RANKERS: A PANACEA FOR SYNCHRONIZATION

Ambuj K. Singh* and Mohamed Gouda†

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-15 May 1989

* Work supported in part by ONR Grants N00014-86-K-0763 and N00014-87-K-0510.
† Work supported in part by ONR Grant N00014-86-K-0763.

Rankers: A Panacea For Synchronization

Ambuj K. Singh* and Mohamed G. Gouda†
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract

The variety of synchronization problems in the computing science is immense: dining philosophers, FCFS doorway, producer-consumer, mutual exclusion, atomic register construction etc. We develop the abstraction of rankers in order to compare the synchronization requirements of these problems and solve them modularly. Rankers have two mandatory properties – *responsiveness* and *precedence* and three optional ones – *acyclicity, comparability,* and *stability*. Adding the three optional properties one after another we obtain a chain of four rankers. Each ranker in this chain is best suited to solve a particular class of synchronization problems and together, the four rankers are sufficient to solve most synchronization problems in the literature. We devise a general strategy to solve synchronization problems using rankers and illustrate this strategy by solving some of these problems. In each solution, we only use the properties of the particular ranker chosen and not to how the ranker is implemented, thus separating the two concerns of what a ranker's properties are and how these properties are achieved. Finally, we present implementations for the four rankers mentioned above.

*Work supported in part by ONR Grants N00014-86-K-0763 and N00014-87-K-0510
†Work supported in part by ONR Grant N00014-86-K-0763

1 Introduction

Synchronization is at the heart of concurrent computation. Any study of synchronization usually begins by focusing on a domain of interest and by abstracting the significant aspects of the domain into a suitable synchronization problem. The problem is then studied and solved by an algorithm. The variety of these synchronization problems and their solutions is immense. To name a few of these problems, we have, atomic register construction [15], dining philosophers [7], drinking philosophers [3], FCFS doorway [12], mutual exclusion [6], producer-consumer [2], readers-writers [5], resource allocation [9], etc. Each of these problems has been solved in a variety of ways. For example, the mutual exclusion problem has been solved by time-stamps, semaphores (weak and strong, binary and n-ary), tokens, tickets, trees, and acyclic graphs. In fact new solutions are being continually designed for these problems.

Given the multitude of synchronization problems and their solutions, one is led to search for a unifying framework, a framework in which the problems can be stated and solved modularly at a high level. Such solutions should focus on the synchronization requirements of the problem at hand and identify the properties that are required to solve the problem. Once these properties have been isolated and a solution that relies on these properties has been designed, implementations that achieve these properties can be defined. Such an implementation along with the high level solution constitutes a complete solution to the problem. Besides the obvious advantages of modularity and separation of concerns, this approach allows us to study and compare the various synchronization problems. The need and the usefulness of such a unifying framework that permits us to define the interface between the properties that are required to solve a synchronization problem, and how they are achieved by an implementation is the underlying idea behind this work.

Given that a unifying framework is both needed and useful, how should one go about designing it? We recognized that every solution to a synchronization problem necessarily involves some form of arbitration among the processes wishing to synchronize, and that the basis for this arbitration is usually some value associated with each process. We call this value the *rank* of a process. Any solution to a synchronization problem is thus separated into two phases: computation of a rank, and arbitration based on the computed rank. Depending upon the particular synchronization problem and its solution these two phases may be done in a number of different ways. For example, consider a time-stamp based solution to the mutual exclusion problem [13]. In such a solution every process that wishes to enter its critical section first obtains a time-stamp thus executing the first phase. Next, the processes compare their time-stamps and the process with the lowest time-stamp enters its critical section. This is the second phase. As another example, consider a solution to the dining philosophers problem [4] in which a *hungry* philosopher sends out requests for forks which establishes its place in an acyclic graph, thus executing the first phase. Next, the *hungry* philosophers compare their places in the graph and the philosopher at the top

of the graph *eats*. This is the second phase.

In order to separate the two phases of computation of ranks and arbitration based on the computed ranks, we propose a program module called the *ranker* that carries out the first phase of computation of ranks. (This module need not be implemented as one central process; a distributed implementation in which there is one submodule per process is the one we envision.) When a process wishes to synchronize, it requests a rank from the ranker and waits until this rank has been computed. After that the process compares its rank with the ranks of other competing processes and proceeds accordingly. (The exact mode of comparison of ranks depends upon the particular synchronization problem. For example, in the mutual exclusion problem, a process compares its rank with all other processes and enters its critical section if it has the highest rank.) When the process no longer needs its rank, it returns the rank to the ranker.

The idea of rankers has its roots in the bakery algorithm [12] that Lamport proposed as a solution to the mutual exclusion problem. In this algorithm, every process that wishes to enter its critical section first obtains a ticket (a natural number) by examining the tickets of other processes. Then, it compares its ticket with those of other processes and proceeds if it has the lowest ticket (i.e., the highest priority). Though this algorithm hints at a general abstraction of ranks and the separation of the two phases of computation of ranks and comparison of ranks, however, the author's definition of ranks is specific to the problem being solved. As a result, the interface between the implementation of rankers and their application is not defined. In fact, in order to obtain a total order on the ranks, the author constrains them to be natural numbers, thus overlooking synchronization problems that can be solved without a total order on the ranks.

More recently, the bounded time-stamps algorithm of Israeli and Li [11] and the colored tickets algorithm of Fischer et al [9] use a notion of ranks. Israeli and Li define a sequential time-stamp system and a concurrent time-stamp system and use the latter to construct a multi-writer atomic register. Fischer et al solve the resource allocation problem (also called the K-mutual exclusion problem) using colored tickets, a generalization of tickets used in the bakery algorithm. In both these algorithms, the authors concentrate on defining specific ranker implementations using bounded variables and not on delineating the interface between ranker implementations and ranker applications. As a result, their definitions remain specific to the particular problem that they solve and they do not achieve the desired modularity.

The idea of eventcounts and tickets proposed by Kanodia and Reed [21] was an earlier attempt at classifying synchronization problems. The authors solve the producer-consumer problem and the readers-writers problem problem using this abstraction. However, they do not state the properties enjoyed by eventcounts and sequencers; as a result the interface between the application and the implementation of these abstractions is not clear. Thus, eventcounts and sequencers appear to be implementations rather than specifications.

Since the presentation of an idea and the illustration of its usefulness is done best in

a formal framework, and there are more than one formalisms available, we had to make a choice. We chose UNITY [4] mainly because of our familiarity with it and partly because it is amenable to formal manipulation. We present a brief overview of this formalism in Section 2. In Section 3 we present the specification of rankers and in Section 4 we present the classification of rankers. We illustrate how synchronization problems are solved using rankers by considering the FCFS doorway problem in Section 5, the dining philosophers problem in Section 6, and the resource allocation problem in Section 7. In Section 8 we present implementations for the rankers classified in Section 4. Section 9 contains concluding remarks.

2 The Formalism

In this section we present a brief introduction to UNITY; this presentation is somewhat simplified as we discuss only those aspects that we use.

A program consists of a predicate describing the initial values of the variables and a set of assignment statements. The set of assignment statements is written down either by enumerating every statement and using $[]$ as the set constructor, or by using a quantification of the form $\langle [] \ var : range : statement \rangle$. Symbol $[]$ is called the union operator.

Program properties are expressed using four relations on predicates — **unless, invariant, ensures,** and **leads-to**. The first two are used for stating safety properties whereas the last two are used for stating progress properties.

For any two predicates p and q, the property p **unless** q holds in a program iff for all statements s in the program

$$\{p \wedge \neg q\}\ s\ \{p \vee q\}.$$

Informally, if p is true at some point in the computation, then either q never holds and p holds for ever, or q holds eventually and p continues to hold until q holds. As an example consider $x = k$ **unless** $x > k$ which states that x is monotonically increasing. As another example consider $thinking.u$ **unless** $hungry.u$ which states that philosopher u remains thinking until becoming hungry.

For any predicate p, the property **invariant** p holds in a program iff

$$\textbf{initially}\ p\ \wedge\ p\ \textbf{unless}\ false.$$

Informally, p holds initially and the program never falsifies p. As an example consider **invariant** $eating.u \Rightarrow hasfork.u$ which states that an eating philosopher u has all the required forks.

For any two predicates, p and q, the property p **ensures** q holds in a program iff p **unless** q holds in the program and there exists a statement s in the program such that

$$\{p \wedge \neg q\}\ s\ \{q\}.$$

Informally, if p is true at some point in the computation then q holds eventually and p continues to hold until q holds.

The relation **leads-to** is denoted $\mapsto$, and is defined to be the strongest relation satisfying

the following three rules.

- $(p \textbf{ ensures } q) \Rightarrow (p \mapsto q)$,
- $((p \mapsto q) \wedge (q \mapsto r)) \Rightarrow (p \mapsto r)$, and
- For any set W,
  $(\forall m : m \in W : p.m \mapsto q) \Rightarrow ((\exists m : m \in W : p.m) \mapsto q)$.

The first two rules imply that $\mapsto$ includes the transitive closure of **ensures** and the third rule allows us to induct over sets. As an example of this property consider $hungry.u \mapsto eating.u$ which states that a hungry philosopher u eventually eats. As another example consider $send.m \mapsto receive.m$ which states that if a message m is sent, it is eventually received.

Programs are composed by taking the union of their assignment statements. The union of two programs F and G is denoted $F \,[\!]\, G$.

3 Specification of Rankers

In this section we specify the properties of the user processes and the program *ranker* that computes the ranks for the user processes. For convenience, we assume that all the user processes are grouped together in a program called *user*. Program *user* is specified in Section 3.1 and program *ranker* is specified in Section 3.2.

3.1 Specification of Program *user*

Program *user* consists of a set of processes $\{u, v, ..\}$. Associated with every process u is a variable $state.u$ that can take any one of three possible values – *white*, *grey*, or *black*. Initially, $state.u = white$, for each u. For convenience, we define the following three predicates.

$$\begin{aligned} white.u &\equiv state.u = white \\ grey.u &\equiv state.u = grey \\ black.u &\equiv state.u = black \end{aligned}$$

Variables $state.u$ cycle through the values, *white*, *grey*, and *black* in that order; the transitions from *white* to *grey* and from *black* to *white* occur in program *user* while the transition from *grey* to *black* occurs in program *ranker*.

Informally, program *user* consists of some processes that may wish to synchronize with one another. The state of each process is initially *white*. When a process wants to synchronize with other processes, it sets its state to *grey*. The *ranker* then computes a current rank for this process. When the rank has been computed, the *ranker* sets the state of the process to *black*. The process can then compare its rank with the ranks of other processes and proceed accordingly. When it no longer needs the rank, the process sets its state to *white*. And the cycle continues.

3.2 Specification of Program *ranker*

Program *ranker* has a set of distinguished variables $\{r.u, r.v, ..\}$ representing the ranks of the *user* processes. These variables are read by the *user* processes and read and written by the ranker. The ranker can modify variable $r.u$ only when the state of the corresponding process u is *grey*.

The ranker defines an irreflexive relation $\prec$ on the above variables. This relation determines the relative rank of two processes; $r.u \prec r.v$ denotes that process u has a lower rank than process v. One constraint on the definition of $\prec$ is that for all u, v,

$$\neg(r.u \prec r.v \ \wedge \ r.v \prec r.u).$$

The ranker is required to maintain certain temporal relationships among the ranks of the processes and their states. This is stated in terms of five properties, *Responsiveness, Precedence, Acyclicity, Comparability,* and *Stability*, which the ranker must maintain in the composite program *user* [] *ranker*.

Responsiveness: This property states that if a process requests a rank, then the ranker eventually responds by computing a rank, i.e., every *grey* process eventually becomes *black*. Formally, for every process u,

$$grey.u \ \mapsto \ black.u \ .$$

Observe that this progress property is local to a user process. It is required of all rankers because, for a ranker to be of any use in any application, it must eventually compute ranks; otherwise, a process that wishes to synchronize may starve forever.

Precedence: This property relates the relative rank of processes u and v to the order in which the variables *state.u* and *state.v* change; if the last granting of a rank to process u precedes the last request of a rank by process v, then process u has a higher rank than process v. This property is stated as follows: for every distinct u, v,

$$\textbf{invariant } black.u \wedge black.v \wedge precedes.u.v \ \Rightarrow \ r.v \prec r.u$$

where *precedes.u.v* is an auxiliary boolean variable that captures the order in which processes u and v change their states; it is defined shortly. The predicate $(black.u \wedge black.v \wedge precedes.u.v)$ is *true* iff both processes u and v are *black* and process u transits from from *grey* to *black* before process v transits from *white* to *grey* (i.e., the last granting of a rank to process u precedes the last request for a rank by process v). The condition of precedence states that, in that case, process u has a higher rank than process v.

Observe that precedence is a safety property on pairs of processes. It is required of all rankers because it is the only property that relates the ranks of two processes to their state transitions and therefore, is the only property that requires the ranks of processes to change. In its absence, trivial ranker implementations in which the rank of a process never changes can be defined.

Like all auxiliary variables, variable $precedes.u.v$ is not implemented; it is used only in the specification and the proof of correctness. It is defined by the following two assertions. (The definition of auxiliary variables by logical assertions on the underlying variables was proposed by Misra in [16].)

initially $\neg precedes.u.v$, and

$state.u = x \wedge state.v = y \wedge precedes.u.v = b$ **unless**

$\neg(state.u = x \wedge state.v = y) \wedge (precedes.u.v \equiv black.u \wedge (white.v \vee b))$.

The first assertion states that $precedes.u.v$ is $false$ initially and the second assertion defines how $precedes.u.v$ changes with each change in the states of processes u and v. It is set to $true$ if process v becomes *white* while process u is *black*. Once $true$, it continues to remain $true$ until process u becomes non-*black* and then it is set to $false$.

Acyclicity: The two properties of rankers that we have stated so far are either local or pairwise over processes. For a number of synchronization problems, this is insufficient as some global properties on the ranks of the processes are required. The acyclicity condition imposes such a global constraint by requiring that the relation $\prec$ be acyclic at all times. It is stated formally by requiring that the transitive closure of $\prec$, denoted by $\prec^*$, be irreflexive, i.e., for all u,

invariant $\neg(r.u \prec^* r.u)$.

Acyclicity is a weaker condition than transitivity. This is because transitivity implies acyclicity (for a proof, note that by the irreflexivity of $\prec$, $\neg(r.u \prec r.u)$; consequently, on account of transitivity, $\neg(r.u \prec^* r.u)$), though acyclicity does not imply transitivity (for a proof, consider three processes u, v and w with $r.u \prec r.v$ and $r.v \prec r.w$ but no relation between the ranks of u and w). As a result, acyclicity of ranks is a weaker condition than the requirement of a partial order on ranks. Though partial orderings have nice mathematical properties, we found acyclicity to be a more natural requirement for solving synchronization problems.

Comparability: For some synchronization problems like the mutual exclusion problem, a total order on the ranks seems essential. In order to achieve such a total order, we introduce the property of comparability, which along with acyclicity gives us an irreflexive total order on the ranks. Comparability states that the ranks of any two processes can be compared, i.e., for every distinct u, v,

invariant $r.u \prec r.v \ \vee \ r.v \prec r.u$.

The proof that this property together with acyclicity implies an irreflexive total order is as follows. Let u, v, w be any three distinct processes such that $r.u \prec r.v$ and $r.v \prec r.w$. On account of comparability, $r.u \prec r.w$ or $r.w \prec r.u$. But, because $\prec$ is acyclic, $\neg(r.w \prec r.u)$; therefore, $r.u \prec r.w$. This means that $\prec$ is transitive, and therefore an irreflexive total order.

Stability: The property of precedence relates the ranks of the processes to the order in which their states changed. However, if the interval of *greyness* of one process interleaves with the interval of *greyness* of another, precedence does not impose any constraints on the ranks of the two processes. For some problems this turns out to be insufficient as we need some property about the ranks of the processes even when their *greyness* intervals interleave. (An example of such a synchronization problem is one in which a process can timeout and make a transition from a state in which it is waiting to synchronize to a state in which it does not wish to synchronize.) The property of stability allows us to make such an assertion. It states that if a *black* process u has a higher rank than another process (no matter what the state of the other process is, *white*, *grey*, or *black*) then it continues to have a higher rank until u becomes *white*. Formally, for every distinct u, v,

$$black.u \wedge r.v \prec r.u \quad \textbf{unless} \quad white.u \ .$$

4 Classification of rankers

In this section we present our classification of rankers which is obtained by examining the synchronization problems in the literature and identifying the collection of properties that are required to solve each of them. On the basis of this evidence, we obtain a chain of four rankers. The first ranker in this chain, ranker Φ, satisfies the responsiveness and precedence properties; other rankers in this chain are obtained by adding an optional property to the preceding ranker. We add the optional properties in the order — acyclicity, comparability, and stability to obtain ranker A (for acyclicity), ranker C (for comparability), and ranker S (for stability). This chain of rankers and the class of synchronization problems that they solve is shown in the following figure.

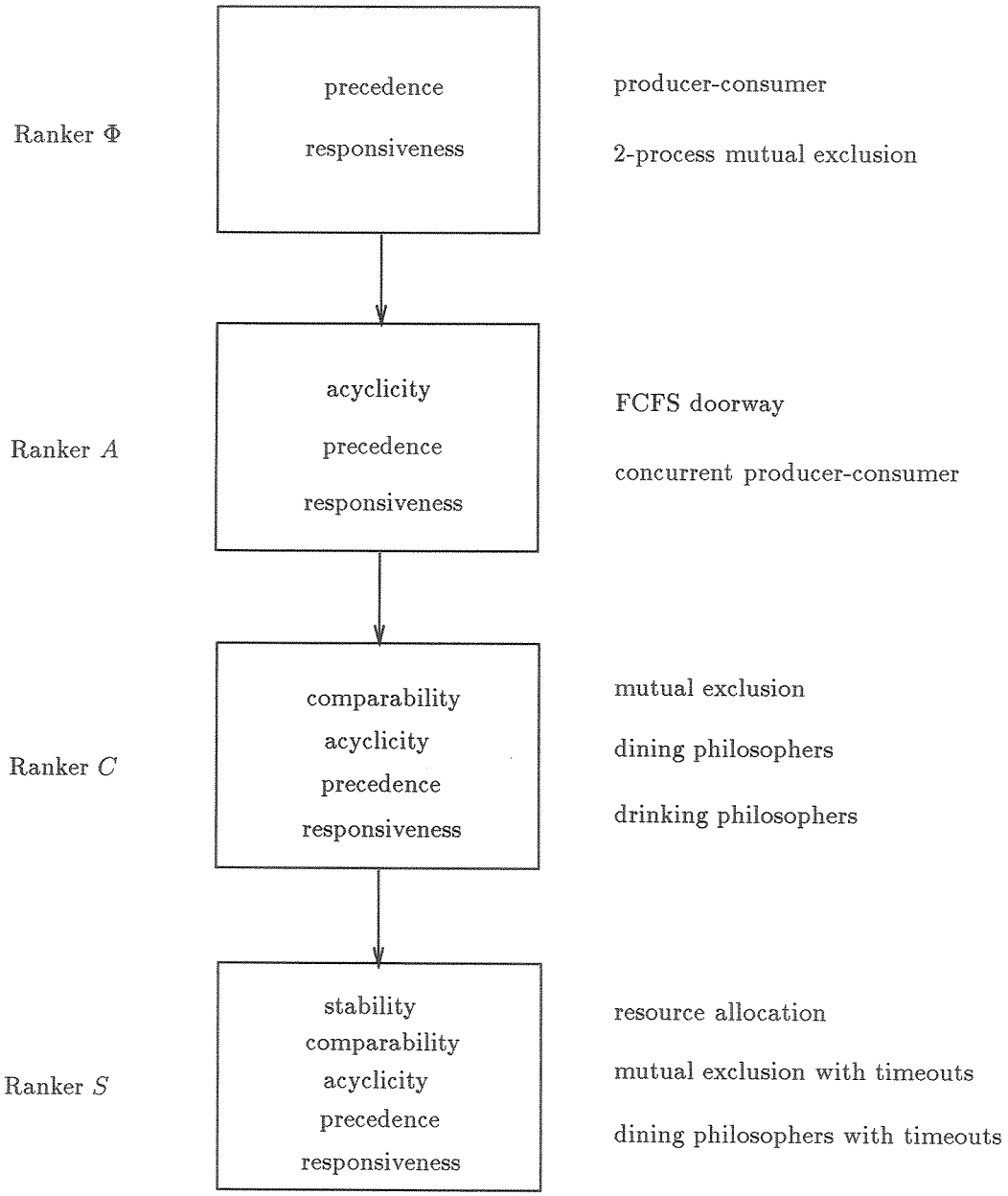

Figure 1: Classification of rankers and synchronization problems

As shown in the figure, the four rankers form a chain. As we proceed from the top of the chain to the bottom, the rankers become stronger, i.e., they satisfy more properties. However, their implementations also become more difficult. Therefore, though ranker S can

be used to solve any problem the others can solve, the problem at hand may not require all the properties of ranker S. So, we may be better off using another ranker that is weaker than ranker S, and easier to implement. Given a particular synchronization problem, it is possible to associate it with the weakest ranker that can be used to solve it. Such a classification of synchronization problems and the ranker associated with each such class is discussed next. Parts of this classification have also been observed by other researchers [11, 21].

4.1 Ranker Φ

This ranker solves synchronization problems that do not require any properties involving the ranks of more than two processes. This is because the properties satisfied by this ranker, responsiveness and precedence, are, respectively, local and pairwise properties on process ranks. Examples of problems that are solved by this ranker are the the producer-consumer problem [2] and the 2-process mutual exclusion problem [17]. Observe that if the requests for ranking by the processes are serial, i.e., *precedes* is a total order, then precedence alone guarantees a total order on the ranks of the *black* processes. As a result, this ranker corresponds to the "sequential time-stamp system" discussed by Israeli and Li [11]. The implementations that they propose for sequential time-stamp systems, therefore, can also be used for this ranker.

4.2 Ranker A

Acyclicity, a global property over all processes, guarantees the absence of cycles on the ranks of the processes. However, it allows any number of processes with maximal ranks. Therefore, this ranker solves problems that require global properties over the ranks of the processes but do not limit the number of processes that are performing the critical action at any time. Examples of such problems are the the first-come, first-served doorway problem [9, 12, 14], the concurrent producer-consumer problem [24], and the atomic register construction problem [15, 20, 23]. This ranker corresponds to the mechanism of "eventcounts" proposed by Kanodia and Reed [21]. Though eventcounts satisfy the stronger condition of a partial order, the class of problems that eventcounts can solve appears to be the same as that can be solved by this ranker. This ranker also corresponds to the abstraction of "bounded time-stamps", discussed by Israeli and Li [11], which is used to solve the atomic register construction problem.

4.3 Ranker C

As seen earlier, the property of comparability along with acyclicity makes $\prec$ an irreflexive total order on all ranks. Therefore, this ranker solves problems that require a total order on the processes. Examples of such problems are the mutual exclusion problem [6] (requires

a total order on all processes), the dining philosophers problem [7] (requires a total order on neighboring processes), and the drinking philosophers problem [9] (requires a total order on neighboring processes with conflicting requests). Because this ranker guarantees a total order on the ranks of the processes, it corresponds to the abstraction of time-stamps [13] and the mechanism of "sequencers" proposed by Kanodia and Reed [21].

4.4 Ranker S

None of the previous three rankers allow for a meaningful comparison of ranks between processes when one of them is *grey*. This is because the rank of a *grey* process may change. For some synchronization problems like the mutual exclusion problem, this is not a hindrance as a *grey* process eventually turns *black* and ranks of *black* processes can be compared. However, in some other synchronization problems like the resource allocation problem, it is possible for processes to be continuously cycling through their states, and thus, some process may always find the state of another process to be *grey* and wait forever. (This is also true of synchronization problems where a process may time out or fail in the state that it is waiting to synchronize, and make a direct transition to a state in which it is not waiting to synchronize, thus bypassing the state in which it synchronizes or performs the critical action). In order to solve these problems we use ranker S because the property of stability in conjunction with the property of precedence ensures that if a process v continuously cycles through its states while another process u is waiting, then eventually u has a higher rank than v (due to precedence) and it continues to have a higher rank (due to stability). Examples of problems that are solved by this ranker are the resource allocation problem, the mutual exclusion problem with timeouts, the dining philosophers problem with timeouts, and the drinking philosophers problem with timeouts. This ranker appears to be new and does not correspond to any of the proposed abstractions in the literature.

5 The FCFS Doorway

The FCFS (First-Come-First-Served) doorway, first proposed by Lamport as a part of the Bakery algorithm [12], has been used in a number of different mutual exclusion algorithms [14]. Its usefulness stems from the fact that it can be combined with a mutual exclusion solution that achieves global progress, to yield a composite solution that achieves individual progress. In other words, suppose we have a mutual exclusion algorithm that guarantees mutually exclusive access (i.e., no two processes are in their critical section at the same time) and global progress (i.e., if there are some processes waiting to enter their critical section, then some process eventually enters its critical section). Then, we can compose this algorithm with a FCFS doorway so that the composite program ensures individual progress (i.e., every waiting process eventually enters its critical section) in addition to mutually exclusive access. We present the specification of the problem in Section 5.1, discuss our

solution using rankers in Section 5.2, and sketch a proof of its correctness in Section 5.3.

5.1 Problem Specification

We have a set of processes $\{u, v, ..\}$. Associated with every process u is a variable $mode.u$ that can take any one of four possible values – out, at, $wait$, or $enter$. Initially, $mode.u = out$, for each u. For convenience, we define the following four predicates.

$$\begin{array}{rcl} out.u & \equiv & mode.u = out \\ at.u & \equiv & mode.u = at \\ wait.u & \equiv & mode.u = wait \\ enter.u & \equiv & mode.u = enter \end{array}$$

Variables $mode.u$ cycle through the values, out, at, $wait$, and $enter$ in that order; the transitions from out to at and from $enter$ to out are determined by the process while the transitions from at to $wait$ and $wait$ to $enter$ are determined by the solution. The intuition behind the four states is as follows. If the state of a process is out then the process is not interested in entering the doorway; if the state of a process is at then the process is interested in entering the doorway (the transition from out to at is made internally by the process); if the state of a process is $wait$ then the process is waiting to enter the doorway (the transition from at to $wait$ is made by the solution and corresponds roughly to the few lines of code that a process typically executes in order to "enter" the doorway in the FCFS mutual exclusion algorithms); if the state of a process is $enter$ then it has entered the doorway (it is in this state that a process typically participates in mutual exclusion).

It is required that the composite program satisfy the following three properties. For all distinct u, v,

- $\{no\ waiting\}$ $\quad at.u \mapsto wait.u$,
- $\{no\ deadlock\}$ $\quad (\exists u :: wait.u) \mapsto (\exists u :: enter.u)$, and
- $\{FCFS\}$ $\quad$ **invariant** $\neg(before.u.v \wedge enter.v)$,

where $before.u.v$ is an auxiliary boolean variable that captures the order in which processes u and v change their states. It is $true$ iff process u transits from at to $wait$ before process v transits from out to at.

The no waiting condition states that a process that is at the doorway, eventually transits to the $wait$ state. This transition corresponds to the few lines of code that a process typically executes in existing solutions to the problem. The no deadlock condition states that if there exists some $waiting$ processes then eventually some process will $enter$ the doorway. The FCFS condition states that if the predicate $before.u.v$ holds, i.e., process u completed its at to $wait$ transition before process v completed its out to at transition, then process v does not $enter$ the doorway until process u goes out of the doorway.

Auxiliary variable $before.u.v$ is defined by the following two assertions.

initially $\neg before.u.v$
$mode.u = x \wedge mode.v = y \wedge before.u.v = b$ **unless**
$\neg(mode.u = x \wedge mode.v = y) \wedge (before.u.v \equiv (wait.u \vee enter.u) \wedge (out.v \vee b))$

The first assertion states that $before.u.v$ is $false$ initially and the second assertion defines how $before.u.v$ changes with each change in the states of processes u and v. It is set to $true$ if the predicate $wait.u \vee enter.u$ holds simultaneously with the predicate $out.v$ (i.e., the state of process u is *wait* or *enter* while the state of process v is *out*). Once $true$, it continues to remain $true$ until process u goes *out* of the doorway, and then it is set to $false$.

5.2 Solution

In order to satisfy the FCFS condition, we design our solution so that if the predicate $before.u.v$ holds then so does the predicate $precedes.u.v$, i.e., if process u transits from *at* to *wait* before process v transits from *out* to *at* (thus, setting $before.u.v$ to $true$) then process u also transits from *grey* to *black* before process v transits from *white* to *grey* (thus, setting $precedes.u.v$ to $true$). We ensure that process v does not *enter* the doorway if its rank is less than the rank of any other process. Then, on account of precedence, and because $before.u.v \Rightarrow precedes.u.v$, if processes u and v are *black* and $before.u.v$ holds, then the rank of v will be less than the rank of u, and consequently, process v will not *enter* the doorway until process u goes *out* of the doorway.

Now, in order to satisfy the no deadlock condition, there should always be at least one *black* process with a maximal rank. This implies that the ranks of the *black* processes should be acyclic. Therefore, we solve this problem using ranker A (ranker Φ does not suffice as it places no global constraints on the ranks of the processes).

Process u manipulates variables $state.u$ and $mode.u$ as follows. When it sets $mode.u$ to *at*, it also sets $state.u$ to *grey* (to request a rank). Then, it waits for $state.u$ to become *black* (this happens eventually due to responsiveness) and sets $mode.u$ to *wait*. Then, it compares its rank with all other processes and sets $mode.u$ to *enter* if no other *black* process has a higher rank. Finally, when it sets $mode.u$ to *out*, it also sets $state.u$ to *white*. Predicate $high.u.v$ is defined to be $\neg(black.v \ \wedge \ r.u \prec r.v)$. The program for process u is as follows.

initially $white.u \wedge out.u$
assign

$mode.u, state.u$	$:=$ *at*, *grey*	**if**	$out.u$
$[]$ $mode.u$	$:=$ *wait*	**if**	$at.u \wedge black.u$
$[]$ $mode.u$	$:=$ *enter*	**if**	$wait.u \wedge (\forall v : v \neq u : high.u.v)$
$[]$ $mode.u, state.u$	$:=$ *out*, *white*	**if**	$enter.u$

end

5.3 Proof of Correctness

A formal proof of correctness for this solution is detailed in [24]; we present a sketch of this proof. Recall that ranker A satisfies the properties of responsiveness, precedence, and acyclicity. These three properties are used to prove that our solution satisfies the no waiting, no deadlock, and FCFS properties. The proof of no waiting is based on responsiveness, the proof of no deadlock is based on responsiveness and acyclicity, and the proof of FCFS is based on precedence.

Our solution satisfies the no waiting condition because of the following reason. A process which is *at* the doorway sets its state to *grey* and waits for this state to become *black*. This happens eventually because of responsiveness of ranker A. Once the state becomes *black*, the process makes the required transition to the *wait* state.

To see why our solution satisfies the no deadlock condition consider the processes which are in the *wait* state at any time. Because of the responsiveness and acyclicity properties of ranker A, eventually one of these processes has a rank no smaller than all other *black* processes, i.e., for some process u, the predicate $high.u.v$ holds for all other v. Therefore, either this process makes a transition to the *enter* state or some other process starts *waiting* with a higher rank replacing u as the process with a rank no smaller than all other *black* processes. But, events of the latter kind are bounded by the number of processes. Therefore, some *waiting* process will eventually go to the *enter* state, thus satisfying the no deadlock condition.

Our solution satisfies the FCFS condition because of the following reason. If $before.u.v$ holds, i.e., a process u transits from *at* to *wait* before another process v transits from *out* to *at*, then process u also requests and obtains a rank before v. As a result, because of precedence, process v has a lower rank than process u, i.e., $r.v \prec r.u$. Therefore, when process v starts *waiting*, the predicate $high.v.u$ will be $false$ (as process u is *black* and $r.v \prec r.u$) and consequently, it does not *enter* the doorway until $before.u.v$ becomes $false$, i.e., process u goes *out* of the doorway.

Observe that our solution uses a coarse grain of atomicity because process u computes the predicates $high.u.v$ for all other processes in one step. This is solely for expository reasons. It is possible to compute these predicates asynchronously one after another. The resulting low atomicity solution appears in [24].

6 The Dining Philosophers

The dining philosophers problem is a paradigm for conflict resolution [7]. The problem consists of a number of processes placed on the vertices of a graph that may wish to enter their critical section from time to time (if this graph is fully connected, we have the mutual exclusion problem) . We have to design a solution that ensures mutual exclusion (i.e., that no two neighboring processes are in their critical sections at the same time) and freedom

from starvation (every process wishing to enter its critical section eventually gets to enter it). We present the specification of the problem in Section 6.1, discuss our solution in Section 6.2, and sketch a proof of its correctness in Section 6.3. Finally, in Section 6.4 we consider a variation of the problem in which a hungry process may timeout and transit directly to the thinking state.

6.1 Problem Specification

We have a set of processes $\{u, v, ..\}$ and a symmetric, irreflexive relation N defined on this set. Processes u and v are said to be *neighbors* iff $N.u.v$ is true. Associated with every process u is a variable $mode.u$ that can take any one of three possible values — t (thinking), h (hungry), or e (eating). Initially, $mode.u = t$, for each u. For convenience, we define the following three predicates.

$$\begin{array}{lcl} t.u & \equiv & mode.u = t \\ h.u & \equiv & mode.u = h \\ e.u & \equiv & mode.u = e \end{array}$$

Variables $mode.u$ cycle through the values, t, h, and e in that order; the transitions from t to h and from e to t are determined by the process while the transition from h to e is determined by the solution. It is given that all eating periods are finite, i.e., $(\forall u :: e.u \mapsto t.u)$.

It is required that the composite program satisfy the following two properties. For all distinct u, v,

- $\{mutual\ exclusion\}$ **invariant** $\neg(e.u \wedge e.v \wedge N.u.v)$, and
- $\{no\ starvation\}$ $h.u \mapsto e.u$.

The mutual exclusion condition states that no two neighboring processes may eat at the same time. The no starvation condition states that every hungry process eventually eats.

6.2 Solution

We use ranks to arbitrate between competing philosophers: a hungry process u eats only if it has a higher rank than all its competing neighbors. Thus, in order to satisfy the mutual exclusion condition, the ranks of any set of neighboring processes should be acyclic (otherwise, two neighbors may be in their critical section at the same time). Also, in order to satisfy the no starvation condition, the ranks of any two neighboring processes should be comparable (otherwise, neither of the processes has a higher rank than the other and consequently, locked in a deadly embrace, they will wait for each other forever). Due to the above two requirements of acyclicity and comparability, we solve this problem using ranker C.

Process u manipulates variables $state.u$ and $mode.u$ as follows. When it sets $mode.u$ to h, it also sets $state.u$ to *grey* (to request a new rank). Then, it waits for $state.u$ to become

black (this happens eventually due to responsiveness). After that, it compares its rank with all neighboring processes and sets $mode.u$ to e if every neighboring process v is either *white* (in which case, on account of precedence, process v will have a lower rank than process u when it becomes hungry) or is *black* and has a lower rank. If process u finds a neighboring process to be *grey*, then, because the rank of a *grey* process may change, it waits until that process becomes non-*grey*. (However, due to responsiveness, a *grey* process eventually becomes *black* and hence, process u is eventually able to proceed.) Finally, when process u sets $mode.u$ to t, it also sets $state.u$ to *white*. The program for process u is as follows.

initially $white.u \wedge t.u$
assign
$mode.u, state.u \quad := \quad h, grey \quad$ **if** $\quad t.u$
$[]\ mode.u \quad := \quad e \quad$ **if** $\quad h.u \wedge black.u \wedge (\forall v : N.u.v : high.u.v)$
$[]\ mode.u, state.u \quad := \quad t, white \quad$ **if** $\quad e.u$
end

6.3 Proof of Correctness

Recall that ranker C satisfies the properties of responsiveness, precedence, acyclicity, and comparability. These four properties are used to prove that our solution satisfies mutual exclusion and no starvation. The proof of mutual exclusion is based on precedence while the proof of no starvation is based on all four properties — responsiveness, precedence, acyclicity, and comparability.

Our solution satisfies the mutual exclusion condition because it satisfies the following invariant (due to precedence of ranker C) — an eating process is *black*, and has a higher rank than all its *black* neighbors. It follows from this invariant that for two neighboring processes to be eating at the same time, each has a rank higher than the other, a condition that is prohibited on account of the definition of ranks. Therefore, two neighboring process do not eat at the same time.

To see why our solution is free from starvation, consider any process u that is hungry and let v be any neighboring process. Because of the responsiveness and comparability properties of ranker C, eventually either the predicate $high.u.v$ holds (in which case process u does not wait for process v) or process v becomes *black* and has a higher rank than u (in which case process u waits for process v). In the latter case, consider the directed *wait-for* graph rooted at u. Because of acyclicity of ranker C, this graph is acyclic, and therefore, has some leaf nodes. Eventually, each process at a leaf node will start eating and will later start thinking (because all eating periods are finite). This reduces the size of the wait-for graph (due to the precedence and responsiveness properties). It follows by induction on the size of this graph that it eventually contains no edges, i.e., process u is not waiting for any other process. When this happens, process u will make a hungry to eating transition, thus ensuring no starvation.

Observe that our solution uses a coarse grain of atomicity because process u computes the predicates $high.u.v$ for all neighboring processes in one step. This is so solely for expository reasons. It is possible to compute these predicates asynchronously one after another. The resulting low atomicity solution appears in [24].

6.4 Dining Philosophers with Timeouts

In the dining philosophers problem we discussed earlier, a hungry process cannot make a direct transition to thinking without eating first (as $h.u$ **unless** $e.u$). In some situations, this may be undesirable and we may want to allow a hungry process that has waited long enough to timeout, and make a direct transition to the thinking state. Such a direct transition is also desirable if we are modeling process failures and assume that a failed process returns to its initial state, the thinking state. These two considerations motivate the problem that we discuss next.

As before, we have a set of processes $\{u, v, ..\}$, a symmetric and irreflexive relation N that defines neighborhood, and a variable $mode.u$ associated with a process u. The set of values that this variable takes and the order in which these values are assumed are as before. Once more, all eating periods are assumed to be finite. It is required that the composite program satisfy the following two properties for all distinct u, v,

- $\{mutual\ exclusion\}$ **invariant** $\neg(e.u \wedge e.v \wedge N.u.v)$, and
- $\{no\ starvation\}$ $h.u \mapsto e.u \vee t.u$.

The mutual exclusion condition is same as before; however, the no starvation condition has been weakened in order to allow the possibility of a timeout.

Contrary to the previous version of the problem (where processes do not timeout), this problem cannot be solved by ranker C. To see why, consider any two neighboring processes u and v. Assume that process u is hungry and *black* and is waiting for the predicate $high.u.v$ (defined to be $white.v \vee (black.v \wedge r.v \prec r.u)$) to become *true* in order to enter its critical section. Also, assume that process v is continuously timing out, i.e., cycling through the states t and h. Then, it is possible that each time process u attempts to compute the predicate $high.u.v$, the state of process v is *grey*, and consequently, $high.u.v$ is $false$. Therefore, process u may remain hungry forever and thus, the no starvation requirement is not met.

The problem arises in the above scenario because of the definition of the predicate $high.u.v$ as by this definition, $high.u.v$ is $false$ if process v is *grey*. So, we need to redefine this predicate so that if process v continuously times out, then the predicate is set to *true* and remains *true* until process u eats. Consequently, we use ranker S, and redefine the predicate $high.u.v$ to be $white.v \vee r.v \prec r.u$ (in other words, the predicate is *true* iff process v is not interested in its critical section or if it has a lower rank than process u).

Then, if process v continuously times out, then this predicate will be set to *true* (due to precedence) and remain *true* until process u eats (due to stability).

The solution is similar to the previous solution; the only difference being that a process that is hungry and *black* can now make a direct transition to the thinking state without eating first. Predicate $high.u.v$ is now defined to be $white.v \vee r.v \prec r.u$. The program for process u is as follows.

initially $white.u \wedge t.u$
assign

$mode.u, state.u$	$:= h, grey$	**if**	$t.u$
$[]$ $mode.u$	$:= e$	**if**	$h.u \wedge black.u \wedge (\forall v : N.u.v : high.u.v)$
$[]$ $mode.u, state.u$	$:= t, white$	**if**	$e.u \vee (h.u \wedge black.u)$

end

The proof of correctness of this solution is similar to the proof of correctness of the previous solution. The proof of mutual exclusion is based on the precedence and stability properties while the proof of no starvation is based on all the five properties of ranker S — responsiveness, precedence, acyclicity, comparability, and stability. The proofs appear in [24].

7 Resource Allocation

The resource allocation problem (or the K-mutual exclusion problem [9]) is a generalization of the mutual exclusion problem [6]. In this problem, there are K identical copies of some resource, and a set of processes that contend for a copy of the resource. We are required to design a solution that ensures mutual exclusion (no more than K processes are accessing the resource at any one time) and no starvation (if a process wants a resource, it eventually gets one).

This problem can be easily solved using a solution to the mutual exclusion problem by maintaining a global queue of all the processes that desire to obtain a copy of the resource. Once a process wishes to obtain a copy, it places itself at the rear of the queue; once it comes within K of the head of the queue, it obtains a copy and when it is finished, it removes itself from the queue. All manipulations of the queue are carried out in exclusion by using the solution to the mutual exclusion problem. However, the above solution suffers from three drawbacks. The first drawback is the use of a global data structure. The second drawback is the loss of concurrency as there may be be a lot of unnecessary contention for the queue. The third drawback is that even if one process fails in its critical section, the whole system comes to a halt. In this chapter, we present a solution with none of these drawbacks.

We present the specification of the problem in Section 7.1, discuss our solution in Section 7.2, and sketch a proof of its correctness in Section 7.3.

7.1 Problem Specification

The specification of this problem is similar to the specification of the dining philosophers problem. However, now the *neighbors* relation N is complete, i.e., any two processes are neighbors. The definition of variable *mode.u* and the values it takes — t, h, and e, is as before. Once more all eating periods are finite, i.e., $(\forall u :: e.u \mapsto t.u)$.

It is required that the composite program satisfy the following two properties. For every u,

- $\{$*mutual exclusion*$\}$ **invariant** $(\#u :: e.u) \leq K$, and
- $\{$*no starvation*$\}$ $h.u \mapsto e.u$.

The mutual exclusion condition states that at any time at most K processes are accessing the shared resource. (The notation $(\#u :: e.u)$ denotes the number of processes for which the predicate $e.u$ holds.) The no starvation condition states that every process that is waiting to access a resource eventually obtains a copy of the resource. Observe that when $K = 1$, we have the usual mutual exclusion problem.

7.2 Solution

We solve this problem by using ranker S for the following reason. First, recall that the mutual exclusion problem is solved using ranker C; therefore, as the resource allocation problem is a generalization of the mutual exclusion problem, the resource allocation problem is solved by a ranker at least as powerful as ranker C. Next, we argue the need for stability, and hence the need for ranker S. Assume to the contrary that we solve this problem using ranker C (i.e., a ranker without stability). Consider a process u that is waiting to access a copy of the resource. Because of the absence of stability, process u can make no assertions about the rank of another process if the state of the other process is *grey*. Therefore, process u has to wait until it observes the state of the other processes to be *white* or *black*. But, because up to K different processes may be accessing the resource at any given time, it is possible for all other processes to be continuously cycling through the states t, h, and e and therefore, it is possible that process u always observes the state of other processes to be *grey* (i.e., when they are obtaining a rank). When this happens, process u waits forever, thus violating the no starvation condition. It is due to the above reason that we require stability (and hence, ranker S) to solve this problem. The above scenario does not occur if we use ranker S because if another process v continuously keeps cycling through its states while process u is waiting, then eventually process u will have a higher rank than process v (due to precedence) and will continue to have a higher rank (due to stability) until it accesses the resource.

Process u manipulates variables *state.u* and *mode.u* as follows. When it sets *mode.u* to h, it also sets *state.u* to *grey* (to request a new rank). Then, it waits for *state.u* to become *black* (this happens eventually due to responsiveness). After that, it compares its

rank with all other processes and sets $mode.u$ to e if the number of non-*white* processes with a higher rank is less than K. Finally, when it sets $mode.u$ to t, it also sets $state.u$ to *white*. Predicate $high.u.v$ is defined, as in Section 6.4, to be $white.v \vee r.v \prec r.u$. In other words, the predicate $high.u.v$ is *true* iff process v is not interested in accessing the resource (i.e., it is *white*), or if it has a lower rank than process u. The program for process u is as follows.

initially $white.u \wedge t.u$
assign
$mode.u, state.u \quad := \quad h, grey \quad \textbf{if} \quad t.u$
$[]\ mode.u \quad := \quad e \quad \textbf{if} \quad h.u \wedge black.u \wedge (\#v :: v \neq u \wedge \neg high.u.v) < K$
$[]\ mode.u, state.u \quad := \quad t, white \quad \textbf{if} \quad e.u$
end

7.3 Proof of Correctness

Recall that ranker S satisfies the properties of responsiveness, precedence, acyclicity, comparability, and acyclicity. These five properties are used to prove that our solution satisfies mutual exclusion and no starvation. The proof of mutual exclusion is based on the precedence, acyclicity, comparability, and stability properties while the proof of no starvation is based on all the five properties of ranker S — responsiveness, precedence, acyclicity, comparability, and stability.

Our solution satisfies the mutual exclusion condition because of the following invariant (due to precedence and stability) — an eating process is *black*, and the number of *black* processes with a rank higher than an eating process is less than K. Now, due to the total order on the ranks (ensured by acyclicity and comparability), if more than K processes are eating at the same time, then one of them has a rank lower than K other processes. But, this contradicts the invariant, and therefore, at most K processes can eat at any one time.

To see why our solution is free from starvation, consider any process u that is hungry and let v be any other process. Because of the responsiveness and comparability properties of ranker S, eventually either the predicate $high.u.v$ holds (in which case process u does not wait for process v) and continues to hold (due to stability), or process v becomes *black* and has a higher rank than u (in which case process u waits for process v). In the latter case, consider the directed *wait-for* graph rooted at u. Because of acyclicity of ranker S, this graph is acyclic, and therefore, has some leaf nodes. Eventually, each process at a leaf node will start eating and will later start thinking (because all eating periods are finite). This reduces the size of the wait-for graph (due to precedence). It follows by induction on the size of the wait-for graph that it eventually contains no edges, i.e., process u is not waiting for any other process. When this happens, process u will make a hungry to eating transition, thus ensuring no starvation.

Observe that our solution uses a coarse grain of atomicity because process u computes

the predicates $high.u.v$ for all other processes in one step. This is so solely for expository reasons. It is possible to compute these predicates asynchronously one after another. The resulting low atomicity solution appears in [24].

8 Implementation of Rankers

In this section we present implementations for the four rankers — Φ, A, C, and S, discussed earlier. In all of these implementations program *ranker* is designed to be a composition of submodules *ranker.u*, one for every process. The submodule *ranker.u* is responsible for ranking process u and it does not modify the interface variables, *state.v* and *r.v*, of any other user process. The implementations that are presented here are highly concurrent as they use a fine grain of atomicity (i.e., every statement reads or writes at most one shared variable) and are "wait-free" (i.e., the ranking of a process is done within a bounded number of steps [1, 10]).

8.1 Implementation of Ranker Φ

Recollect that ranker Φ is required to satisfy two properties — responsiveness, a local property over processes, and precedence, a pairwise property over processes. Thus, this ranker is not required to satisfy any global properties and consequently, this ranker composes, i.e., given two ranker implementations for m and n processes, we can compose them to obtain a ranker implementation for $m + n$ processes. Based on this observation, we define an implementation for 2 processes; the implementation for any arbitrary number of processes is obtained by repeated composition.

Let u and v be two user processes. As stated earlier, we design program *ranker* to be the composition of two submodules, *ranker.u* and *ranker.v*, which we define next. Variables *r.u* and *r.v* take on integer values and the ranking relation $\prec$ is defined as follows:

$r.u \prec r.v \equiv r.u < r.v$,

where $<$ is the less-than relation on integers. Because $<$ is a total order, $\prec$ is irreflexive and does not contain cycles of length two.

The design of program *ranker.u* is simple — when process v turns *grey*, variable *r.v* is assigned a value less than *r.u*, and then *state.v* is set to *black*. Observe that if *precedes.u.v* holds and processes u and v are *black*, then variable *r.v* contains a value less than *r.u*, and consequently, $r.v \prec r.u$, thus satisfying the precedence condition. Program *ranker.u* is given below (program *ranker.v* can be obtained by a simple substitution). Variable *b.u* is a program counter, variable *t.u* stores intermediate values, and variable *checked.u.v* is a boolean and is *true* iff the intermediate value for process v has been computed.

initially $r.u = 0 \wedge b.u = 0 \wedge \neg checked.u.v$
assign

$b.u$	$:= 1$	**if**	$b.u = 0 \wedge grey.u$

$[]\ t.u, checked.u.v \quad := \quad r.v - 1, true \quad \textbf{if} \quad b.u = 1 \wedge \neg checked.u.v$
$[]\ r.u, b.u \quad := \quad t.u, 2 \quad \textbf{if} \quad b.u = 1 \wedge checked.u.v$
$[]\ state.u, b.u, checked.u.v \quad := \quad black, 0, false \quad \textbf{if} \quad b.u = 2$
end

Program *ranker.u* sets $b.u$ to 1 if *state.u* is *grey*. Then, it reads the rank of process v and sets variable $t.u$ to the value $r.v - 1$. After that, the rank of process $r.u$ is set to $t.u$. Finally, the state of process u is set to *black*, and variables $b.u$ and *checked.u.v* are reset. Observe that we have used a very fine grain of atomicity as each statement reads or writes at most one shared variable. Also, the above program is "wait-free' as the ranking of process u is completed within four steps.

When the above implementation for two processes is extended to an arbitrary number of processes, variables $r.u$ and $t.u$ become arrays with one component per process, and the definition of $\prec$ is changed to

$r.u \prec r.v \quad \equiv \quad r.u.v < r.v.u,$

where $r.u.v$ denotes the component of $r.u$ corresponding to process v.

8.2 Implementation of Ranker A

Because ranker A satisfies acyclicity, a global property over processes, this ranker does not compose. However, the implementation is similar to that for ranker Φ presented earlier. Variable $r.u$ takes on integer values and the ranking relation is defined as before:

$r.u \prec r.v \quad \equiv \quad r.u < r.v.$

When process v turns *grey*, variable $r.v$ is assigned a value less than the rank of *all other black* processes. Consequently, if predicate *precedes.u.v* holds, then variable $r.v$ is assigned a value less than $r.u$ (note that *precedes.u.v* $\Rightarrow$ *black.u*) during the ranking process. Thus, when process v turns *black*, the rank of process v is less than that of process u, as required by the precedence condition. The proof of acyclicity follows due to the acyclicity of relation $<$.

In the code for *ranker.u* presented below, variables $b.u, t.u.v$, and $c.u.v$ are all local variables; variables $b.u$ and $c.u.v$ are program counters and variable $t.u.v$ stores the intermediate value with respect to process v. When process u turns *grey*, variable $b.u$ is set to 1. Then, the state of all other processes is checked; if the state of v is non-*black* then $t.u.v$ is set to 0, otherwise, $t.u.v$ is set to $r.v - 1$ (this takes place in two steps in order to use a fine grain of atomicity). After all the intermediate values $t.u.v$ have been computed, $r.u$ is set to the minimum of those values. Finally, the state of u is set to *black* and all program counters are reset. In the program below, variable v ranges over all processes distinct from u and the value $(min\ i :: x.i)$ refers to the minimum value among the $x.i$'s.

initially $r.u = 0 \wedge b.u = 0 \wedge (\forall v :: c.u.v = 0)$
assign
$b.u \quad := \quad 1 \quad \textbf{if} \quad b.u = 0 \wedge grey.u$

$$
\begin{array}{lll}
[] \langle [] v :: c.u.v, t.u.v & := 2, 0 & \textbf{if} \quad b.u = 1 \wedge c.u.v = 0 \wedge \neg black.v \\
\quad c.u.v & := 1 & \textbf{if} \quad b.u = 1 \wedge c.u.v = 0 \wedge black.v \\
\quad c.u.v, t.u.v & := 2, r.v - 1 & \textbf{if} \quad c.u.v = 1 \\
\rangle & & \\
[] \ b.u, r.u & := 2, (min\ v :: t.u.v) & \textbf{if} \quad b.u = 1 \wedge (\forall v :: c.u.v = 2) \\
[] \ b.u, state.u & := 0, black & \textbf{if} \quad b.u = 2 \\
\quad \|\langle \| v :: c.u.v & := 0 & \textbf{if} \quad b.u = 2 \rangle
\end{array}
$$

end

Once more, we have used a very fine grain of atomicity (as each statement reads or writes at most one shared variable), and program *ranker.u* is "wait-free" (as the ranking of a process is completed within $3n$ steps where n is the number of processes). The proofs of responsiveness and precedence are similar to the proofs in the previous implementation and as stated earlier, the proof of acyclicity follows from the acyclicity of the relation $<$ on integers. □

8.3 Implementation of Ranker C

The programs for this implementation are as in the last implementation; only the ranking relation $\prec$ is defined differently. It is now defined to be the lexicographic ordering of the previous ranking relation and some fixed total order q, i.e., for all distinct u, v,

$r.u \prec r.v \quad \equiv \quad r.u < r.v \ \vee \ (r.u = r.v \wedge q.u.v)$

The proofs of responsiveness and precedence are as in the previous implementations. Properties acyclicity and comparability follow from the total order of $\prec$.

8.4 Implementation of Ranker S

This implementation is similar to the previous implementation. The ranking relation and the program variables are exactly the same; the only difference is that variable $r.u$ is now ensured to be monotonically decreasing (in order to satisfy stability). The program is as follows.

initially $r.u = 0 \wedge b.u = 0 \wedge (\forall v :: c.u.v = 0)$

assign

$$
\begin{array}{lll}
b.u & := 1 & \textbf{if} \quad b.u = 0 \wedge grey.u \\
[] \langle [] v :: c.u.v, t.u.v & := 2, 0 & \textbf{if} \quad b.u = 1 \wedge c.u.v = 0 \wedge \neg black.v \\
\quad c.u.v & := 1 & \textbf{if} \quad b.u = 1 \wedge c.u.v = 0 \wedge black.v \\
\quad c.u.v, t.u.v & := 2, r.v - 1 & \textbf{if} \quad c.u.v = 1 \\
\rangle & & \\
[] \ b.u, r.u & := 2, min(r.u, (min\ v :: t.u.v)) & \textbf{if} \quad b.u = 1 \wedge (\forall v :: c.u.v = 2) \\
[] \ b.u, state.u & := 0, black & \textbf{if} \quad b.u = 2 \\
\quad \|\langle \| v :: c.u.v & := 0 & \textbf{if} \quad b.u = 2 \rangle
\end{array}
$$

end

The proofs of responsiveness, precedence, acyclicity, and comparability are as in the previous implementation. The proof of stability follows from the fact that each variable $r.u$ is monotonically decreasing.

9 Concluding Remarks

Our aim is to study synchronization problems by defining the interface between ranker implementations and ranker applications. For this purpose, we identified five properties of rankers — responsiveness, precedence, acyclicity, comparability, and stability, and obtained a hierarchy of four rankers based on these properties. We have showed that such a separation of concerns leads to simple and modular solutions to various synchronization problems.

The first evidence of modularity comes from the fact that we solve all synchronization problems by the same strategy; as a result seemingly different problems have similar solutions. As stated earlier, any solution consists of the following three steps. First, a process that wishes to synchronize sets its state to *grey* and waits for a new rank. Then, on obtaining a rank (signaled by the state of the process becoming *black*) the process compares its rank with those of competing processes and proceeds accordingly. Finally, when the process completes its critical action, it informs the ranker by setting its state to *white*. We used the above strategy to solve a number of problems in Sections 5, 6, and 7.

Another evidence of modularity comes from the fact that small modifications in the problem description lead to only minor modifications in the solution. For example, when we added timeout to the dining philosophers problem (Section 6.4), the resulting solution was very similar to the original problem; in fact, the only difference was that now we used ranker S instead of ranker C, and we added a disjunct to the condition under which a philosopher makes a transition to the thinking state.

Finally, the modularity of our solutions is also reflected in the modularity of the proofs; solutions to similar problems share a considerable amount of proof effort. For example, the proofs of mutual exclusion for the dining philosophers problem without timeouts and the dining philosophers problem with timeouts are almost the same. The same assertion can also be made about the rest of the proofs. In summary, the abstraction of rankers allows us to design simple modular solutions to synchronization problems.

We examined alternative formulations of the property of precedence and discovered that it is impossible to implement rankers that satisfy some of those formulations; we discuss two of these next.

Suppose we require a process to have a higher rank if it turned *grey* before another process, i.e., if process u turned *grey* before process v then process u has a higher rank. Such a ranker cannot be implemented because the state transitions from *white* to *grey* occur in the user processes asynchronously with the ranker and therefore, it is not possible for a

ranker implementation to detect every occurrence of a state in which one process is *grey* and the other is *white*.

Consider a different formulation of precedence. Suppose we require that a process has a higher rank if it turned *black* before another process. In other words, if process u turned *black* while another process v is not *black*, then process u has a higher rank. Also suppose that we require the ranker programs to be wait-free and use a fine grain of atomicity. Such a ranker cannot be implemented because it can be used to solve the binary election problem which has been shown to be impossible in [1]. These experiments have led us to the current formulation of precedence which is both useful and implementable, as demonstrated in Section 8.

Acknowledgement: We are grateful to Edsger W. Dijkstra, Jay Misra, and members of the Distributed Systems Discussion Group at UT Austin for reading earlier versions of this paper. Our presentation has improved considerably from their comments.

References

[1] Anderson, J. H., and M. G. Gouda. "The Virtue of Patience — Concurrent Programming With and Without Waiting," work in progress.

[2] Brinch Hansen, P., "Concurrent Programming Concepts," *ACM* Computing Surveys, 5, 1973, pp. 223 – 245.

[3] Chandy, K. M., and J. Misra, "The Drinking Philosophers Problem," *ACM* Transactions on Programming Languages and Systems, 6:4, October 1984, pp. 144 – 156.

[4] Chandy, K. M., and J. Misra, *Parallel Program Design: A Foundation*, Reading, Massachusetts: Addison-Wesley, 1988.

[5] Courtois, P. J., F. Heymans, and D. L. Parnas, "Concurrent Control with 'Readers' and 'Writers'," Communications of the *ACM*, 14:10, October 1971, pp. 667 – 668.

[6] Dijkstra, E. W., "Solution of a Problem in Concurrent Program Control," *C.ACM*, 8:9, Sept. 1965, pp. 320 – 322.

[7] Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes, *Operating Systems Techniques*, eds. C. A. R. Hoare and P. J. Perrott, London: Academic Press, 1972, pp. 72 – 93.

[8] Fischer, M. J., N. Lynch, J. E. Burns, and A. Borodin, "Distributed FIFO Allocation of Identical Resources Using Small Shared Variables," *ACM* Transactions on Programming Languages and Systems, 11:1, January 1989, pp. 90 – 118.

[9] Herlihy, M., "Wait-Free Implementations of Concurrent Objects," Proceedings of the Sixth ACM Symposium on the Principles of Distributed Computing, 1988, pp. 276 – 290.

[10] Israeli, A., and M. Li, "Bounded Time-stamps," Proceedings of Twenty-eighth Annual Symposium on Foundations of Computer Science, 1987, pp. 371 – 382.

[11] Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem," Communications of the *ACM*, 17:8, August 1974, pp. 453 – 455.

[12] Lamport, L., "Time, Clock, and the Ordering of Events in a Distributed System," Communications of the *ACM*, 21:7, July 1978, pp. 558 – 565.

[13] Lamport, L., "The Mutual Exclusion Problem: Part II — Statement and Solutions," Journal of the *ACM*, 33:2, July 1986, pp. 327 – 348.

[14] Lamport, L., "On Interprocess Communication, Parts I and II," Distributed Computing, Vol. 1, 1986, pp. 77 – 101.

[15] Misra, J., "Specification of Objects: An Example," unpublished manuscript.

[16] Peterson, G. L., "Myths About the Mutual Exclusion Problem," Information Processing Letters, 12:3, June 1981, pp. 115 – 116.

[17] Peterson, G. L., and J. E. Burns, "Concurrent Reading While Writing II: The Multi-writer Case," Proceedings of the Twenty-eighth Annual Symposium on Foundations of Computer Science, 1987, pp. 383 – 392.

[18] Reed, D. P., and R. K. Kanodia, " Synchronization with Eventcounts and Sequencers," Communications of the *ACM*, 22:2, Feb. 1979, pp. 115 – 123.

[19] Singh, A. K., J. H. Anderson, and M. G. Gouda, "The Elusive Atomic Register Revisited," Proceedings of the Sixth ACM symposium on Principles of Distributed Computing, 1987, pp. 206 – 221.

[20] Singh, A. K., "Ranking in Distributed Systems," Ph.D. Dissertation, in preparation, The University of Texas at Austin, Austin, Texas.