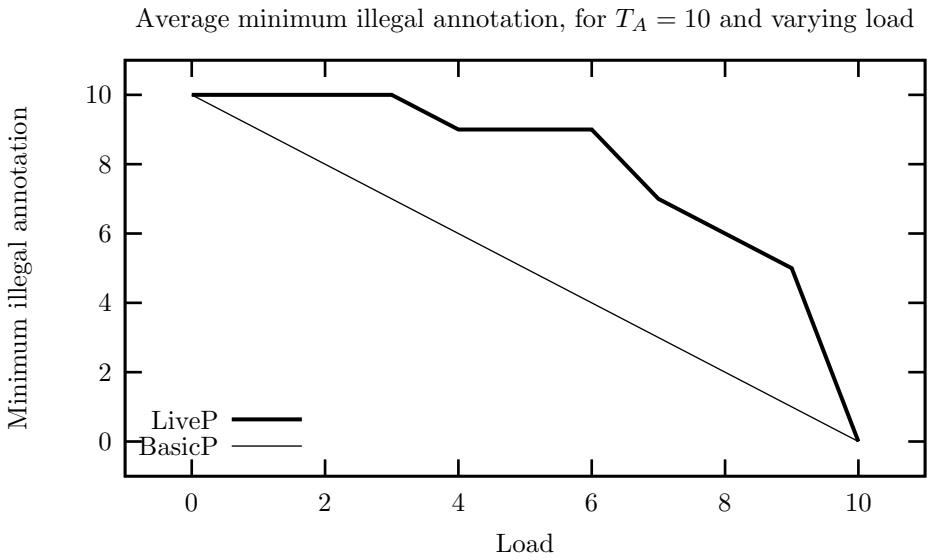


Figure 5.5: Average Minimal Illegal annotation of Basic-P and Live-P

Experiment #1: Level of Concurrency of Live-P vs Basic-P

All the deadlock avoidance protocols presented so far have the property that if a request for an annotation value is enabled then every request for a smaller annotation value is also enabled. Similarly, if a request is denied, all requests for higher values must also be denied. This fact gives rise to the notions of maximal legal annotation and minimal illegal annotation. In the case of Live-P, these values are given by Corollary 5.3.5. For Basic-P the minimal illegal annotation is directly given by the enabling condition $\alpha(n) < t_A$. This enabling condition can, in turn, be related to load (the number of active processes) and total number of resources by Load $= T_A - t_A$.

In this experiment we measure the minimal illegal annotation as a function of the load. For a given load, we create a configuration by selecting active processes with annotations chosen uniformly at random, ensuring that the state is legal, i.e., it is an φ-state. Then then we compute the minimal illegal annotation for each protocol. Figure 5.5 shows the (average) minimal illegal annotation allowed by Basic-P and Live-P as a function of the load. The curve indicates that Live-P allows more concurrency than Basic-P, by granting more requests. The two curves coincide when all the resources are available (minimum load) and when no resource is available (maximum load). In the former case the minimal illegal annotation is T_A, so all requests are allowed. In the latter, the minimal illegal annotation is

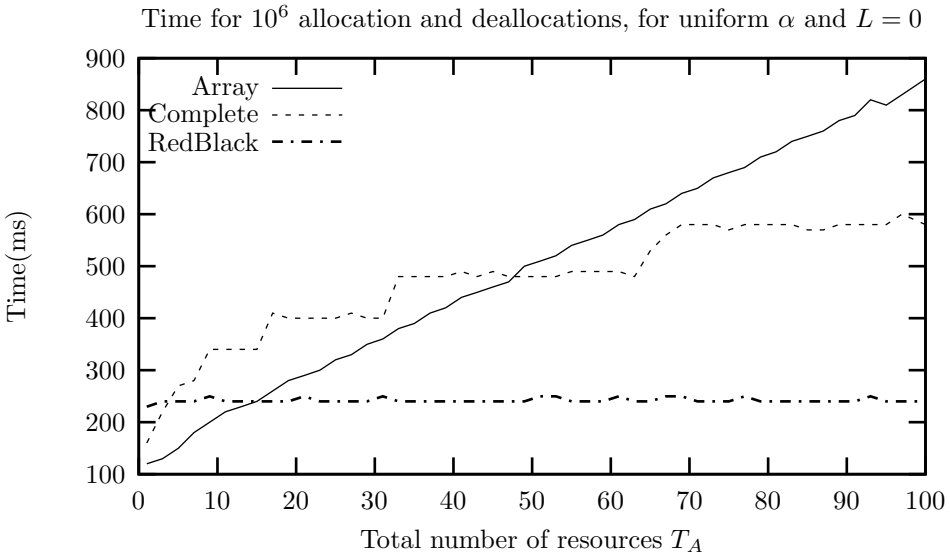


Figure 5.6: Comparing implementations of LIVE-P, for $L = 0$

0, so no request is permitted. With an intermediate load, BASIC-P allows only the values according to the linear law dictated by the enabling condition, while LIVE-P statistically allows more concurrency.

Experiment #2: Comparing Implementations of Live-P

In this group of experiments we compare the running times of the three different implementations of LIVE-P presented in Section 5.4. The "Array" implementation stores all the values of $act_A[\cdot]$ and $act_A[\cdot]$ in a table, and traverses the table for each operation. The "Complete" implementation uses an augmented complete binary search tree to implement the active tree. The "RedBlack" algorithm implements the active tree using an augmented Red-Black tree.

The experiments consist of repeatedly performing one million allocation operations (each followed by a corresponding deallocation) under different circumstances, and measuring the time needed to execute this sequence of operations. The annotation of the request is a legal annotation picked uniformly at random. We consider three different scenarios:

1. No load ($L = 0$): Figure 5.6 reports the execution times for the sequence of allocations in a state with no load, for different values of T_A (the total number of resources handled). Since "RedBlack" does not depend on the size of the resource pool, the

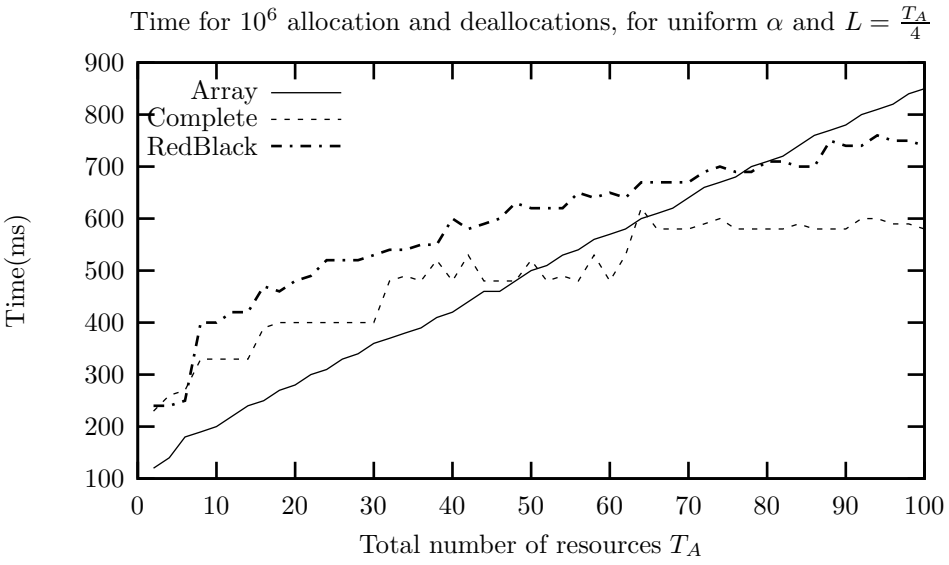


Figure 5.7: Comparing implementations of LIVE-P, for $L = \frac{T_A}{4}$

running time is constant. For "Complete", the time reported grows logarithmically, since the height of the complete binary tree grows logarithmically with the size of the resource pool. Finally, "Array" reports a linear growth.

2. Some load ($L = \frac{T_A}{4}$): in this experiment the allocations are performed in a state with a load of 25%, for different values of T_A. Figure 5.7 shows that, in this case, the implementations that use active trees report a logarithmic growth, while "Array" reports a linear growth. The growth of "ReadBlack" is logarithmic because the number of operations in this implementation depends (logarithmically) on the load.

3. in this experiment we fix the size of the resource pool ($T_A = 63$) and perform the allocations under different diversity loads. Figure 5.8 shows the dependency of the Red-Black implementation with the diversity load. While the "Array" and "Complete" implementations do only depend on the total size of the resource pool, and not on the number or kind of active processes, the time required for an operation in the "RedBlack" implementation grows logarithmically with the diversity load.

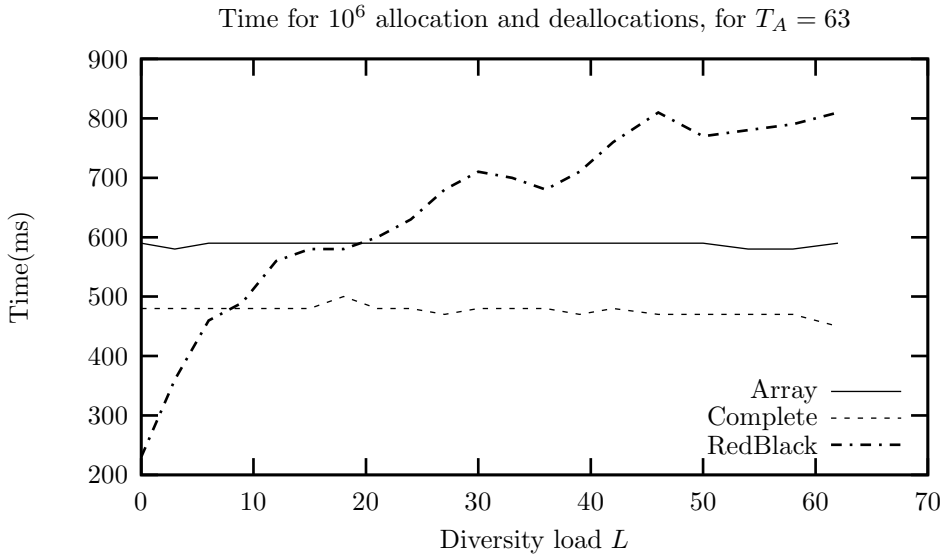


Figure 5.8: Comparing implementations of LIVE-P, for $T_A = 63$

5.6 Summary

This chapter has shown that not only deadlock avoidance, but also individual liveness, is enforceable in a completely distributed fashion, without communication. The protocols investigated in previous chapters allow runs in which some processes starve. In this chapter we have introduced the protocol LIVE-P and pro ved that it prevents starvation, provided that the local scheduler arbitrates ties between competing processes fairly: no enabled process loses forever. Liveness is accomplished by granting precisely those allocations that preserve the invariant φ that guarantees deadlock avoidance, thereby allowing more concurrency than all previous protocols. The price to pay with LIVE-P is that the operations of allocation and deallocation are no longer implementable in constant time. This chapter also presents an implementation that requires a logarithmic number of operations, in terms of the size of the initial resource pool.

Chapter 6

Proving New Protocols

This chapter presents a uniform framework that relates all the protocols introduced in the previous chapters. All protocols are seen as safe approximations of the invariant φ, illustrating a trade-off between the complexity of the implementation and precision of the approximation. We show that, somewhat surprisingly, even though the more precise protocols allow more allocation sequences, the reachable set of states of all the protocols is the same. This enables an important design principle: to prove that a new protocol provides deadlock avoidance, it is sufficient to show that none of its allocation decisions are ever more conservative than Basic-P or more liberal than Live-P . The first restriction guarantees that enough progress is made. The second, that deadlock freedom is not compromised by allowing too much concurrency.

6.1 A Family of Local Protocols

In this section we restate the protocols presented in previous chapters in a uniform framework. This framework will allow comparing the protocols based on their enabling conditions, and prove that all the protocols have the same reachable state spaces. The main idea in the development of the framework is to redefine the enabling condition of as an strengthening of the invariant φ, which is the essence of deadlock avoidance for all these protocols. Let us recall that $\varphi_A[k]$, the clause of φ for site A and annotation value k, is $Act_A[k] \leq T_A - k$. Even though Basic-P makes φ invariant, the enabling condition of the protocol Basic-P was not stated directly in terms of φ. The enabling condition of Basic-P, for a method $n{:}A$ with annotation $i = \alpha(n)$ is:

$$i < t_A \qquad (En_n^{\text{BASIC-P}})$$

Now, $Act_A[0] = T_A - t_A$ accounts for the total number of resources being used in A. Reordering terms, $t_A = T_A - Act_A[0]$. This expressions allows to re-write the enabling condition of BASIC-P as:

$$i < T_A - Act_A[0]$$

or, equivalently,

$$Act_A[0] < T_A - i\ . \qquad (En_n^{\text{BASIC-P}})$$

Note that since every successful allocation will increment $Act_A[0]$, the value of $Act_A^{(i)}[0]$ is always $Act_A[0]+1$, independently of i. This way, the enabling condition of BASIC-P is also equivalent to:

$$Act_A^{(i)}[0] \leq T_A - i \qquad (En_n^{\text{BASIC-P}})$$

or, equivalently,

$$Act_A[0] + 1 \leq T_A - i\ . \qquad (En_n^{\text{BASIC-P}})$$

This expression is almost identical to $\varphi_A^{(i)}[k]$, except that the left hand side considers all the resources allocated ($Act_A[0]$) instead of just the resources with annotation value k or higher ($Act_A[k]$). As we will see, this expression is stronger than $\varphi_A^{(i)}$. In other words, the enabling condition of BASIC-P implies that of LIVE-P .

Given $k \leq i$ we now define the k-th strengthening formula for a request to run a method with annotation value i as:

$$\begin{array}{rcccl} \chi_A^{(i)}[k] & \stackrel{\text{def}}{=} & Act_A^{(i)}[k] & \leq & T_A - i \\ & \equiv & Act_A[k] + 1 & \leq & T_A - i \end{array}$$

It is easy to see that the following holds for all $k \leq j \leq i$,

$$\chi_A^{(i)}[k] \rightarrow \varphi_A^{(i)}[j]$$

and therefore

$$\chi_A^{(i)}[k] \rightarrow \bigwedge_{k \leq j \leq i} \varphi_A^{(i)}[j].$$

$$n :: \left[\begin{array}{l} \ell_0 : \left[\begin{array}{c}\textbf{when } \chi_A^{(i)}[0] \textbf{ do}\\ act_A[i]\texttt{++}\end{array}\right] \\ \ell_1 : n.run() \\ \ell_2 : act_A[i]\texttt{--} \\ \ell_3 : \end{array}\right]$$

Figure 6.1: The protocol BASIC-P, restated using strengthenings

Also, if φ_A holds before a request for annotation value i, then $\varphi_A^{(i)}[j]$ also holds for all $i \geq j$, since the formulas for $\varphi_A^{(i)}[j]$ and $\varphi_A[j]$ are identical in this case. Hence:

$$\chi_A^{(i)}[k] \rightarrow \bigwedge_{k \leq j} \varphi_A^{(i)}[j]. \tag{6.1}$$

Finally, if $\varphi_A^{(i)}[j]$ is satisfied for all values less than k, and $\chi_A^{(i)}[k]$ also holds, then $\varphi_A[i]$ can be concluded:

$$\begin{aligned} (\bigwedge_{j<k} \varphi_A^{(i)}[j]) \wedge \chi_A^{(i)}[k] &\rightarrow (\bigwedge_{j<k} \varphi_A^{(i)}[j]) \wedge (\bigwedge_{j \geq k} \varphi_A^{(i)}[j]) \\ &\leftrightarrow \bigwedge \varphi_A^{(i)}[j] \\ &\leftrightarrow \varphi_A^{(i)}. \end{aligned}$$

Therefore, if a protocol enforces that for some k, both $\bigwedge_{j<k} \varphi_A^{(i)}[j]$ and the k-strengthening $\chi_A^{(i)}[k]$ hold, then the protocol preserves φ_A as an invariant.

In general, the lower the value of the strengthening point k, the less computation is

$$n :: \left[\begin{array}{l} \ell_0 : \left[\begin{array}{c}\textbf{when } \varphi_A^{(i)}[0] \wedge \chi_A^{(i)}[1] \textbf{ do}\\ act_A[i]\texttt{++}\end{array}\right] \\ \ell_1 : n.run() \\ \ell_2 : act_A[i]\texttt{--} \\ \ell_3 : \end{array}\right]$$

Figure 6.2: The protocol EFFICIENT-P, restated using strengthenings

$$
n :: \left[\begin{array}{ll}
\ell_0 : & \left[\begin{array}{l}\textbf{when } (\bigwedge_{j<k} \varphi_A^{(i)}[j]) \wedge \chi_A^{(i)}[k] \textbf{ do} \\ \quad act_A[i]\texttt{++}\end{array}\right] \\
\ell_1 : & n.run() \\
\ell_2 : & act_A[i]\texttt{--} \\
\ell + 3 : &
\end{array}\right]
$$

Figure 6.3: The protocol k-EFFICIENT-P

needed to check the enabling condition (the number of comparisons is reduced) but the less precise the enabling condition becomes. In the case of $k = 0$ the strengthening is $\chi_A^{(i)}[0]$, and the protocol obtained (see Figure 6.1) is equivalent to BASIC-P . Note that this protocol is *logically* equivalent to BASIC-P: the result of the enabling condition, and the effect of the input and output actions on future tests are the same. The implementation of BASIC-P introduced in Chapter 3 uses a single counter t_A, while in this restated version, several counters are used: $act_A[i]$ and $Act_A[0]$. However, the effect on $Act_A[0]$ of the increments and decrements of $act_A[i]$ are independent of i. Therefore, these actions can be implemented instead as $Act_A[0]$++ and $Act_A[0]$--. Similarly, with a strengthening point of $k = 2$ we obtain a protocol equivalent to EFFICIENT-P , shown in Figure 6.2.

The general form of the family of protocols can now be given as k-EFFICIENT-P , shown in Figure 6.3. It covers the full spectrum of protocols: BASIC-P—which is equivalent to 0-EFFICIENT-P—lies at one end, and LIVE-P—which is equal to $(T_A - 1)$-EFFICIENT-P—lies at the other end of the spectrum. The protocols k-EFFICIENT-P can be implemented in several ways. The simplest implementation needs space $O(k \ log \ T_A)$ to store k counters and requires $O(k)$ operations per allocation decision. A more sophisticated implementation using an *active tree* data-structure—introduced in Section 5.4—still needs $O(k \ log \ T_A)$ space, but requires only $O(log \ k)$ operations per allocation decision.

Experiment #3: Level of Concurrency of k-Efficient-P for varying k

In this experiment we compare the minimum illegal annotation for different instances of k-EFFICIENT-P, as a function of the load. The load was generated by picking active processes with annotations uniformly at random, ensuring that state generated is a φ-state. Figure 6.4 shows how for any given load k-EFFICIENT-P allows more requests as k grows. This experiment generalizes the experimental results of Section 5.5 where only BASIC-P

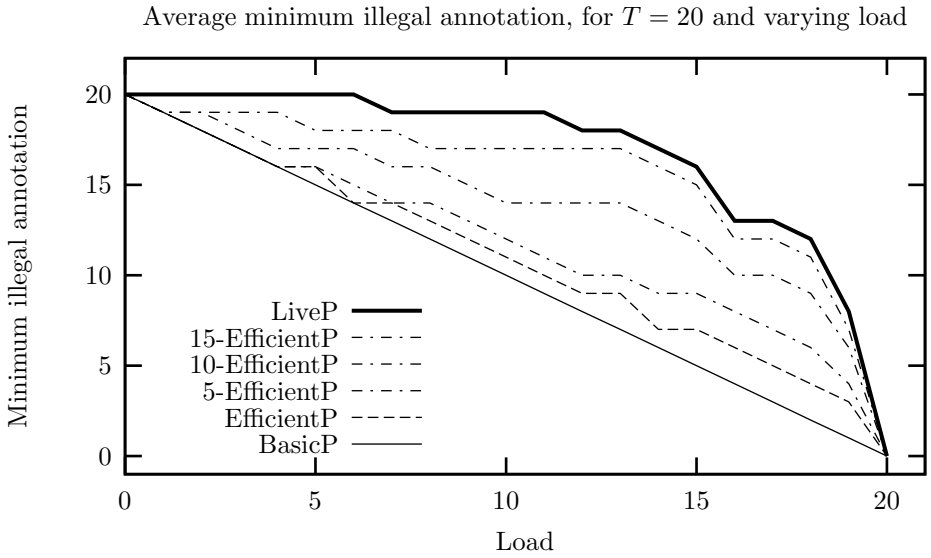


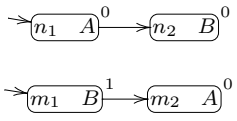
Figure 6.4: Average minimum illegal annotation of k-EFFICIENT-P for $T = 20$

and LIVE-P (the extreme cases) were compared.

6.2 Allocation Sequences

This section continues the study of allocation sequences, introduced in Section 2.5, to compare the runs that each protocol allows. Recall that a sequence s of allocations and deallocation is accepted by a protocol P if $P(s) \neq \bot$. We use $\mathcal{L}(P)$ to denote the set of sequences accepted by a protocol, and use $P \sqsubseteq Q$ for the partial order defined by language inclusion $\mathcal{L}(P) \subseteq \mathcal{L}(Q)$.

Example 6.2.1. Reconsider the system of Examples 2.4.4 and 3.5.5. Let BASIC-P be used as the resource allocation protocol, with $T_A = T_B = 2$ and the following annotated call graph:



The allocation sequence that leads to a deadlock if no resource allocation protocol is used is $s : n_1 n_1 m_1 m_1 m_1$. Even though $n_1 n_1 m_1$ is in $\mathcal{L}$(BASIC-P), the enabling condition of m_1 becomes disabled, so BASIC-P$(s) = \bot$ and $s \notin \mathcal{L}$(BASIC-P). ⊿

We order the protocols according to the allocation sequences they allow. First, we prove

an auxiliary lemma, that relates enabling conditions and protocol languages.

Lemma 6.2.2. The following are equivalent:

(i) $\mathcal{L}(P) \subseteq \mathcal{L}(Q)$.

(ii) For all strings s and methods n, if $En_n^P(P(s))$ then $En_n^Q(Q(s))$.

Proof. We prove each implication separately:

- Assume $\mathcal{L}(P) \subseteq \mathcal{L}(Q)$, and let s and n be such that $En_n^P(P(s))$. Since $s \in \mathcal{L}(P)$ then $s \in \mathcal{L}(Q)$. Moreover, $s \cdot n \in \mathcal{L}(P)$ and then $s \cdot n \in \mathcal{L}(Q)$. Hence, $En_n^Q(Q(s))$.
- Assume now (ii). We reason by induction on strings:
  - First, both $\epsilon \in \mathcal{L}(P)$ and $\epsilon \in \mathcal{L}(Q)$.
  - Let $s \cdot n \in \mathcal{L}(P)$. Then $En_n^P(P(s))$, so also $En_n^Q(Q(s))$. Hence, $s \cdot n \in \mathcal{L}(Q)$.
  - Let $s \cdot \overline{n} \in \mathcal{L}(P)$. This implies $s \in \mathcal{L}(P)$ and by the inductive hypothesis $s \in \mathcal{L}(Q)$. Then $s \cdot \overline{n} \in \mathcal{L}(Q)$, as desired.

Therefore (i) and (ii) are equivalent. □

Let P, Q be any two of Basic-P , Efficient-P , k-Efficient-P and Live-P . We showed in Section 6.1 that the entry and exit actions are logically identical for all these protocols, and that the enabling conditions can be restated in terms of the $Act_A[\cdot]$ and $act_A[\cdot]$. Hence, these protocols operate on the same state space of protocol variables. Consequently, if s is in the language of both P and Q then the states reached are the same, i.e., $P(s) = Q(s)$. It follows, by Lemma 6.2.2, that if for all global states σ, $En_n^P(\sigma)$ implies $En_n^Q(\sigma)$, then $\mathcal{L}(P) \subseteq \mathcal{L}(Q)$. This allows the reasoning in terms of enabling conditions of the protocols at an arbitrary state.

Lemma 6.2.3. If j-Efficient-P allows an allocation then k-Efficient-P also allows the allocation, provided $j \leq k$.

Proof. Let $j \leq k$. It follows from the definition that $\chi_A^{(i)}[j]$ implies $\chi_A^{(i)}[k]$. Moreover, by (6.1), $\chi_A^{(i)}[j]$ implies $\bigwedge_{j \leq l \leq k} \varphi_A^{(i)}[l]$. Consequently,

$$\underbrace{\bigwedge_{l<j} \varphi_A^{(i)}[l] \wedge \chi_A^{(i)}[j]}_{En_n^{j\text{-Efficient-P}}} \rightarrow \underbrace{\bigwedge_{l<k} \varphi_A^{(i)}[l] \wedge \chi_A^{(i)}[k]}_{En_n^{k\text{-Efficient-P}}}$$

Therefore if j-Efficient-P allows a request so does k-Efficient-P. □

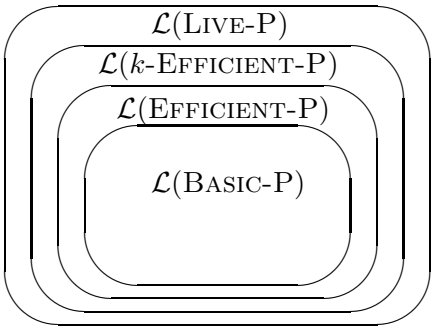


Figure 6.5: Sequences allowed by the deadlocks avoidance protocols

Lemma 6.2.3 states that the enabling condition of k-Efficient-P becomes weaker as k grows, that is, the enabling condition of Basic-P is stronger than that of Efficient-P, which in turn is stronger than k-Efficient-P, which is stronger than Live-P. An immediate consequence of Lemma 6.2.3 is:

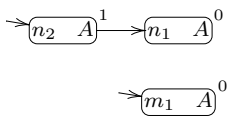
$$\text{Basic-P} \sqsubseteq \text{Efficient-P} \sqsubseteq \ldots \sqsubseteq k\text{-Efficient-P} \sqsubseteq \ldots \sqsubseteq \text{Live-P}$$

The following example shows that these language containments are strict:

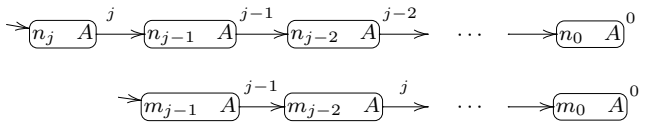
$$\text{Basic-P} \not\supseteq \text{Efficient-P} \not\supseteq \ldots \not\supseteq k\text{-Efficient-P} \not\supseteq \ldots \not\supseteq \text{Live-P}$$

which is depicted in Figure 6.5.

Example 6.2.4. Consider the following call graph, with initial resources $T_A = 2$.



The string $m_1 n_2$ is accepted by Efficient-P but not by Basic-P. This system can be generalized to show that there is a string accepted by k-Efficient-P but not by j-Efficient-P (for $j < k$). Consider the following annotated call graph, with initial resources $T_A = j + 1$.



The string $m_{j-1}n_j$ is accepted by k-EFFICIENT-P, but is not accepted by j-EFFICIENT-P.⌟

6.3 Reachable State Spaces

The reachable state space of a protocol P, denoted by $\mathcal{S}(P)$, is the set of global states that P can reach following some admissible allocation sequence. Clearly, for two protocols P and Q, if their actions are equivalent and $P \sqsubseteq Q$ then every state reachable by P is also reachable by Q. Indeed any allocation string that reaches a state for P also reaches that same state for Q.

Lemma 6.3.1. For every two protocols P and Q with the same entry and exit actions, if $P \sqsubseteq Q$ then $\mathcal{S}(P) \subseteq \mathcal{S}(Q)$.

Consequently,

$$\mathcal{S}(\text{BASIC-P}) \subseteq \mathcal{S}(\text{EFFICIENT-P}) \subseteq \ldots \subseteq \mathcal{S}(k\text{-EFFICIENT-P}) \subseteq \ldots \subseteq \mathcal{S}(\text{LIVE-P})$$

Let $\mathcal{S}(\varphi)$ describe the set of φ-states that are reachable by some admissible allocation string. These are φ-states that correspond to some run, if allowed by the allocation protocol. In the rest of this section we show that the above containment relation collapses into equalities by proving

$$\mathcal{S}(\text{BASIC-P}) \;=\; \mathcal{S}(\text{LIVE-P}) \;=\; \mathcal{S}(\varphi)$$

The proof relies on the existence of a *preference order* on the nodes of the annotated call graph, such that, if allocations are made following this order, then every allocation request that succeeds in LIVE-P also succeeds in BASIC-P.

6.3.1 Preference Orders

A *preference order* of an annotated call graph is an order on the methods such that, if all allocations in a given admissible string are performed following that order—instead of the order in the string—then (1) the sequence obtained is also admissible, and (2) higher annotations for each site are visited first. This will allow us to show that BASIC-P can reach all $\mathcal{S}(\varphi)$.

Given a call graph, a total order $>$ on its methods is called topological if it respects the descendant relation, that is, if for every pair of methods n and m, if $n \to m$ then $n > m$.

Analogously, we say that an order $>$ respects an annotation α if for every pair of methods n and m residing in the same site, if $\alpha(n) > \alpha(m)$ then $n > m$. A total order that is topological and respects annotations is called a preference order.

Lemma 6.3.2. Every acyclically annotated call graph has a preference order.

Proof. The proof proceeds by induction on the number of call graph methods. The result trivially holds for the empty call graph. For the inductive step, assume the result holds for all call graphs with at most k methods and consider an arbitrary call graph with $k+1$ methods.

First, there must be a root method whose annotation is the highest among all the methods residing in the same site. Otherwise a dependency cycle can be formed: take the maximal methods for all sites, which are internal by assumption, and their root ancestors. For every maximal (internal) method there is a $\rightarrow^+$ path reaching it, starting from its corresponding root. Similarly, for every root there is an incoming $\dashrightarrow$ edge from the maximal internal method that resides in its site. A cycle exists since the (bipartite) sub-graph of roots and maximal methods is finite, and every method has a successor (a $\rightarrow^+$ for root methods, and a $\dashrightarrow$ for maximal methods). This contradicts that the annotation is acyclic.

Now, let n be a maximal root method, and let $>$ be a preference order for the graph that results by removing n, which exists by the inductive hypothesis. We extend $>$ by adding $n > m$ for every other method m. The order is topological since n is a root. The order respects annotations since n is maximal in its site. □

The following corollary follows immediately.

Corollary 6.3.3. Every acyclically annotated call graph has a preference order $>$ that satisfies that if $n \succ m$ then $n > m$.

6.3.2 Reachable States

A global state of a distributed system is *admissible* if all existing processes (active or waiting) in a method n have a matching caller process in every ancestor method of n. That is, if the state corresponds to the outcome of some admissible allocation sequence.

Theorem 6.3.4. The set of reachable states of a system using LIVE-P as the allocation manager is precisely the set $\mathcal{S}(\varphi)$.

Proof. It follows directly from the specification of LIVE-P that all its reachable states satisfy φ. Therefore, we only need to show that all admissible φ-states are reachable.

We proceed by induction on the number of active processes in the system. The base case, with no active process, is the initial state of the system Θ, which is trivially reachable by LIVE-P. For the inductive step, consider an arbitrary admissible φ-state σ with some active process. Since the call graph is acyclic and finite, there must be some active process P in σ with no active descendants. The state σ' obtained by removing P from σ is an admissible φ-state (all the conditions of admissibility and the clauses of φ are either simplified or identical); by the inductive hypothesis, σ' is reachable by LIVE-P. Since σ is obtained from σ' by an allocation that preserves φ, then σ is reachable by LIVE-P. □

Theorem 6.3.4 states that for every sequence s that leads to a φ-state there is a sequence s' arriving at the same state for which all prefixes also reach φ-states. The sequence s' is in the language of LIVE-P. Perhaps somewhat surprisingly, the set of reachable states of BASIC-P is also $\mathcal{S}(\varphi)$, the set of all admissible φ-states. To prove this we first need an auxiliary lemma.

Lemma 6.3.5. In every φ-state, an allocation request in site A with annotation k has the same outcome using BASIC-P and LIVE-P, if there is no active process in A running a method with annotation value strictly smaller than k.

Proof. First, in every φ-state, if BASIC-P grants a resource so does LIVE-P, by Lemma 6.2.3. We need to show that in every φ-state, if LIVE-P grants a request for annotation value k and $act_A[j] = 0$ for all $j < k$, then BASIC-P also grants the request. In this case,

$$T_A - t_A = Act_A[0] = \sum_{j=0}^{T_A-1} act_A[j] = \sum_{j=k}^{T_A-1} act_A[j] = Act_A[k]. \tag{6.2}$$

Since LIVE-P grants the request, then $Act_A[k] + 1 \leq T_A - k$ and $Act_A[k] < T_A - k$. Using (6.2), $T_A - t_A < T_A - k$, and consequently $k < t_A$, so BASIC-P also grants the resource. □

Theorem 6.3.6. The set of reachable states of a system using BASIC-P as allocation manager is precisely the set $\mathcal{S}(\varphi)$.

Proof. The proof is analogous to the characterization of the reachable states of LIVE-P,

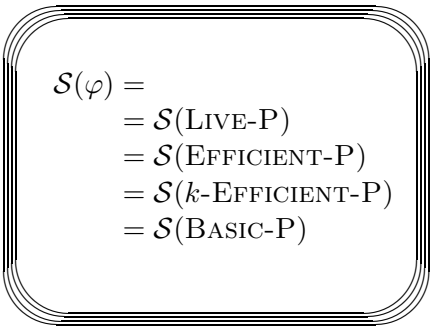


Figure 6.6: Reachable state spaces of the deadlock avoidance protocols

except that the process P removed in the inductive step is chosen to be a minimal active process in some preference order $>$. This guarantees that P has no subprocesses (by the topological property of $>$), and that there is no active process in the same site with lower annotation (by the annotation respecting property of $>$). Consequently, Lemma 6.3.5 applies, and the resulting state is also reachable by BASIC-P. □

Theorem 6.3.6 can also be restated in terms of allocation sequences. For every admissible allocation string that arrives at a φ-state there is an admissible allocation string that arrives at the same state and (1) contains no deallocations, and (2) all the allocations occur following some preference order. It follows from Theorem 6.3.6 that $\mathcal{S}(\text{BASIC-P}) = \mathcal{S}(\varphi)$, and hence, as depicted in Figure 6.6:

$$\mathcal{S}(\text{BASIC-P}) = \mathcal{S}(\text{EFFICIENT-P}) = \ldots = \mathcal{S}(k\text{-EFFICIENT-P}) = \ldots = \mathcal{S}(\text{LIVE-P}).$$

6.4 Summary

In this chapter we have generalized all the distributed deadlock avoidance algorithms presented earlier by restating them in terms of strengthenings of the invariant φ. The most liberal protocol, LIVE-P , also ensures liveness at the cost of maintaining more complicated data-structures, which requires a non-constant number of operations per allocation request. The simplest protocol, BASIC-P , can be implemented with one operation per request but allows less concurrency.

We have shown that the reachable state spaces of these protocols are the same. This

fact enables the following design principle, which provides the system designer with more freedom to implement new protocols: every local protocol P guarantees deadlock avoidance if P satisfies the following conditions:

(1) whenever BASIC-P enables a request, so does P, and

(2) whenever P enables a request so does LIVE-P.

If both conditions are fulfilled, then all reachable states of the protocol P satisfy φ and BASIC-P guarantees that P allows some progress. Informally, (2) guarantees that the system stays in a safe region, while (1) ensures that enough progress is made. This principle immediately allows, for example, the combination of different protocols in the family at different sites. If a site has a constraint in memory or CPU time, then the simpler BASIC-P is preferable, while LIVE-P is a better choice if a site needs to maximize concurrency.

This result also facilitates the analysis of alternative protocols. Proving a protocol correct (deadlock free) can be a hard task if the protocol must deal with scheduling, external environmental conditions, etc. With the results presented in this chapter, to show that an allocation manager has no reachable deadlocks it is enough to map its reachable state space to that of an abstract system in which all states satisfy φ, and all allocation decisions are at least as liberal as in BASIC-P. This technique is used in next chapter to design an efficient distributed priority inheritance mechanism.

Chapter 7

Dealing with Priority Inversions

This chapter studies how to alleviate priority inversions in distributed real-time and embedded systems, and proposes a solution based on a distributed version of the priority inheritance protocol (PIP). The PIP protocol does not prevent deadlocks, but we remedy this shortcoming by starting from an already deadlock free system, with the use of the deadlock avoidance protocols presented in previous chapters. In our framework, priority inversions can be detected locally, so we obtain an efficient dynamic resource allocation system that avoids deadlocks and handles priority inversions.

Alternative approaches to priority inversions in distributed systems use variations of the priority ceiling protocol (PCP). The PCP was designed in centralized systems as a modification of PIP that also prevents deadlocks. However, its adaptation to distributed system requires maintaining a global view of the acquired resources, which involves a high communication overhead.

7.1 Distributed Priority Inheritance

In this section we describe the distributed priority inheritance protocol. We will later show how to implement it using resource allocation managers, and illustrate how this mechanism helps to alleviate priority inversions. In this chapter we restrict our attention to fixed-priorities.

Priorities are used in real-time systems to gain a higher confidence that more important tasks will be accomplished in time. Higher priority processes are chosen by schedulers over lower priority ones. Since resources are limited and non-preemptable, it can happen that a

process must wait for a lower priority process to complete and release the resource. This situation is called a priority inversion, which can affect performance and the ability of the system to satisfy safety requirements. We show here how to alleviate priority inversions and bound their blocking times.

Given a system $\mathcal{S} : \{\mathcal{R}, M, \mathcal{G}\}$ with call graph $(M, \rightarrow, I)$ a priority specification consists of a description of the possible priorities at which processes can run. Priorities are specified as a fixed, finite and totally ordered set *Prio*. Without loss of generality, we take $Prio = \{1, \ldots, p_m\}$, where lower value means higher priority. The highest priority is represented by 1, while p_m stands for the lowest priority.

Definition 7.1.1 (Priority Assignment). *A* priority assignment *is a map from initial methods to sets of priorities:*

$$\mathbf{P} : I \rightarrow 2^{Prio}.$$

Informally, a priority assignment of an initial method i indicates the possible priorities at which processes starting at i can execute. A *priority specification* $\langle \mathcal{R}, M, \mathcal{G}, \mathbf{P} \rangle$ equips a system with a priority assignment. When a process is created, it declares both the initial method i—as in the unprioritized environment— and its initial priority from $\mathbf{P}(i)$, called the *nominal priority* of the process.

We now describe the distributed priority inheritance protocol:

(PI1) A process maintains a *running priority*, which is set initially to its nominal priority.

(PI2) Let P be a process, running at priority p, that is denied access to a resource in site A, and let Q be an active process in A running at a priority lower than p. Q and all its subprocesses set their priority to p or their current running priority, whichever is higher. We say that Q is *accelerated* to p. Since subprocesses of Q may be running in remote sites, an acceleration can require sending "acceleration messages".

(PI3) When a process is accelerated it does not decrease its running priority until it completes.

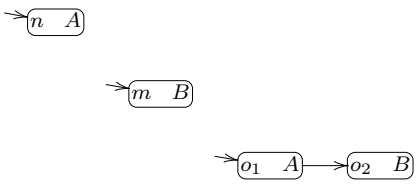
Essentially, (PI2) captures the situation of a lower priority process blocking a higher priority one. The lower priority process temporarily inherits the higher priority to complete faster and minimize the blocking time. Rule (PI3) is concerned with remote invocations that a processes being accelerated executed previously, indicating that the subprocesses serving the remote invocations must also accelerate to the higher priority.

We now characterize the set of priorities at which a method can run. A method $n{:}A$ can be executed at a priority p either if $p \in \mathbf{P}(i)$ for some initial ancestor of n, or if a process can execute n at priority lower than p and block another process running at p. This block can be produced either if there is some method in A that can be executed at p, or if some ancestor of n can block one such process. Formally,

Definition 7.1.2 (Potential Priority). *The set of potential priorities $M^{pr} \subseteq M \times Prio$ is the smallest set containing:*

1. *(n, p) for every method n that descends from $i \rightarrow^* n$, with $i \in I$ and $p \in \mathbf{P}(i)$.*
2. *(n, p) for every $(m, p) \in M^{pr}$ with $n \equiv_{\mathcal{R}} m$, and $(n, q) \in M^{pr}$ for some $q \geq p$.*
3. *(n, p) if some ancestor $m \rightarrow^+ n$ can also run at p, that is $(m, p) \in M^{pr}$.*

Example 7.1.3. Consider the prioritized system $\langle \{A, B\}, M, \mathcal{G}, \mathbf{P} \rangle$ where the set of methods M is $\{n, m, o_1, o_2\}$, the call graph $\mathcal{G}$ is:



and the priority assignment $\mathbf{P}(n) = \{1\}, \mathbf{P}(m) = \{2\}, \mathbf{P}(o_1) = \{3\}$. The set of potential priorities of this system is:

n	m	o_1	o_2
$\{1\}$	$\{1, 2\}$	$\{1, 3\}$	$\{1, 2, 3\}$

Method o_1 can run at priority 1 because o_1 resides in the same site as n and o_1 can run at 3. Since o_2 is a descendant of o_1, o_2 can also run at 1, and since m resides in the same site as o_2—and o_2 can run at higher priority 1—m can also run at 1. Moreover, m can run at 2, higher than o_2 running at 3, so o_2 can also run at 2. This way, $M^{pr} = \{(n, 1), (m, 1), (m, 2), (o_1, 1), (o_1, 3), (o_2, 1), (o_2, 2), (o_2, 3)\}$. ⊣

7.2 Priority Based Annotations

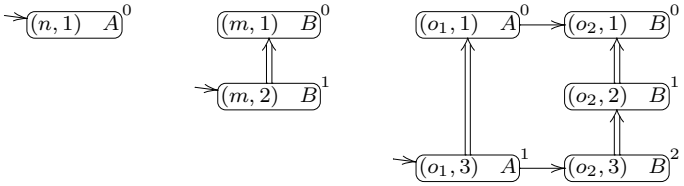
We extend the state transition system with a new transition called *acceleration*. When an acceleration transition is taken, a process P running at priority q accelerates to higher

priority p ($p < q$). It is easy to see that, if the priority inheritance protocol is used, a process running in n can only accelerate from q to p when both (n, p) and (n, q) are in M^{pr}. In this section we adapt the notion of a call graph and its annotation to handle prioritized specifications, and illustrate how blocking delays caused by priority inversions are bounded. Then, in Section 7.3, we formally present the extended model of computation and show that BASIC-P still prevents deadlock in the presence of accelerations.

Definition 7.2.1 (Prioritized Call Graph). *Given a call graph $G : (M, \rightarrow, I)$ the* prioritized call graph *is defined as $G^{pr} : (M^{pr}, \xrightarrow{pr}, \Rightarrow, I^{pr})$, where*

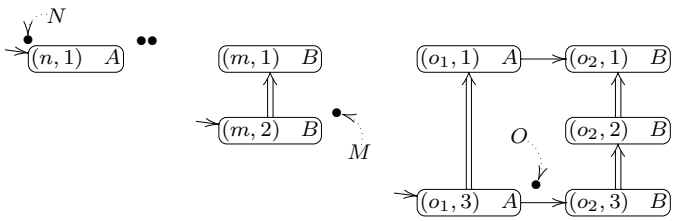
- *M^{pr} is the set of potential priorities,*
- *there is a remote invocation edge $(n, p) \xrightarrow{pr} (m, p)$ for every invocation in the original graph $n \rightarrow m$ for which both methods can run at the same priority p,*
- *accelerations connect different priorities of the same method $(n, q) \Rightarrow (n, p)$ if $p < q$, and*
- *each initial method starts at a nominal priority: $(i, p) \in I^{pr}$ whenever $i \in I$, and $p \in \mathbf{P}(i)$.*

A prioritized annotation α is a map from M^{pr} to the natural numbers. It *respects priorities* if for every two pairs (n, p) and (m, q) in M^{pr}, with $n \equiv_{\mathcal{R}} m$, $\alpha(n, p) > \alpha(m, q)$ whenever $p > q$, that is, if *higher priorities receive lower annotations.* As with unprioritized call graphs, we create an annotated call graph by adding an edge relation $\overset{pr}{\dashrightarrow}$ connecting two methods $(n, p) \overset{pr}{\dashrightarrow} (m, q)$ whenever n and m reside in the same site and $\alpha(n, p) \geq \alpha(m, q)$. If there is a path from (n, p) to (m, q) that contains at least a $\xrightarrow{pr}$ edge we say that (n, p) depends on (m, q), and we write $(n, p) \succ (m, q)$. An annotation is *acyclic* if there is no path from (n, p) to (m, q) that contains at least one $\xrightarrow{pr}$ edge. The following diagram represents the prioritized call graph of Example 7.1.3, with an annotation that is acyclic and respects priorities:

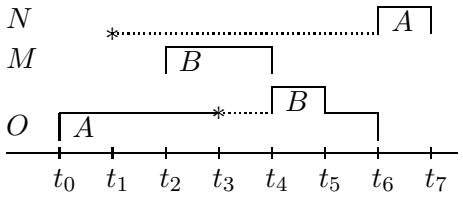


Example 7.2.2. This example shows how priority inheritance bounds the blocking time caused by priority inversions. Consider the specification of Example 7.1.3, annotated as above, and using BASIC-P as resource allocation manager for every method. Let the total number of resources be $T_A = 3$ and $T_B = 2$, and let state σ contain two active processes

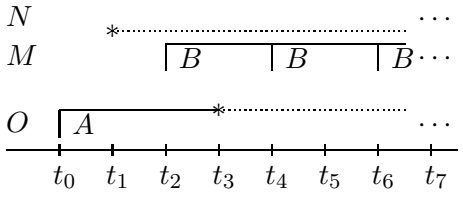
executing n at priority 1, one active process M running in m at priority 2, and one active process O running in o_1 at priority 3. In σ the available resources are $t_A = 0, t_B = 1$. Let N be a new process spawned to run n with nominal priority 1:



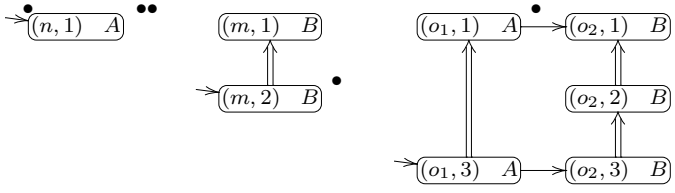
N is blocked trying to access $(n, 1)$ because no more resources are available in A, and there is a priority inversion since O holds an A resource, while running at lower priority 3. If no acceleration is performed, then the remote call of O to o_2 is blocked until M completes, so N will be blocked indirectly by M, as represented by the time interval $[t_3, t_4]$ in the following diagram:



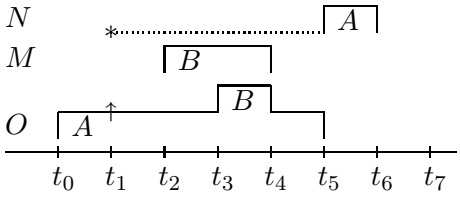
Even worse, if there are several processes waiting to execute $(m, 2)$, all of them will be scheduled in preference to O, thus blocking O and indirectly N, causing an unbounded blocking delay. This unbounded delay is represented by the open time interval $[t_3, \ldots)$:



With priority inheritance in place, O inherits priority 1 from N, and the resulting state after the acceleration is:



In this state, O will be granted a resource in o_2 in spite of M and other processes waiting at m with priority 2. O will terminate, freeing the resource demanded by N. The blocking time of N is bounded by the running time of O at priority 1, as depicted in the following time diagram, where the acceleration of O occurs at instant t_1:



The following results hold in spite of when and how accelerations are produced:

Lemma 7.2.3. If an annotation respects priorities, then accelerations preserve φ.

Proof. Let P be a process that accelerates from priority q to p. If P is waiting, the result holds immediately since the global state does not change. If P is active, executing method $n : A$, its annotation $\alpha(n, q) > \alpha(n, p)$ decreases. Therefore, all terms $Act_A[k]$ are either maintained or decreased, and φ is preserved. □

The following is an immediate corollary:

Corollary 7.2.4. The set of reachable states of a prioritized system that uses BASIC-P as the allocation manager with an acyclic annotation that respects priorities is a subset of the φ-states.

7.3 Model of Computation with Priorities

In this section we show that if BASIC-P is used with accelerations according to the priority inheritance protocol, deadlocks are not reachable. When a process inherits a new priority, all its existing subprocesses must accelerate as well, including those running in remote sites as a result of a remote invocation. A message is sent to all sites where a subprocess may be running. When the message is received, if the process exists, then it is accelerated. If the process does not yet exist, the acceleration is recorded as a future obligation. We first show deadlock-freedom if all acceleration requests are delivered immediately with global atomicity. Then, we complete the proof for asynchronous delivery in general.

Formally, we enrich the model of computation introduced in Chapter 2 to cover executions of the distributed priority inheritance protocol. First, the state of a process is enriched to include the running priority. The state of a process P is then modeled as (G, γ, Π) where (G, γ) is a labeled call graph as before, and $\Pi : M \rightarrow \mathit{Prio}$ is the priority value of each subprocess.

7.3.1 Synchronous Accelerations

We enrich the model of computation with a new transition corresponding to synchronous accelerations, where both the process and all its subprocesses accelerate instantaneously[1].

9. **Synchronous acceleration:** Let P be a process waiting to execute $n : A$, and let Q be an active process running in A with $\Pi(Q) > \Pi(P)$. The process Q and all its subprocesses accelerate instantaneously to $\Pi(P)$. Formally,

$$\tau_9 \ : \gamma(Q) = \ell_{1,2} \ \wedge \ \gamma(P) = \ell_0 \ \wedge \ \neg \mathit{En}_n(V_A) \ \wedge \ \Pi(Q) > \Pi(P) \ \wedge \bigwedge_{S \in \mathit{desc}(Q)} \Pi'(S) = \mathit{max}\{\Pi(S), \Pi(P)\} \ \wedge \ \mathit{pres}(V_{\mathcal{R}})$$

In order to prove that Basic-P provides deadlock avoidance in the extended model of computation, we derive an abstract (unprioritized) system and show that if the prioritized system has reachable deadlocks so does the derived system.

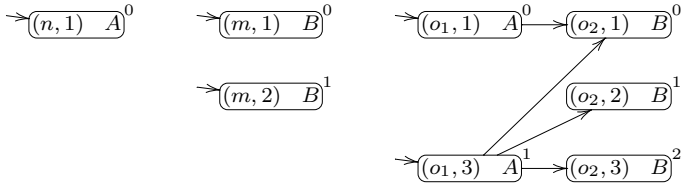
Definition 7.3.1 (Flat call graph). *Given a priority specification and its prioritized call graph $(M^{pr}, \xrightarrow{pr}, \Rightarrow, I^{pr})$, the derived flat call graph $\mathcal{G}^\flat : \langle M^{pr}, \rightarrow^\flat, I^\flat \rangle$ consists of the following components:*

- *M^{pr} is the set of methods with their potential priorities,*
- *there is an edge $(n, q) \rightarrow^\flat (m, p)$ if $p \leq q$ and $n \rightarrow m$ in the original call graph, and*
- *$I^\flat$ is the set of initial methods with all their potential priorities.*

We use $\mathcal{S}^\flat : \langle \mathcal{R}, M, \mathcal{G}^\flat \rangle$ for the (unprioritized) system that results from the flat call graph. It is easy to see that the reachable state space of a process (the resources and running priorities of the process and each of its active subprocesses) is the same in a system and its flat version. Moreover, if an annotation α of a prioritized specification is acyclic and respects

[1] We number this new transition τ_9 because it is added to the set $\{\tau_1, \ldots, \tau_8\}$ introduced in the basic model of computation in Chapter 2.

priorities then α, when interpreted in the flat call graph, is also acyclic. For example, the following diagram shows the flat call graph for the specification in Example 7.1.3, annotated:



Theorem 7.3.2. Given a prioritized system $\mathcal{S}$ and an acyclic annotation that respects priorities, every global state reachable by $\mathcal{S}$ is also reachable by $\mathcal{S}^\flat$, if Basic-P is used as the allocation manager.

Proof. Corollary 7.2.4 states that every global state reachable by $\mathcal{S}$ is a φ-state. Theorem 6.3.6 says that Basic-P can reach all $\mathcal{S}(\varphi)$ states. The result follows immediately. □

It is important to note that Theorem 7.3.2 states that for every sequence of requests and accelerations that leads to state σ in $\mathcal{S}$, there is a—possibly different— sequence that leads to σ in the abstract system $\mathcal{S}^\flat$. Theorem 7.3.2 does not imply, though, that every transition in $\mathcal{S}$ can be mimicked in $\mathcal{S}^\flat$, which is not the case—in general—for accelerations. A consequence of Theorem 7.3.2 is that deadlocks are not reachable in $\mathcal{S}$, since the same deadlock would be reachable in $\mathcal{S}^\flat$, which is deadlock free:

Corollary 7.3.3. If α is an acyclic annotation that respects priorities, and Basic-P is used as the resource allocation manager, then all runs of $\mathcal{S}$ when accelerations are executed in global atomicity are deadlock free.

7.3.2 Asynchronous Accelerations

Let us first introduce an alternative extension to the model of computation that does not assume that all subprocesses accelerate in global atomicity.

9. **Asynchronous acceleration:** Let P be a process waiting to execute $n : A$, and let Q be an active process running in A with $\Pi(Q) > \Pi(P)$. This transition models the acceleration of Q to $\Pi(P)$. Formally,

$$\begin{aligned}\tau_9 \ : \gamma(Q) = \ell_{1,2} \ \wedge \ \gamma(P) = \ell_0 \ \wedge \ \neg En_n(V_A) \ \wedge \ \Pi(Q) > \Pi(P) \ \wedge \\ \Pi'(Q) = \Pi(P) \ \wedge \ pres(V_{\mathcal{R}})\end{aligned}$$

10. **Subprocess acceleration:** Let S be a proper subprocess of a process Q, such that S runs at a lower priority. This corresponds to a situation where Q has accelerated, and the acceleration of S is pending, waiting for the acceleration message to arrive. Formally,

$$\tau_{10} \ : \ S \in \mathit{desc}(Q) \ \wedge \ \Pi(S) > \Pi(Q) \ \wedge \ \Pi'(S) = \Pi(Q) \ \wedge \ \mathit{pres}(V_{\mathcal{R}})$$

When an arbitrary asynchronous subsystem is assumed, the proof of deadlock freedom is more challenging. In this case, the flat system does not directly capture the reachable states of the system with priorities, since subprocesses may accelerate later than their ancestors.

Theorem 7.3.4 (Annotation Theorem for Prioritized Specifications). If α is an acyclic annotation that respects priorities, and BASIC-P is used as a resource allocation manager for every call graph method, then all runs are deadlock free.

Proof. Not all states reachable in the prioritized system are reachable states of the derived flat system $\mathcal{S}^\flat$. However, it is easy to see that this is the case when no transition τ_{10} is enabled. Assume, by contradiction, that deadlocks are reachable, and let σ be a deadlock state. In σ no progressing transition is enabled, so all the pending messages have been delivered (no τ_{10} transition is enabled). Therefore, the reachable state is indeed a state of $\mathcal{S}^\flat$. By Theorem 7.3.2 BASIC-P ensures that some process can progress, which contradicts that σ is a deadlock state. □

7.4 Summary

This chapter has presented a distributed priority inheritance protocol built using a deadlock avoidance mechanism. This protocol involves less communication overhead than a distributed PCP, since inversions can be detected locally. Using this protocol, an explicit calculation of the upper-bound on blocking times can be carried out, assuming known latencies of messages, periods of processes and running times of methods. Then, potentially, schedulability analysis can be performed, but that is out of the scope of this dissertation.

The message complexity of a single occurrence of an acceleration due to priority inheritance is given by the number of different sites in the set of descendants of the accelerated process, which in the worst case is $|\mathcal{R}|$. However, this communication is one-way, in the

sense that once the message is sent to the network, the local process can immediately proceed with the acceleration. Moreover, broadcast can be used when available. Under certain semantics for remote calls, this worst case bound can be improved. For example, with synchronous remote calls (a caller is blocked until its remote invocation returns), one can build, using a pre-order traversal of the descendant subtree, an order on the visited sites. Then, a binary search on this order can be used to find the active subprocess where the nested remote call is executing. This gives a worst case $log\ |\mathcal{R}|$ upper-bound on the message complexity.

Most dynamic priority assignment mechanisms, like EDF, require querying for the current status of processes in order to compare their relative priorities. Our priority inheritance mechanism can be used with dynamic priorities if there is some static discretization of the set of priorities at which processes may run. The only requirement of a priority change mechanism presented in this chapter is that subprocesses must only increase (never decrease) their priorities. In the case of EDF, accelerations would not only be caused by a priority inversion but also by the decision of the processes to increase its priority to meet a deadline, which does not compromise deadlock freedom.

Chapter 8

Conclusions

Modern distributed real-time and embedded systems are built using a middleware that abstracts away lower level details and provides a variety of useful services. However, this approach will only be effective as long as the services have well defined and predictable semantics. In this thesis we have studied one important middleware service: thread allocation with deadlock avoidance guarantees. The main contribution is to show that even though the general adaptation of centralized deadlock avoidance techniques to distributed systems is widely regarded as impractical, efficient solutions exist if more information about the systems is known in the form of remote call dependencies between components. These dependencies can be extracted by static program analysis. Even though static anlysis is usually performed for verification or bug hunting purposes, this dissertation shows that static analysis can be used to generate distributed deadlock avoidance algorithms for particular cases for a problem that admits no general solution. The techniques developed in this dissertation offer a practical trade-off: if an accurate description of the processes involved is known before deployment, then a specific allocation manager that guarantees deadlock free operation can be tailored for the particular scenario at hand.

The following are possibilities for further work in this line of research.

- the protocols presented here assume that each method receives a single annotation based on a call graph, which captures the remote invocations that each method *may* perform. In other words, each method is treated monolithically and all possible scenarios are considered at once. A more sophisticated schema could exploit further information about the calls that *will* be performed. This extra information can be

available statically if, for example, it is known that in differente phases or modes of execution some methods can only perform certain remotes calls. Also, a finer grain annotation could assign different values to different code segments within the same method. This can thenq be exploited dynamically, by allowing a process to change (decrease) its annotation based on the fact that from its current state the process can only perform some subset of the remote calls.

- The policy for thread allocation studied in this dissertation is *WaitOnConnection*, which stablishes that process cannot be preempted and the thread assigned is only reclaimed after the process terminates. One alternative policy, called *WaitOnReactor* permits the reuse of a thread as soon as the process blocks waiting for a remote reply. This policy immediately prevents deadlocks, but can lead to deadline violations. In fact, it is easy to show that under *WaitOnReactor* there are scenarios in which no process ever terminates, even though *WaitOnConnection* would allow every process to complete in time. An interesting follow-up problem consists of the study of mixed strategies. For example, a policy can allow a resource to be reused by nested upcalls generated by the process that holds the thread. This strategy will permit the adaptation of the techniques presented in this dissertation to increase resource utilization while still avoiding deadlocks. Moreover, one such strategy could handle recursive calls to the same method (that is, cyclic call graphs), which is not supported by the scheme presented here.

- Another factor that is not exploited in the algorithms presented in this dissertation is the existence of more global information about the system being deployed, for example a bound on the maximum number of processes in the system. If these bounds are known, a deadlock free system can be constructed by allocating enough resources—even with no resource allocation manager. Alternatively, if resources are fixed, the annotations could be reduced for a given scenario based on the maximum allocation possible, as provided by the bounds.

Summary

This dissertation has studied how to efficiently prevent deadlocks in distributed systems. The processes involved in a deadlock state cannot proceed to complete their assigned tasks,

in spite of sufficient availability of resources. The circular block is caused by an unfortunate lack of coordination between the actions of the processes involved. The target application domain of this work is distributed real-time and embedded systems, where reaching a deadlock state—even if it can be detected—is usually not acceptable.

The model of execution in this application domain is the two-way remote procedure call, where the execution of a method in a remote processor may be required to complete a task. The resource handled is the thread or execution context needed locally to run a method. Naturally, a nested dependency between execution contexts at different processing sites arises due to the flow of computation of a process. One strategy commonly used for managing threads in DRE systems is *WaitOnConnection*. This strategy specifies that a process locks the thread until its running method terminates, which requires the collection of results from remote invocations. Since thread pools are of fixed size, and there is no restriction on the nature or number of processes running in parallel, deadlocks can potentially be reached.

The conventional mechanism to prevent deadlocks in DRE systems is monotone locking, where resources are forced to be acquired in a predetermined order. Even though monotone locking suffers a high overhead due to the reduced concurrency and the delay due to the two-way communication necessary to reserve resources upfront, many current state of the art implementations of DRE systems use this strategy to prevent deadlock. This thesis presents an efficient alternative, in the form of a distributed deadlock avoidance mechanism: at runtime deadlocks are dynamically avoided by a controller that keeps the system in a safe state, where progress is ensured.

A *general* distributed adaptation of centralized deadlock avoidance algorithms is widely considered impractical, due to the communication overhead necessary to maintain global views in asynchronous distributed systems. This thesis shows that efficient solutions exist, provided that accurate information about the control flow of the processes is known—in the form of call graphs—before deployment. An efficient solution in this context is one that requires no communication between sites to decide the safety of an allocation.

The solution consists on the combination of static computations based on the call graphs, and runtime protocols that control the thread allocation. The call graph information can be obtained from the specification or directly from the code by static analysis. These call graphs are then analyzed to generate an annotation for each of the methods, which provides an estimate of the level of resource availability needed to safely grant a thread to run the

method. These annotations are agreed upon by all processors before the system starts its execution. At runtime, the protocols can inspect the current local resource availability, and together with the annotation of the method that a process requests, use that information to decide whether to allocate the resource.

Chapter 3 includes formal proofs of the correctness of these deadlock avoidance protocols. The simplest protocol, Basic-P , compares the annotation against the current resource availability, assigning a resource if the availibity is higher. More sophisticated protocols keep track of the number of processes running local methods with different annotation levels. The ability of these protocols to prevent deadlocks is based on properties of annotations, in particular on a simple notion of acyclicity. Chapter 4 details methods to generate the most efficient acyclic annotations, and contains proofs of the risks involved when using cyclic annotations. First, deciding statically whether a particular scenario with a cyclic annotation and fixed initial set of resources can reach a deadlock is computationally hard. Second, every scenario with a *cyclic* annotation will have reachable deadlocks if provided with enough initial resources in each site. Hence, in practice, to ensure the correctness of a deployment the annotations must be acyclic.

Chapter 5 introduces Live-P , an advanced version of these protocols, that can also prevent starvation, by ensuring that every process eventually gets its needed resources. This solution is still accomplished in a completely local fashion, without extra communication. This protocol, however, cannot be implemented in a conventional programming language with a constant number of operations per allocation and deallocation. We show that there are efficient ways to implement Live-P using a logarithmic number of operations, in terms of the size of the resource pool managed.

Finally, Chapter 6 shows that, quite surprisingly, all the deadlock avoidance protocols have the same reachable state space, even though the more efficient protocols allow more runs than the less liberal protocols. Beyond its theoretical interest, this result has a practical impact, since it eases the design and proof of new deadlock avoidance protocols. It is enough to show that, for each request, the new protocol is never more conservative than Basic-P—the most conservative in the family—and never more liberal than Live-P—the most liberal. In particular, this technique enables the construction of a distributed priority inheritance mechanism in Chapter 7, which is one of the most efficient ways known to alleviate priority inversions in distributed real-time and embedded systems. Since the system is already deadlock free, a simple priority inheritance protocol bounds the blocking

time caused by priority inversions efficiently. Alternative solutions in the literature either drastically restrict the nature of the processes, or deal with deadlock and priority inversions altogether, obtaining less efficient solutions.

Bibliography

[AA91] Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, February 1991.

[AD76] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering (ICSE '76)*, pages 562–570, Los Alamitos, CA, 1976. IEEE Computer Society Press.

[AS90] Baruch Awerbuch and Michael Saks. A dining philosophers algorithm with polynomial response time. In *Proceedings of the IEEE Symposium on the Foundations of Computer Science (FOCS'90)*, pages 65–74. IEEE, 1990.

[ASK71] Toshiro Araki, Yuji Sugiyama, and Tadao Kasami. Complexity of the deadlock avoidance problem. *2nd IBM Symposium on Mathematical Foundations of Computer Science*, pages 229–257, 1971.

[ASS+99] Marcos Kawazoe Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar Deepak Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[Bir89] Andrew D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.

[BNS70] Laszlo A. Belady, Robert A. Nelson, and Gerald S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, 1970.

[But05] Giorgio C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer, New York, NY, 2005.

[Centy] Center for Distributed Object Computing. nORB—Special Purpose Middleware for Networked Embedded Systems. `http://deuce.doc.wustl.edu/nORB/`, Washington University.

[CES71] Edward G. Coffman, M. J. Elphick, and Arie Shoshani. System deadlocks. *Computing Surveys*, 3:67–78, 1971.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[CM84] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[CR83] O. S. F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Communications of the ACM*, 26(2):146–147, 1983.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[Dij65] Edsger W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

[Dij68] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communication of the ACM*, 11(5):341–346, 1968.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles (SOSP '79)*, pages 150–162, Pacific Grove, CA, 1979. ACM Press.

[GMB85] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.

[GMUW01] Hector García-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database Systems: The Complete Book*. Prentice-Hall, 2001.

[Hab69] Arie N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12:373–377, 1969.

[Hav68] James W. Havender. Avoiding deadlock in multi-tasking systems. *IBM Systems Journal*, 2:74–84, 1968.

[HCG01] Frank Hunleth, Ron Cytron, and Christopher D. Gill. Building customizable middleware using aspect oriented programming. In *Proceedings of the Workshop on Advanced Separation of Concerns (OOPSLA'01)*, pages 1–6, 2001.

[Hol72] Richard C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4:179–196, 1972.

[Insty] Institute for Software Integrated Systems. The ACE ORB (TAO). `http://www.dre.vanderbilt.edu/TAO/`, Vanderbilt University.

[LR81] Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'81)*, pages 133–138. ACM Press, 1981.

[Mae85] Mamory Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*. Springer-Verlag, 1995.

[Mue99] Frank Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *Proceedings of 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 340–349, Phoenix, AZ, December 1999. IEEE Computer Society.

[NTA96] Mohamed Naimi, Michel Trehel, and André Arnold. A $log(n)$ distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, 1996.

[Pap86] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[RA81] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.

[RA83] Glenn Ricart and Ashok K. Agrawala. Author's response to On mutual exclusion in computer networks. *Communications of the ACM*, 26(2):147–148, 1983.

[Ray89] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, February 1989.

[Rev05] Spyros A. Reveliotis. *Real-time Management of Resource Allocation Systems: a Discrete Event Systems Approach*. International Series In Operation Research and Management Science. Springer, 2005.

[SA97] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Queensland AUUG Summer Technical Conference, Brisbane, Australia*, 1997.

[SABS05] Julien Sopena, Luciana Arantes, Marin Bertier, and Pierre Sens. A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. In *Proceedings of Euro-Par'05*, volume 3648 of *LNCS*, pages 654–663, Lisboa, Portugal, 2005. Spring-Verlag.

[Sch98] Douglas C. Schmidt. Evaluating Architectures for Multi-threaded CORBA Object Request Brokers. *Communications of the ACM Special Issue on CORBA*, 41(10), October 1998.

[SG04] Venkita Subramonian and Christopher D. Gill. A generative programming framework for adaptive middleware. In *Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS'04)*. IEEE, 2004.

[SGG03] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, Sixth edition, 2003.

[Sin89] Mukesh Singhal. Deadlock detection in distributed systems. *IEEE Computer*, 22(11):37–48, November 1989.

[SK85] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, November 1985.

[SLH97] Douglas C. Schmidt, David L. Levine, and Timothy H. Harrison. The design and performance of a real-time CORBA object event service. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 184–200, 1997.

[SMFGG98] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha S. Gokhale. Alleviating priority inversion and non-determinism in real-time CORBA ORB core architectures. In *Proc. of the Fourth IEEE Real Time Technology and Applications Symposium (RTAS'98)*, pages 92–101, Denver, CO, June 1998. IEEE Computer Society Press.

[Spu95] Marco Spuri. *Earliest deadline scheduling in real-time systems*. PhD thesis, Scoula Superiore S. Anna, Pisa, Italy, 1995.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[SS94] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. McGraw-Hill, Inc., New York, NY, 1994.

[SSGM06] César Sánchez, Henny B. Sipma, Christopher D. Gill, and Zohar Manna. Distributed priority inheritance for real-time and embedded systems. In Alex Shvartsman, editor, *Proceedings of the 10th International Conference On Principles Of Distributed Systems (OPODIS'06)*, volume 4305 of *LNCS*, Bordeaux, France, 2006. Springer-Verlag.

[SSM+06] 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 *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, Rhodas, Greece, 2006. IEEE Computer Society Press.

[SSM07a] César Sánchez, Henny B. Sipma, and Zohar Manna. A family of distributed deadlock avoidance protocols and their reachable state spaces. In *Fundamental Approaches to Software Engineering (FASE'07)*, volume 4422 of *Lecture Notes in Computer Science*, pages 155–169. Springer-Verlag, 2007.

[SSM07b] César Sánchez, Henny B. Sipma, and Zohar Manna. Generating efficient distributed deadlock avoidance controllers. In *Proceedings of the Fifteenth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'07)*. IEEE Computer Society Press, 2007.

[SSMG06] César Sánchez, Henny B. Sipma, Zohar Manna, and Christopher Gill. Efficient distributed deadlock avoidance with liveness guarantees. In *Proceedings of the 6th Annual ACM Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, 2006. ACM Press.

[SSRB00] 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.

[SSS+03] César Sánchez, Sriram Sankaranarayanan, Henny B. Sipma, Ting Zhang, David Dill, and Zohar Manna. Event correlation: Language and semantics. In Rajeev Alur and Insup Lee, editors, *EMSOFT 2003*, volume 2855 of *LNCS*, pages 323–339. Spring-Verlag, 2003.

[SSS+05] César Sánchez, Henny B. Sipma, Venkita Subramonian, Christopher Gill, and Zohar Manna. Thread allocation protocols for distributed real-time and embedded systems. In Farn Wang, editor, *25th IFIP WG 2.6 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'05)*, volume 3731 of *LNCS*, pages 159–173, Taipei, Taiwan, October 2005. Springer-Verlag.

[SSSM05a] César Sánchez, Henny B. Sipma, Matteo Slanina, and Zohar Manna. Final semantics for Event-Pattern Reactive Programs. In José Luiz Fiadeiro, Neil

Harman, Markus Roggenbach, and Jan Rutten, editors, *First International Conference in Algebra and Coalgebra in Computer Science (CALCO'05)*, volume 3629 of *LNCS*, pages 364–378. Springer-Verlag, September 2005.

[SSSM05b] César Sánchez, Matteo Slanina, Henny B. Sipma, and Zohar Manna. Expressive completeness of an event-pattern reactive programming language. In Farn Wang, editor, *25th IFIP WG 2.6 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'05)*, volume 3731 of *LNCS*, pages 529–532. Springer-Verlag, October 2005.

[Sta98] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, Inc., Upper Saddle River, NJ, Third edition, 1998.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

Index