A Modular Drinking Philosophers Algorithm

Jennifer L. Welch* Nancy A. Lynch†

June 1, 2001

Abstract

A variant of the drinking philosophers algorithm of Chandy and Misra is described and proved correct in a modular way. The algorithm of Chandy and Misra is based on a particular dining philosophers algorithm and relies on certain properties of its implementation. The drinking philosophers algorithm presented in this paper is able to use an arbitrary dining philosophers algorithm as a subroutine; nothing about the implementation needs to be known, only that it solves the dining philosophers problem. An important advantage of this modularity is that by substituting a more time-efficient dining philosophers algorithm than the one used by Chandy and Misra, a drinking philosophers algorithm with $O(1)$ worst-case waiting time is obtained, whereas the drinking philosophers algorithm of Chandy and Misra has $O(n)$ worst-case waiting time (for n philosophers). Careful definitions are given to distinguish the drinking and dining philosophers problems and to specify varying degrees of concurrency.

*Department of Computer Science, Texas A&M University, College Station, TX 77843. Much of this work was performed while this author was at the Laboratory for Computer Science, Massachusetts Institute of Technology, supported by the Advanced Research Projects Agency of the Department of Defense under contract N00014-83-K-0125, the National Science Foundation under grants DCR-83-02391 and CCR-86-11442, the Office of Army Research under contract DAAG29-84-K-0058, and the Office of Naval Research under contract N00014-85-K-0168. This author was also supported in part by NSF grant CCR-9010730, an IBM Faculty Development Award, and NSF Presidential Young Investigator Award CCR-9158478.

†Laboratory for Computer Science, MIT, Cambridge, MA 02139. This author was supported by the Office of Naval Research under contract N00014-91-J-1046, the Advanced Research Projects Agency of the Department of Defense under contract N00014-89-J-1988, and the National Science Foundation under grant CCR-89-15206.

1 Introduction

We present a modular description and proof of correctness for an algorithm to solve the drinking philosophers problem in a message-passing distributed system. Our algorithm uses an arbitrary solution to the dining philosophers problem as a subroutine; by using a time-efficient subroutine, one can obtain a drinking philosophers algorithm with $O(1)$ worst-case waiting time. Careful definitions are given to distinguish the drinking and dining philosophers problems and to specify varying degrees of concurrency.

The drinking philosophers problem is a dynamic variant due to Chandy and Misra [2] of the dining philosophers problem, a much-studied resource allocation problem. In the original dining philosophers problem of Dijkstra [4], five philosophers (processes) are arranged in a ring with one fork (resource) between each pair of neighbors, and in order to eat (do work), a philosopher must have exclusive access to both of its adjacent forks. A more general version of the problem allows any number of processes and puts no restrictions on which processes share resources. In the drinking philosophers problem, for each process there is a maximum set of resources that it can request, and each time a process wishes to do some work, it may request an arbitrary subset of its maximum set.

Our drinking philosophers algorithm is a variant of the one of Chandy and Misra [2]. Their algorithm is based on a particular dining philosophers algorithm and relies on certain properties of its implementation. Our drinking philosophers algorithm is able to use an arbitrary dining philosophers algorithm as a subroutine; nothing about the implementation needs to be known, only that it solves the dining philosophers problem. We show that in a system of n philosophers the maximum waiting time for a drinking philosopher to enter its critical region is dominated by the maximum waiting time for a dining philosopher to enter its critical region in the subroutine. Thus, by replacing the dining philosophers algorithm of [2], which has waiting time $O(n)$, with a dining philosophers algorithm such as the one of Lynch [7], which has waiting time independent of n, we obtain a more time-efficient drinking philosophers algorithm.

We provide definitions that distinguish the drinking and dining philosophers problem and that specify varying degrees of concurrency. We use the input-output automaton model of Lynch and Tuttle [8], which is useful for stating properties that concern the infinite behavior of a system, such as no-lockout, and which supports modular algorithm design and verification.

Other work on the drinking philosophers problem includes the following. The algorithm in [2] is proven correct assuming a strong fairness property on the execution of actions. Murphy and

Shankar [9] consider weaker fairness properties and show what modifications had to be made to preserve correctness. The resulting algorithm is very similar to the original and has the same performance.

Singh and Gouda [11, 12] study abstract properties of synchronization problems. Their work shows that the same set of abstract properties are required of both drinking and dining philosophers, explaining somewhat *why* a dining philosophers subroutine can solve the drinking philosophers problem.

Ginat, Shankar and Agrawala [5] propose a drinking philosophers algorithm that solves the problem directly without using a dining philosophers subroutine. As a result, it is more message-efficient, but it requires unbounded counters. Its time performance is $O(n)$.

Awerbuch and Saks [1] define a general "dynamic job scheduling" problem, which includes drinking philosophers as a special case. In this model, a process is created whenever a job to be executed enters the system. The process is initially given an identifier for the job and the set of identifiers of currently conflicting jobs; it is responsible for determining when the job can begin. The time performance is $O(\delta^2 \cdot \log U)$, where δ is the maximum number of conflicting jobs in the system at one time, and U is the size of the set from which job identifiers are drawn.

Choy and Singh [3] have extended the work of [1] to improve performance and to include discussion of fault-tolerance.

In Section 2, the dining philosophers and drinking philosophers problems are defined. In Section 3, we describe our algorithm, as an automaton. Section 4 contains the proof of correctness of our algorithm, and in Section 5 we analyze its performance with respect to various complexity measures. Section 6 contains our conclusions. Earlier versions of this work appeared in [13, 14].

2 Problem Statement

There are n user processes in the system being modeled, and each one needs, at various times, some of the system resources. Only one user at a time may have access to any one resource. Each user's states are partitioned into four regions. In its *trying region*, the user is vying for access to its required resources. Once the resources are obtained, the user may enter its *critical region*. When the user is through with the resources, it enters its *exit region*, which usually involves some "cleaning up" activities. Otherwise, the user is in its *remainder region*. The user cycles through these four regions. In the dining philosophers problem, each user (or philosopher) always requests

the same set of resources. In the drinking philosophers problem, each user can request a different set of resources each time it enters its trying region.

A resource allocation algorithm decides which user gets which resources at which time; thus, it supplies the code for the trying and exit regions. A distributed resource allocation algorithm consists of one component for each user; the components communicate with each other by message passing.

We describe algorithms for two resource allocation problems, dining philosophers and drinking philosophers, as input-output automata ([8]); see Section 2.1 for a brief summary. We imagine an automaton that, given input from some number of users informing the automaton of their desire to gain or give up a set of resources, determines when particular users are allowed to enter their critical and remainder regions, indicated by output actions.

Let $\mathcal{S}$ be a finite non-empty set of resources. Define an *n-user resource requirement* to be a collection of n sets S_i, $1 \leq i \leq n$, such that each S_i is a non-empty subset of $\mathcal{S}$, and no resource is in more than two S_i's. The last restriction makes the algorithm much simpler to describe and reason about, but is not substantive: if a resource is shared by k users, then it can be represented by $k(k-1)/2$ virtual resources, one shared between each pair of the original k users; to gain the "real" resource, a user must gain the $k-1$ virtual resources shared with it.

In the context of the dining philosophers problem, resources will be referred to as *forks*; in the context of the drinking philosophers problem, resources will be referred to as *bottles*. (This terminology comes from [2].)

2.1 Model

In this subsection, we briefly describe the input-output automaton model ([8]), as simplified for our purposes.

Each system component is modeled by an automaton. The automaton is a state machine whose state transitions are labeled with *actions*. If there is a transition from a state labeled with an action, then that action is *enabled* in that state. Actions are partitioned into *input* actions, *output* actions, and *internal* actions. The input and output actions model communication with the outside world. Since the component has no control over when inputs occur, each input action is enabled in every state. Internal actions are private to the component, i.e., not visible to its environment.

An *execution* of an automaton is an alternating sequence of states and actions, beginning with an initial state, in which each action is enabled in the previous state and each state change correctly

reflects the transition relation for the intervening action. An execution is *fair* if every output or internal action that is continuously enabled eventually occurs. Informally, an execution is fair if the automaton eventually gets to perform a pending output or internal action (and is not, say, swamped with inputs). We will require liveness properties only of fair executions.

The system as a whole is also modeled by an automaton, the automaton resulting from the *composition* of the components. In order for the composition to be defined, each action must be shared by at most two automata, and then the action must be an input of one and an output of the other. The state set of the composition is the Cartesian product of the state sets of the component automata. There is a transition from state s' of the composition to state s labeled with action π if and only if (1) π is enabled in each component of s' that corresponds to an automaton with that action, and (2) each component in s correctly reflects the corresponding transition for π (or is unchanged if the corresponding automaton lacks π). Each action of the composition retains its previous classification as input, output, or internal, except that an action that is input to one component and output to another becomes internal.

2.2 Dining Philosophers

Fix an n-user fork requirement $\mathcal{F} = \{F_i : 1 \leq i \leq n\}$. The following definitions are all made relative to this fork requirement.

An automaton is a *dining philosophers algorithm* if it satisfies the following five conditions.

1. Its input actions are $\{T_i, E_i : 1 \leq i \leq n\}$. T_i means that user i wants to enter its trying region, E_i means that user i wants to enter its exit region.

2. Its output actions are $\{C_i, R_i : 1 \leq i \leq n\}$. C_i means that user i may enter its critical region, R_i means that user i may enter its remainder region.

These input and output actions are called the *dining actions.*

We are only interested in executions in which the environment (which generates the inputs) and the automaton (which generates the outputs) cooperate so that the dining actions for each i cycle through T_i, C_i, E_i, R_i. Formally, an execution e of an automaton is *dining-well-formed* if for all i, the subsequence of e restricted to dining actions conforms to the pattern $T_iC_iE_iR_iT_iC_iE_iR_i\ldots$. An automaton *preserves dining-well-formedness* if for all executions e' and e of the automaton, where e is the result of extending e' by one output action, if e' is dining-well-formed, then e is

dining-well-formed. When we use a dining philosophers algorithm as a subroutine, we will make sure that its environment—the caller—preserves dining-well-formedness.

3. (Dining-well-formedness) The automaton preserves dining-well-formedness.

4. (Exclusion) In any dining-well-formed execution, for all i and j with $i \neq j$ and $F_i \cap F_j \neq \emptyset$, if an occurrence of C_i precedes an occurrence of C_j then E_i occurs between the C_i and C_j. This is the property that guarantees exclusive access to the resources.

5. (No-lockout) In any fair dining-well-formed execution, if for all i, every occurrence of C_i is followed by an occurrence of E_i, then for all i, every occurrence of T_i is followed by an occurrence of R_i. This property means that no diner is stuck in either its trying or exit region, assuming no diner is stuck in its critical region.

2.3 Drinking Philosophers

Fix an n-user bottle requirement $\mathcal{B} = \{B_i : 1 \leq i \leq n\}$. The following definitions are made relative to this bottle requirement; most are analogous to those in Section 2.2. A new condition is introduced to distinguish the drinking philosophers problem from the dining philosophers problem, as will be discussed.

An automaton is a *drinking philosophers algorithm* if it satisfies the following six conditions.

1. Its input actions are $\{T_i(B), E_i(B) : 1 \leq i \leq n, B \subseteq B_i, B \neq \emptyset\}$. $T_i(B)$ means that user i wants to enter its trying region with set of resources B, etc. B must be a nonempty subset of B_i, the maximum set of resources that user i can ever request.

2. Its output actions are $\{C_i(B), R_i(B) : 1 \leq i \leq n, B \subseteq B_i, B \neq \emptyset\}$. $C_i(B)$ means that user i may enter its critical region with set of resources B, etc.

These input and output actions are called the *drinking actions*.

We are only interested in executions in which the environment and the automaton cooperate so that for all i, the drinking actions for i cycle through groups, where each group is of the form $T_i(B)$, $C_i(B)$, $E_i(B)$, $R_i(B)$ for a fixed value of B, and different groups may involve different values of B. Formally, an execution e of an automaton is *drinking-well-formed* if for all i, the subsequence of e restricted to dining actions conforms to the pattern $T_i(B)C_i(B)E_i(B)R_i(B)T_i(B')C_i(B')E_i(B')R_i(B')\ldots$.

An automaton *preserves drinking-well-formedness* if for all executions e' and e where e is the result of extending e' by one output action, if e' is drinking-well-formed, then e is drinking-well-formed.

3. (Drinking-well-formedness) The automaton preserves drinking-well-formedness.

4. (Exclusion) In any drinking-well-formed execution, for all i, j, B, and B' with $i \neq j$ and $B \cap B' \neq \emptyset$, if an occurrence of $C_i(B)$ precedes an occurrence of $C_j(B')$ then $E_i(B)$ occurs between the $C_i(B)$ and $C_j(B')$. This is the property that guarantees exclusive access to the resources.

5. (No-lockout) In any fair drinking-well-formed execution, if for all i and B, every occurrence of $C_i(B)$ is followed by an occurrence of $E_i(B)$, then for all i and B, every occurrence of $T_i(B)$ is followed by an occurrence of $R_i(B)$. This property means that no drinker is stuck in either its trying or exit region, assuming no drinker is stuck in its critical region.

(Another condition that might be of interest for both the dining and drinking philosophers problem is that no user is ever stuck in its exit region, no matter what happens in the execution. Our algorithm satisfies this condition as well.)

According to the definitions presented so far, any dining philosophers algorithm is also a drinking philosophers algorithm: if the set of resources for each drinker i is identified with the total set of resources that could ever be requested by i, then a dining philosophers algorithm will satisfy all the conditions for drinking philosophers. Yet this is not very satisfying, since if drinkers i's and j's current resource requirements are disjoint, they should be able to enter their critical regions simultaneously, even if their potential resource requirements intersect. We should be able to get "more concurrency" from a drinking philosophers algorithm than from a dining philosophers algorithm.

One way to formalize this requirement is to require the following property: if drinker i requests the set of resources B at some point, and no other drinker wants or uses any of the resources in B from that point onwards until i gets to use B, then i cannot be stuck in its trying region, *even if other drinkers keep other resources forever.* To state this "more-concurrent" property, we need the following definition. Given i, B, and an occurrence of $T_i(B)$ in a drinking-well-formed execution, the occurrence of $T_i(B)$ is *non-overlapping* if for all $j \neq i$ and all B' that intersect B, (i) every preceding occurrence of $T_j(B')$ is followed by an $E_j(B')$ that also precedes the $T_i(B)$, and (ii) every following occurrence of $T_j(B')$ follows a $C_i(B)$ that also follows the $T_i(B)$.

6. (More-concurrent) In any fair drinking-well-formed execution, for all i, B, and all occurrences of $T_i(B)$, if the occurrence of $T_i(B)$ is non-overlapping, then the $T_i(B)$ is followed by an occurrence of $C_i(B)$.

Other possibilities for defining "more concurrency" are discussed in the conclusion, including the stronger condition that i is not stuck in its trying region as long as no other drinker *uses* any of the resources in B while i is trying.

3 Drinking Philosophers Algorithm

In this section we describe an automaton $Drink(\mathcal{B})$ to solve the drinking philosophers problem for the n-user bottle requirement $\mathcal{B} = \{B_i : 1 \leq i \leq n\}$, in a message-passing distributed system. It is created by composing the following automata:

- D_i, for $1 \leq i \leq n$, the part of the algorithm for user i;
- Net, a reliable communication network that delivers messages between any pair of users in FIFO order; and
- any automaton $Dine(\mathcal{B})$ that is a dining philosophers algorithm for $\mathcal{B}$ (the subroutine).

Net is an automaton whose state contains a FIFO queue $channel_{ij}$ for all users i and j, holding the messages sent from i to j but not yet received, and whose input actions are $Send_i(m, j)$ and output actions are $Receive_i(m, j)$ for all users i and j and every message m. When $Send_i(m, j)$ occurs, m is enqueued in $channel_{ij}$. $Receive_j(m, i)$ can occur when m is at the head of $channel_{ij}$ and results in m being dequeued.

The heart of our algorithm is the D_i automata. First we describe the algorithm informally and then we present the D_i automata.

When D_i enters its trying region needing a certain set of resources, it sends requests for those that it needs but lacks. Recipient D_j of a request satisfies the request unless D_j currently also wants the resource or is already using it. In these two cases, D_j defers the request and satisfies it once D_j is finished using the resource.

In order to prevent drinkers from deadlocking, a dining philosophers algorithm is used as a subroutine. The "resources" manipulated by the dining subroutine are priorities for the "real"

resources (there is one dining resource for each drinking resource). As soon as D_i is able to do so in its drinking trying region, it enters its dining trying region, that is, it tries to gain priority for its maximum set of resources. If D_i ever enters its dining critical region while still in its drinking trying region, it sends demands for needed bottles that are still missing. A recipient D_j of a demand must satisfy it even if D_j wants the resource, unless D_j is using the resource. In that case, D_j defers the request and satisfies it when D_j is through using the resource.

Once D_i is in its dining critical region, we can show that it eventually receives all its needed resources and never gives them up. Then it may enter its drinking critical region. Once D_i enters its drinking critical region, it may relinquish its dining critical region, since the benefits of having the priorities are no longer needed. Doing so allows some extra concurrency: even if D_i stays in its drinking critical region forever, other drinkers needing other resources can continue to make progress.

A couple of points about the code deserve explanation. We can show that when a request is received, the resource is always at the recipient; thus it is not necessary for the recipient to check that it has the resource before satisfying or deferring the request. However, it is possible for a demand for a missing bottle to be received, so before satisfying or deferring a demand, the recipient must check that it has the resource. For example, suppose D_i sends a request for b to D_j, D_j satisfies the request, but before it arrives at D_i, D_i sends a demand for b to D_j.

Another point concerns when the actions of the dining subroutine should be performed. Some drinkers could be locked out if D_i never relinquishes the dining critical region. The reason is that as long as D_i is in its dining critical region, it has priority for the resources. Thus D_i could cycle through its drinking critical region infinitely often*. To avoid this situation, we keep track of whether the current dining critical region was entered on behalf of the current drinking trying region (i.e., whether the latest C_i occurred after the latest $T_i(B)$). If it was, then D_i may enter its drinking critical region (assuming it has all needed bottles). Otherwise D_i must wait until the current dining critical region has been relinquished before continuing.

We now present the automaton D_i for each i. The state of the automaton consists of the following variables.

- *drink-region*: equals T if the most recent drinking action was $T_i(B)$ (for some B), C if $C_i(B)$, etc. Initially R.

*This problem is also considered in [9], where three other solutions are proposed.

- *dine-region*: equals T if the most recent dining action was T_i, C if C_i, etc. Initially R.
- *need*: equals B, where the most recent drinking action had parameter B. Initially empty.
- *bottles*: set of tokens that have been received and not yet enqueued to be sent. Initially the token for a resource shared between D_i and D_j is in the *bottles* variable of exactly one of D_i and D_j, with the choice being made arbitrarily.
- *deferred*: set of tokens in the set *bottles* for which a request or demand has been received since the token was received. Initially empty.
- *current*: Boolean indicating whether current dining critical region is on behalf of current drinking trying region. Initially false.
- $msgs[j]$ for all $j \neq i$: FIFO queue of messages for D_j enqueued but not yet sent. Initially empty.

The actions of D_i are specified next, in an approximate "chronological" order in which they could occur. Input actions have only effects, while output actions have both preconditions and effects.

- $T_i(B)$ for all $B \subseteq B_i$

  EFFECT:

  $drink\text{-}region \leftarrow T$

  $need \leftarrow B$

  for all $j \neq i$ and $b \in (need \cap B_j) - bottles$: enqueue $request(b)$ in $msgs[j]$

- $Send_i(m, j)$ for all $j \neq i$, $m \in \{request(b), token(b), demand(b) : b \in B_i \cap B_j\}$

  PRECONDITION:

  m is at head of $msgs[j]$

  EFFECT:

  dequeue m from $msgs[j]$

- T_i

  PRECONDITION:

  $dine\text{-}region = R$

  $drink\text{-}region = T$

EFFECT:

dine-region $\leftarrow T$

- $Receive_i(request(b), j)$ for all $j \neq i$, $b \in B_i \cap B_j$

  EFFECT:

  if ($b \in need$) and ($drink\text{-}region \in \{T, C\}$) then

  deferred $\leftarrow$ *deferred* $\cup\{b\}$

  else

  enqueue *token*(b) in $msgs[j]$

  bottles $\leftarrow$ *bottles* $-\{b\}$

- C_i

  EFFECT:

  dine-region $\leftarrow C$

  if $drink\text{-}region = T$ then

  for all $j \neq i$ and $b \in (need \cap B_j) - bottles$: enqueue *demand*($b$)) in $msgs[j]$

  current $\leftarrow$ true

- $Receive_i(demand(b), j)$ for all $j \neq i$, $b \in B_i \cap B_j$

  EFFECT:

  if ($b \in bottles$) and (($b \notin need$) or ($drink\text{-}region \neq C$)) then

  enqueue *token*(b) in $msgs[j]$

  bottles $\leftarrow$ *bottles* $-\{b\}$

  deferred $\leftarrow$ *deferred* $-\{b\}$

- $Receive_i(token(b), j)$ for all $j \neq i$, $b \in B_i \cap B_j$

  EFFECT:

  bottles $\leftarrow$ *bottles* $\cup\{b\}$

- $C_i(B)$ for all $B \subseteq B_i$

  PRECONDITION:

  $drink\text{-}region = T$

  $B = need$

  $need \subseteq bottles$

  if $dine\text{-}region = C$ then $current =$ true

EFFECT:

$drink\text{-}region \leftarrow C$

$current \leftarrow$ false

- E_i

PRECONDITION:

$dine\text{-}region = C$

if $drink\text{-}region = T$ then $current =$ false

EFFECT:

$dine\text{-}region \leftarrow E$

- R_i

EFFECT:

$dine\text{-}region \leftarrow R$

- $E_i(B)$ for all $B \subseteq B_i$

EFFECT:

$drink\text{-}region \leftarrow E$

for all $j \neq i$ and $b \in deferred \cap B_j$: enqueue $token(b)$ in $msgs[j]$

$bottles \leftarrow bottles - deferred$

$deferred \leftarrow \emptyset$

- $R_i(B)$ for all $B \subseteq B_i$

PRECONDITION:

$drink\text{-}region = E$

$B = need$

EFFECT:

$drink\text{-}region \leftarrow R$

4 Proof of Correctness

The proof is divided into three parts, concentrating on the exclusion property, the no-lockout property, and the more-concurrent property.

In Section 4.1, Lemma 1 states that the variables do what one would expect; it is used explicitly and implicitly throughout. Lemma 2 states that the automaton preserves drinking-well-formedness. Lemma 3 states that a group of predicates about the state variables form an invariant. Two of the predicates in this group are the key to showing Lemma 4, the exclusion property.

In Section 4.2, Lemma 8 states that the automaton satisfies the no-lockout property for drinking philosophers. Its proof is based on three preliminary lemmas, Lemma 5, which states that the dining philosophers properties hold, Lemma 6, which states that the variable *current* behaves properly, and Lemma 7, which states that if all bottles are eventually released, then all forks are eventually released.

In Section 4.3, Lemma 9 states that the automaton satisfies the more-concurrent property for drinking philosophers. Theorem 1 in Section 4.4 puts all the pieces together.

Variables of D_i will be denoted by appending the subscript i to the variable name. Note that the concatenation of the variable $msgs_i[j]$ of D_i and the variable $channel_{ij}$ of Net forms a FIFO queue. If a message is in this composite queue, then it is said to be *in transit* from i to j.

4.1 Exclusion

Lemma 1 *The following are true in every state of every execution of Drink($\mathcal{B}$), for all i.*
(a) drink-region$_i$ $= T$ if the most recent drinking action is $T_i(B)$ for some B, drink-region$_i$ $= C$ if the most recent drinking action is $C_i(B)$ for some B, drink-region$_i$ $= E$ if the most recent drinking action is $E_i(B)$ for some B, drink-region$_i$ $= R$ if the most recent drinking action is $R_i(B)$ for some B or if there is no preceding drinking action.
(b) need$_i$ $= B$, for all B, if the most recent drinking action has parameter B (if none, then need$_i$ is empty).
(c) dine-region$_i$ $= T$ if the most recent dining action is T_i, dine-region$_i$ $= C$ if the most recent dining action is C_i, dine-region$_i$ $= E$ if the most recent dining action is E_i, dine-region$_i$ $= R$ if the most recent dining action is $R_i$or if there is no preceding dining action.
(d) Each message in transit from i to j concerns a bottle shared by i and j.

Proof Each of these statements can be shown (independently) by a simple induction on the length of executions. ■

Lemma 2 *Drink($\mathcal{B}$) preserves drinking-well-formedness.*

Proof By induction on the length of executions, using Lemma 1. ■

The next lemma states that a collection of predicates form an invariant. Predicates (a) and (f) are used to show exclusion. The remaining predicates are used in the inductive proof of (a); some are also used later in the paper.

Lemma 3 *The following are true in every state of every drinking-well-formed execution of Drink($\mathcal{B}$), for all i and j, $i \neq j$, and all $b \in B_i \cap B_j$.*

(a) Exactly one of the following is true: b is in bottles$_i$, b is in bottles$_j$, token(b) is in transit from i to j, or token(b) is in transit from j to i.

(b) If b is in deferred$_i$, then

1. *b is in bottles$_i$,*
2. *drink-region$_j = T$,*
3. *b is in need$_j$.*

(c) If request(b) is in transit from i to j, then

1. *exactly one request(b) message is in transit from i to j,*
2. *either token(b) precedes request(b) in transit from i to j or b is in bottles$_j$,*
3. *b is not in deferred$_j$,*
4. *drink-region$_i = T$,*
5. *b is in need$_i$.*

(d) If token(b) is in transit from i to j, then

1. *exactly one token(b) message is in transit from i to j,*
2. *drink-region$_j = T$,*
3. *b is in need$_j$.*

(e) If demand(b) is in transit from i to j and either token(b) precedes it or b is in bottles$_j$, then

1. *drink-region$_i = T$,*
2. *b is in need$_i$,*
3. *no request(b) follows it.*

(f) If b is in need$_i$ and drink-region$_i = C$, then b is in bottles$_i$.

Proof The proof is by induction on the length of execution $e = s_0 a_1 s_1 a_2 \ldots$. It is easy to verify that (a) through (f) are true in s_0. Assume they are true for s_{m-1}, $m > 0$, and show they are true

for s_m. For each of (a) through (f), we consider each possibility for the action a_m. We consider only the non-trivial cases. Present tense refers to s_{m-1} and future tense refers to s_m.

- *Proof of (a).* Action $E_i(B)$: By (b1), $deferred_i$ is a subset of $bottles_i$.

  Action $Receive_i(token(b), j)$: By (d1), there is only one token in transit from j to i.

  Action $Receive_i(request(b), j)$: By (c2), b is in $bottles_i$.

- *Proof of (b1).* Action $Receive_i(request(b), j)$: By (c2), b is in $bottles_i$, and by (c3), it is not in $deferred_i$.

- *Proof of (b2).* Action $C_j(B)$: Suppose there exists $b \in B_j$ that is in $deferred_i$. By (b1), b is in $bottles_i$. By (b3), b is in $need_j$. By (a), b is not in $bottles_j$, contradicting the precondition.

  Action $Receive_i(request(b), j)$: By (c4), $drink\text{-}region_j$ equals T.

- *Proof of (b3).* Action $T_i(B)$: By (b2), $deferred_i$ is empty.

  Action $Receive_i(request(b), j)$: By (c5), b is in $need_j$.

- *Proof of (c1).* Action $T_i(B)$: By (c4), since $drink\text{-}region_i$ is not equal to T, no $request(b)$ is in transit from i to j.

- *Proof of (c2).* Action $T_i(B)$: By (d2), since $drink\text{-}region_i$ is not equal to T, no $token(b)$ is in transit from j to i.

  Action $E_j(B)$: Suppose $token(b)$ will be put in transit from j to i. By the code, b is in $deferred_j$. By (c3), no $request(b)$ is in transit from i to j.

  Action $Receive_i(request(b), j)$: By (c1), only one $request(b)$ is in transit from i to j. Thus the predicate will be vacuously true.

  Action $Receive_j(demand(b), i)$: The only effect of this action that could invalidate (c2) is if b will be removed from $bottles_j$. By the code, then b is in $bottles_j$. But by (e3), no $request(b)$ is in transit from i to j.

- *Proof of (c3).*

  Action $T_i(B)$: By Lemma 1 and (b2), no $b \in B_i$ is in $deferred_j$.

  Action $Receive_j(request(b), i)$: By (c1), only one $request(b)$ is in transit from i to j. Thus the predicate will be vacuously true.

- *Proof of (c4).* Action $C_i(B)$: By the precondition, $need_i$ is a subset of $bottles_i$. By (a) and (c2), no $request(b)$ is in transit from i to j.

- *Proof of (c5).* Action $T_i(B)$: By Lemma 1 and drinking-well-formedness, $drink\text{-}region_i$ equals R. By (c4), no $request(b)$ is in transit from i to j.

- *Proof of (d1).* Action $Receive_i(request(b), j)$: Suppose $token(b)$ will be put in transit from i to j. By (c2), b is in $bottles_i$. By (a), no $token(b)$ is in transit from i to j.

  Action $Receive_i(demand(b), j)$: Suppose $token(b)$ will be put in transit from i to j. By the code, b is in $bottles_i$. By (a), no $token(b)$ is in transit from i to j.

- *Proof of (d2).* Action $C_j(B)$: Suppose $token(b)$ is in transit from i to j. By (d3), b is in $need_j$. By (a), b is not in $bottles_j$, contradicting the precondition.

  Action $Receive_i(request(b), j)$: By (c4), $drink\text{-}region_j$ equals T.

  Action $Receive_i(demand(b), j)$: Suppose $token(b)$ will be put in transit from i to j. By the code, b is in $bottles_i$. By (e1), $drink\text{-}region_j$ equals T.

  Action $E_j(B)$: Consider any $b \in B_i$ such that $token(b)$ will be put in transit from j to i. By the code, b is in $deferred_j$. By (b2), $drink\text{-}region_i$ equals T.

- *Proof of (d3).* Action $T_j(B)$: By (d2), no $token(b)$ is in transit from i to j.

  Action $Receive_i(request(b), j)$: By (c5), b is in $need_j$.

  Action $Receive_i(demand(b), j)$: Suppose $token(b)$ will be put in transit from i to j. By the code, b is in $bottles_i$. By (e2), b is in $need_j$.

  Action $E_i(B)$: Consider any $b \in B_j$ such that $token(b)$ will be put in transit from i to j. By the code, b is in $deferred_i$. By (b3), b is in $need_j$.

- *Proof of (e1).* Action $C_i(B)$: Suppose there exists $demand(b)$ in transit from i to j such that $token(b)$ precedes it or b is in $bottles_j$. By (e2), b is in $need_i$. By the precondition, b is in $bottles_i$, contradicting (a). Thus there is no such $demand(b)$.

- *Proof of (e2).* Action $T_i(B)$: By Lemma 1 and (e1), there is no such $demand(b)$ message.

- *Proof of (e3).* Action $T_i(B)$: By Lemma 1 and (e1), there is no such $demand(b)$ message.

- *Proof of (f).* All by the code.

■

The next lemma states that the automaton satisfies the exclusion property for drinking philosophers. The reason is that, for each bottle, there is exactly one token in the system for that bottle at any time. Whenever a drinker is in its critical region, it holds the token for each bottle it needs. Thus no two drinkers can be in their critical regions at the same time if their bottle requirements overlap.

Lemma 4 *Drink*($\mathcal{B}$) *satisfies the exclusion property for drinking philosophers.*

Proof Consider any drinking-well-formed execution. Suppose in contradiction there exist i, j, B, and B' such that $i \neq j$, $B \cap B' \neq \emptyset$, and some occurrence of $C_i(B)$ is followed by an occurrence of $C_j(B')$ with no intervening occurrence of $E_i(B)$. Let b be an element of $B \cap B'$. In the state s immediately after the $C_j(B')$, *drink-region*$_i = C$, *need*$_i$ contains b, *drink-region*$_j = C$, and *need*$_j$ contains b. But Lemma 3 (f) implies that in state s, b is in both *bottles*$_i$ and *bottles*$_j$, contradicting Lemma 3 (a). ■

4.2 No-Lockout

To show the no-lockout property, we need to rely on the subroutine. First we must check that the environment of *Dine*($\mathcal{B}$)—the composition of the D_i's and Net—preserves dining-well-formedness. From that it follows that every execution is dining-well-formed, and that the properties for dining philosophers hold.

Lemma 5 *(a) The composition of the D_i's and Net preserves dining-well-formedness.*

(b) Every execution of Drink($\mathcal{B}$) *is dining-well-formed.*

(c) Every execution of Drink($\mathcal{B}$) *satisfies the exclusion property for dining philosophers.*

(d) Every fair execution of Drink($\mathcal{B}$) *satisfies the no-lockout property for dining philosophers.*

Proof Part (a) is shown by induction on the length of executions. Part (b) follows from part (a) and the fact that *Dine*($\mathcal{B}$) also preserves dining-well-formedness. Parts (c) and (d) follow from part (b) and the fact that *Dine*($\mathcal{B}$) is a dining philosophers algorithm. ■

We next show that the variable *current* behaves properly. Part (a3) is used in the proofs of Lemmas 7 and 9, and part (b) is used in the proof of Lemma 7. Parts (a1) and (a2) are needed to make the inductive proof go through.

The predicate stated in Lemma 6 (b) uses the notion of a "live" demand message. A demand message for b in transit from i to j is *live* if either the token for b precedes the demand in transit from i to j, or D_j has the token for b, or the token for b is in transit from j to i. (A non-live demand message can arise if i sends a request to j, j receives the request and sends the token, i sends a demand to j, and i receives the token from j. The demand still in transit from i to j is now non-live.)

Lemma 6 *In every every drinking-well-formed execution e of Drink($\mathcal{B}$), for all i and all states s of e, the following are true.*
(a) If current$_i$ is true in s, then

1. *dine-region$_i = C$ in s,*
2. *drink-region$_i = T$ in s,*
3. *the most recent occurrence in e (up to s) of $T_i(B)$, for any B, precedes the most recent occurrence of C_i.*

(b) If current$_i$ is false in s, then no live demand(b) message is in transit from i to j in s.

Proof (a) By a simple induction on the length of e, using Lemma 1 (a) and (c) and Lemma 5 (b).

(b) By induction on the length of e. The non-trivial cases are the following.

Action C_i: By the code, whenever any demand message is added, it will be live and *current$_i$* will be true.

Action $C_i(B)$: We show that setting *current$_i$* to false is allowable. Suppose there is a live *demand*(b) message in transit from i to j. By definition of "live" and Lemma 3 (d3) and (e2), b is in *need$_i$*. By definition of "live" and Lemma 3 (a), b is not in *bottles$_i$*, contradicting the precondition. ■

The next lemma states that if no drinker is ever stuck in its critical region, then no diner is ever stuck in its critical region. The heart of the argument is that once C_i occurs during D_i's drinking trying region, D_i sends demands for missing needed bottles. Consider a recipient D_j. If D_j has the bottle and is not using it, then D_j satisfies the request, since by mutual exclusion of the dining subroutine, D_j cannot also be in its dining critical region. If D_j is using the resource, then by the

assumption that no drinker is stuck in its critical region, D_j eventually finishes and satisfies the request. Thus eventually D_i gets all needed bottles and enters its drinking critical region, after which E_i must occur.

Lemma 7 *In any fair drinking-well-formed execution of Drink($\mathcal{B}$), if, for all i and B, every occurrence of $C_i(B)$ is followed by an occurrence of $E_i(B)$, then, for all i, every occurrence of C_i is followed by an occurrence of E_i.*

Proof Consider any fair drinking-well-formed execution in which, for all i and B, every occurrence of $C_i(B)$ is followed by an occurrence of $E_i(B)$. Suppose in contradiction there is an occurrence of C_i, for some i, that is not followed by an occurrence of E_i. Thus *dine-region*$_i$ equals C throughout the rest of the execution and there is no later occurrence of T_i, C_i, E_i, or R_i.

Case 1: drink-region$_i$ equals C, E, or R when the final C_i occurs. If no $T_i(B)$ occurs (for any B) after the final C_i, then *drink-region*$_i$ is never again equal to T. If some $T_i(B)$ does occur after the final C_i, then by Lemma 6 (c), *current*$_i$ is never again true. In either case, E_i is continuously enabled after some point but never occurs, contradicting the fairness of the execution.

Case 2: drink-region$_i$ equals T when the final C_i occurs. Thus *current*$_i$ is set to true at that time. Let B be the set of needed bottles.

Suppose in contradiction that no subsequent $C_i(B)$ occurs. Then *current*$_i$ is true forever after the final C_i. We first argue that once any $b \in B$ is put into *bottles*$_i$ after the final C_i, it stays there forever. If a request for b arrives at D_i, then since b is in *need*$_i$, the request is deferred. Suppose a demand for b arrives at D_i. Then this is a live demand, since b is in *bottles*$_i$. By Lemma 6 (b), *current*$_i$ is true, so by Lemma 6 (a1), *dine-region*$_i$ equals C. But this contradicts Lemma 5 (c).

We now show that eventually every $b \in B$ ends up in *bottles*$_i$ after the final C_i. When the final C_i occurs, D_i sends demands for all needed and missing bottles. When some D_j receives *demand*(b) from D_i, by Lemma 3 (a) either b is in *bottles*$_j$, b is in transit from j to i, or b is already in *bottles*$_i$. (Bottle b is not in transit from i to j because D_i never sends off any needed bottles after the final C_i occurs and message delivery is FIFO.) If b is in *bottles*$_j$ when the demand arrives, then D_j immediately puts b in transit to i, since by Lemma 5 (c) D_j cannot also be in its dining critical region. Thus eventually all $b \in B$ will end up in *bottles*$_i$.

Thus $C_i(B)$ is continuously enabled, once it has acquired all needed bottles. Therefore, it must eventually occur.

Once $C_i(B)$ occurs after the final C_i, $current_i$ is set to false. Since $current_i$ is only set to true when C_i occurs and there is no later occurrence of C_i, it stays false forever. Thus E_i is continuously enabled, but never occurs, contradicting the fairness of the execution. ■

The next lemma shows the no-lockout property for drinking philosophers. The argument is as follows. Once $T_i(B)$ occurs, then T_i occurs subsequently. Lemma 7 and the dining no-lockout property imply that C_i, E_i and R_i occur subsequently. But the E_i only occurs once $C_i(B)$ occurs. By the hypothesis of the drinking no-lockout property, $E_i(B)$ occurs after the $C_i(B)$. Finally, the enabling conditions for $R_i(B)$ ensure that $R_i(B)$ occurs subsequently.

Lemma 8 *In any fair drinking-well-formed execution of Drink($\mathcal{B}$), if, for all i and B, every occurrence of $C_i(B)$ is followed by an occurrence of $E_i(B)$, then, for all i and B, every occurrence of $T_i(B)$ is followed by an occurrence of $R_i(B)$.*

Proof Consider any fair drinking-well-formed execution in which, for all i and B, every occurrence of $C_i(B)$ is followed by an occurrence of $E_i(B)$. Suppose in contradiction there is an occurrence of $T_i(B)$ that is not followed by an occurrence of $R_i(B)$.

Case 1: The final $T_i(B)$ is followed by an occurrence of $C_i(B)$. Then by hypothesis it is followed by an occurrence of $E_i(B)$. Thus $drink\text{-}region_i = E$ for the rest of the execution. But then $R_i(B)$ is continuously enabled without ever occurring, contradicting the fairness of the execution.

Case 2: The final $T_i(B)$ is not followed by an occurrence of $C_i(B)$.

If there is an occurrence of C_i following the $T_i(B)$, then $current_i$ is set to true. By Lemma 7, the C_i is followed by an occurrence of E_i. Thus eventually $current_i$ is set to false, in order for E_i to be enabled. But $current_i$ is only set to false when $C_i(B)$ occurs.

If there is no occurrence of C_i following the $T_i(B)$, then eventually after $T_i(B)$, $dine\text{-}region_i$ is always equal to R, by Lemma 7 and Lemma 5 (d) (the dining no-lockout property). But then T_i is enabled forever without occurring, contradicting the fairness of the execution. ■

4.3 More-Concurrent

The next lemma shows the more-concurrent property for drinking philosophers. Unlike in the proof of Lemma 8 (no-lockout), here we cannot use Lemma 7. Instead, we use Lemmas 3 and 6 together with the non-overlapping property to deduce that eventually the drinker will have all its needed

bottles. The interaction between the enabling conditions for the dining and drinking output actions of D_i ensures that $C_i(B)$ occurs subsequently.

Lemma 9 *Consider any fair drinking-well-formed execution of Drink($\mathcal{B}$) and any occurrence of $T_i(B)$, for any i and B. If the occurrence of $T_i(B)$ is non-overlapping, then the $T_i(B)$ is followed by an occurrence of $C_i(B)$.*

Proof Suppose for contradiction that there is an execution satisfying the hypothesis of the lemma, but the $T_i(B)$ is not followed by an occurrence of $C_i(B)$. Thus *drink-region*$_i = T$ for the rest of the execution.

When the final $T_i(B)$ occurs, D_i sends requests for the needed bottles that it is missing. By the non-overlapping property, no other drinker either wants or uses any bottle in B for the rest of the execution. Thus each recipient of a request from D_i satisfies the request immediately. Furthermore, the non-overlapping property and Lemma 3, parts (c4), (c5), (e1) and (e2), imply that D_i never again receives a request for a needed bottle or a demand for a needed bottle that it holds. Therefore, once D_i obtains a needed bottle, that bottle remains at D_i. Eventually $B = \mathit{need}_i \subseteq \mathit{bottles}_i$, and this remains true for the rest of the execution.

If an occurrence of C_i follows the final $T_i(B)$, then after this C_i, *current*$_i$ is always true, since only the occurrence of $C_i(B)$ can make it false. Suppose no occurrence of C_i follows the final $T_i(B)$. If *dine-region*$_i \neq C$ when the $T_i(B)$ occurs, then *dine-region*$_i$ never equals C after the final $T_i(B)$. If *dine-region*$_i = C$ when the $T_i(B)$ occurs, then by Lemma 6 (c), *current*$_i$ = false, by fairness E_i occurs subsequently, and *dine-region*$_i$ never equals C after the E_i. Thus whether or not an occurrence of C_i follows the final $T_i(B)$, $C_i(B)$ is continuously enabled after some point in the execution. Since $C_i(B)$ never occurs, this contradicts the fairness of the execution. ■

4.4 Main Theorem

Theorem 1 *Drink($\mathcal{B}$) is a drinking philosophers algorithm.*

Proof It is easy to see that *Drink*($\mathcal{B}$) has the correct input and output actions. By Lemma 2 it preserves drinking-well-formedness. The exclusion condition follows from Lemma 4. The no-lockout condition follows from Lemma 8. The more-concurrency condition follows from Lemma 9. ■

5 Complexity Analysis

In this section, we analyze the worst-case waiting time of our algorithm, assuming no drinker is ever stuck in its critical region, as well as evaluating it using the criteria listed by [2]. The analysis of the worst-case waiting time shows that the limiting factor is the dining philosophers subroutine. By replacing the $O(n)$ time subroutine of [2] with an $O(1)$ time subroutine (for instance, that of [7]), we obtain an $O(1)$ time drinking philosophers algorithm.

We would like to bound how long a user must wait after requesting to enter its critical region until it does so. Our measure of time complexity is analogous to that of Peterson and Fischer ([10]) for asynchronous shared memory systems, in which an upper bound on process step time, but no lower bound, is assumed. (Thus, all interleavings of system events are still possible.) Our time measure provides distinct upper bounds on process step time and on message delivery time.

Given an execution e of $Drink(\mathcal{B})$, a *timing function* for e is a nondecreasing function t_e mapping positive integers to nonnegative real numbers such that for each real number t, only a finite number of integers i satisfy $t_e(i) < t$. Intuitively, $t_e(i)$ is the real time at which the i-th action occurs; we rule out an infinite number of actions occurring before a finite real time. We require that t_e satisfy the following conditions whenever e is fair, for some constants s and d. (The constant s is the maximum process step time and d is the maximum message delivery time.)

- For all i, once an output action of D_i becomes enabled (namely $C_i(B)$, $R_i(B)$, T_i, E_i, or $Send_i$), that action occurs within s time.
- Once a message is sent, it is received within d time.

For the rest of this section, we only consider fair drinking-well-formed executions in which for all i and B, every occurrence of $C_i(B)$ is followed by an occurrence of $E_i(B)$, and with timing functions satisfying the conditions given above.

Let try_{Drink} be the maximum time, over all i and all B, between any $T_i(B)$ action and the subsequent $C_i(B)$ action, in any execution. (It is the longest time that a drinker is in its trying region.) Let $crit_{Drink}$ be the maximum time, over all i and all B, between any $C_i(B)$ action and the subsequent $E_i(B)$ action, in any execution. (It is the longest time that a drinker is in its critical region.) Let try_{Dine} be the maximum time over all i between any T_i action and the subsequent C_i action, in any execution. (It is the longest time a diner is in its trying region.) Let $crit_{Dine}$ be the

maximum time over all i between any C_i action and the subsequent E_i action, in any execution. (It is the longest time a diner is in its critical region.) Let $exit_{Dine}$ be the maximum time over all i between any E_i action and the subsequent R_i action, in any execution. (It is the longest time a diner is in its exit region.)

We assume that $crit_{Drink}$ and $exit_{Dine}$ are constants.

Theorem 2 gives an upper bound on try_{Drink}, the maximum time a drinker must wait after requesting to enter its critical region until it is allowed to do so. Its proof uses the fact, stated in Lemma 11, that there is an upper bound on the number of messages in transit between any i and j at any time. Lemma 11 in turn is proved using Lemma 10, which bounds the number of demand messages, and Lemma 3 (c1) and (d1), which bound the number of request and token messages.

Lemma 10 *The following are true in every state of every drinking-well-formed execution, for all i and j, $i \neq j$, and all $b \in B_i \cap B_j$.*

(a) There is at most one live demand(b) message in transit from i to j.

(b) There is at most one non-live demand(b) message in transit from i to j.

Proof The proof is by induction on the length of execution $e = s_0 a_1 s_1 a_2 \ldots$. It is easy to verify that (a) and (b) are true in s_0. Assume they are true for s_{m-1}, $m > 0$, and show they are true for s_m. For each of (a) and (b), we consider each possibility for the action a_m. We present only the non-trivial cases. Present tense refers to s_{m-1} and future tense refers to s_m.

Note that none of the Receive actions causes a demand message to be added or a non-live demand message to become live.

- *Proof of (a).* Action C_i: By Lemmas 1 and 5 (b), $dine\text{-}region_i$ is not equal to C. By Lemma 6 (a1), $current_i$ equals false. By Lemma 6 (b), there is no live demand message in transit from i to j.

- *Proof of (b).* Action $Receive_i(token(b), j)$: Suppose the receipt of this token will cause a $demand(b)$ message that is in transit from i to j to become non-live (in s_m). We must show that there is not already a non-live $demand(b)$ message in transit from i to j (in s_{m-1}). Suppose there is such a non-live demand message. By definition of "live", either there is a $token(b)$ message in transit from i to j after it, or b is in $bottles_i$. Then Lemma 3 (a) implies that no $token(b)$ message is in transit from j to i. But this contradicts the precondition.

■

Let r be the maximum number of bottles shared by any two drinkers.

Lemma 11 *In any state of any execution, for all i and j, $i \neq j$, the maximum number of messages in transit from i to j is $4r$.*

Proof By Lemma 1 (d) the only messages in transit from i to j are those involving bottles in $B_i \cap B_j$. Choose any $b \in B_i \cap B_j$. By Lemma 3 (c1), there is at most one *request*(b) message in transit from i to j. By Lemma 3 (d1), there is at most one *token*(b) message in transit from i to j. By Lemma 10 (a), there is at most one live *demand*(b) message in transit from i to j. By Lemma 10 (b), there is at most non-live *demand*(b) message in transit from i to j. Since there are at most r choices for b, the result follows. ■

Theorem 2 $try_{Drink} \leq (8r+3)s + 2d + exit_{Dine} + try_{Dine} + crit_{Drink}$.

Proof Choose execution e and fix i. Suppose $T_i(B)$ occurs at time t, for some B. In the worst case, *dine-region*$_i = C$ at time t. By time s later, E_i occurs; by time $exit_{Dine}$ later, R_i occurs; by time s later, T_i occurs, and by time try_{Dine} later, C_i occurs.

When this C_i occurs, D_i enqueues a demand message for all required and missing bottles. By Lemma 11 there are fewer than $4r$ messages ahead of each demand in the corresponding $msgs_i$ queue at D_i. By the assumption about the timing function for e, each demand is received by time $4rs + d$ later. As in the proof of Lemma 7 (Case 2), either the recipient immediately enqueues the token to D_i or else the recipient is in its drinking critical region and sends the token by time $crit_{Drink}$ later. As with the demand message, the token is received by time $4rs + d$ later. By time s later, $C_i(B)$ occurs. ■

Since we assume that $crit_{Drink}$, $exit_{Dine}$, r, d and s are constants, the worst-case waiting time of this solution depends on try_{Dine}, the worst-case waiting time of the dining philosophers subroutine. For any dining philosophers algorithm, try_{Dine} will depend on $crit_{Dine}$, because users will have to wait for resources currently in use to become available. We now give an informal argument for an upper bound on $crit_{Dine}$. Once C_i occurs, E_i will occur after D_i has sent demands for needed bottles ($4rs + d$), these demands have been satisfied ($crit_{Drink} + 4rs + d$), and D_i has entered its

drinking critical region (s). The upper bound then is $(8r+1)s + 2d + crit_{Drink}$. Thus $crit_{Dine}$ is also a constant, under our assumptions.

The dining philosophers subroutine used by [2] has try_{Dine} of $O(n)$. By replacing it with, for instance, the dining philosophers algorithm of [7], which has worst-case waiting time of $O(1)$, we obtain a more time-efficient drinking philosophers algorithm. The algorithm of [7] has time $O(1)$ in the sense that the worst-case waiting time is a function of local information, including the maximum number of users for each resource, and the maximum number of resources for each user, and is not necessarily a function of the total number of users.

Our drinking philosophers algorithm could be modified to replace r with a small constant, if the request, demand, and token messages contained a set of bottles instead of a single bottle.

Five criteria for evaluating resource allocation algorithms are given by [2]: fairness, symmetry, economy, concurrency and boundedness. We discuss each in turn.

Fairness corresponds to our definition of no-lockout. Our drinking philosophers solution has the no-lockout property.

Symmetry as defined in [2] means that "there is no priority or any other form of externally specified static partial ordering among philosophers or bottles". This property is true of our solution, as long as it is true of the subroutine. However, the state of the entire system cannot be symmetric, or else no deterministic solution would be possible, as shown by Lehmann and Rabin [6]. Chandy and Misra's dining philosophers algorithm satisfies this definition of symmetry but breaks system symmetry by the locations of the individual resources. Lynch's resource allocation algorithm [7] breaks system symmetry by ordering the resources (and thus does not satisfy this definition of symmetry).

Economy means that processes send and receive a finite number of messages between subsequent entries to their critical regions, and a process that enters its critical region a finite number of times does not send or receive an infinite number of messages. Our solution has this property: Recall that when $T_i(B)$ occurs, D_i sends requests for all missing resource. It defers any request that it receives for a needed bottle while $drink\text{-}region_i = T$, but it yields to demands. When $dine\text{-}region_i$ becomes C, it sends demands for any missing resources. Thus at most four messages (request, the token, demand, the token) are sent on behalf of any one bottle for any one trying attempt. Furthermore, once a drinker stops wanting to enter its critical region, it may receive a request for each of its bottles, but after satisfying the requests, it never sends or receives any more messages.

Concurrency means that "the solution does not deny the possibility of simultaneous drinking from different bottles by different philosophers." This is true of our algorithm, since it satisfies the more-concurrent condition.

Boundedness means that the number of messages in transit between any two drinkers is bounded, and the size of each message is bounded. This is true of our solution (the bound on the number of messages is $4r$, by Lemma 11).

6 Conclusions

We have given careful definitions of what it means to be dining philosophers and drinking philosophers algorithms. We described a modular drinking philosophers algorithm that uses as a subroutine any dining philosophers algorithm. We proved the correctness of our algorithm, and analyzed its time complexity. One advantage of our modular approach is that an algorithm with improved worst-case time performance can be obtained by using a time-efficient dining philosophers subroutine. We close by mentioning some open problems.

The proof we have presented here could be changed technically in a couple of ways. The style of the liveness arguments is operational and informal, yet they are somewhat close to temporal logic statements. It probably would not be difficult to complete a formal proof in temporal logic, although what we have should suffice for understanding. Another change would be to have an explicit model for each user program, as an I/O automaton that just interacts with D_i and only keeps track of its region. Then the mutual exclusion invariant could be stated in terms of no two users being in the same region and the well-formedness hypotheses could be dropped. The drawback would be having to define these extra system components.

Extensions to the results presented in this paper include designing a way to handle the sharing of a resource among more than two processes in a manner that is more efficient than the one suggested in Section 2; comparing the inherent complexity of the drinking philosophers problem using a dining subroutine as opposed to solving the problem directly; and considering other versions of the "more concurrent" condition for drinking philosophers. In the rest of this section, we describe some specific versions of that condition.

The strongest possible condition would require that if a drinker requests a set B of bottles, it should eventually enter its critical region, as long as no other drinker uses any of the bottles in B forever. (Some bottles in B could be kept forever after this request is satisfied.) Neither the

algorithm in this paper nor those of [2, 9, 5] satisfy this condition[†]. An interesting open problem is to devise one that does or prove that none exists.

The following situation shows that our algorithm does not satisfy this condition. (Essentially the same scenario shows that the algorithms of [2, 9] also do not; a simple chain scenario suffices for [5].) Suppose there are three drinkers, 1, 2 and 3; 1 and 2 share bottle a, 2 and 3 share bottle b. First, 1 gets bottle a, enters its drinking critical region, and stays there forever. Then 2 requests a and b, obtains b, and enters its dining critical region. Since 2 can never obtain a, it stays in its dining critical region forever. Finally, 3 requests b. Drinker 2 does not relinquish b upon a mere request, and 3 can never demand b, because it can never enter its dining critical region. Thus, even though 3's bottle request includes no bottle that is ever in use, it can never enter its drinking critical region.

There is a condition intermediate between this strongest condition and the more-concurrent condition of Section 2.3 that the algorithms of [2, 9, 5] solve and ours does not. This condition states that if a drinker requests a set B of bottles, it should eventually enter its critical region, as long as no other drinker uses *or wants* any of the bottles in B forever.

The following scenario shows that our algorithm does not solve this problem. Suppose there are five drinkers, 1 through 5. Drinkers 1 and 2 share bottle a, 2 and 3 share b, 3 and 4 share c, and 3 and 5 share d. First, 1 gets a, enters its drinking critical region and stays there forever. Then 2 requests a and b, obtains b and enters its dining critical region. Since 2 can never obtain a from 1, 2 remains in its dining critical region forever. Next, 3 requests c and d. It obtains c from 4. Then 4 requests c from 3, the request is deferred, 4 demands c from 3, and the demand is satisfied. Now 3 obtains d from 5. Finally 4 finishes using c. But 3 will never get c from 4, because c is not in 4's deferred set and 3 can never demand it. Thus, although none of the bottles required by 3 are ever wanted forever by another drinker, 3 cannot enter its drinking critical region.

In contrast, the algorithm of [2] (and [9]) will allow 3 to enter its drinking critical region. The forks in the dining philosophers algorithm provide a priority for the use of the corresponding bottles by the drinkers. The priority alternates between the two processes sharing the resource. Thus, once 3 obtains c it will not relinquish it until it has gotten to use it. In general, priority is broken down on a link-by-link basis, whereas in our (more modular) algorithm, the priority comes only with entering the dining critical region. This is an example of optimizing to gain extra concurrency at the expense of violating modularity. The algorithm of [5] has priority information explicitly

[†]The concurrency properties of [1] are not discussed.

available in the counters.

7 Acknowledgments

We thank Alan Fekete and Prasad Sistla for helpful discussions, Subodh Kumar for careful readings, and the anonymous referees for useful comments.

References

[1] B. Awerbuch and M. Saks, "A Dining Philosophers Algorithm with Polynomial Response Time," *Proc. 31st IEEE Symposium on Foundations of Computer Science,* October 1990, pp. 65–74.

[2] K. M. Chandy and J. Misra, "The Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems,* vol. 6, no. 4, October 1984, pp. 632–646.

[3] M. Choy and A. K. Singh, "Efficient Fault-Tolerant Algorithms for Resource Allocation in Distributed Systems," *Proc. 24th ACM Symposium on Theory of Computing,* May 1992, pp. 593–602.

[4] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica,* vol. 1, fasc. 2, 1971, pp. 115–138.

[5] D. Ginat, A.U. Shankar, and A.K. Agrawala, "An Efficient Solution to the Drinking Philosophers Problem and its Extensions," *Proc. 3rd International Workshop on Distributed Algorithms,* LNCS 392, Springer-Verlag, September 1989, pp. 83–93.

[6] D. Lehmann and M. Rabin, "On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem," *Proc. 8th ACM Symposium on Principles of Programming Languages,* January 1981, pp. 133–138.

[7] N. A. Lynch, "Upper Bounds for Static Resource Allocation in a Distributed System," *Journal of Computer and System Sciences,* vol. 23, no. 2, October 1981, pp. 254–278.

[8] N. A. Lynch and M. R. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proc. 6th ACM Symposium on Principles of Distributed Computing,* August 1987, pp. 137–151. (Also available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, April 1987.)

[9] S.L. Murphy and A.U. Shankar, "A Note on the Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems,* vol. 10, no. 1, Jaunary 1988, pp. 178–188.

[10] G. L. Peterson and M. J. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System," *Proc. 9th ACM Symposium on Theory of Computing,* May 1977, pp. 91–97.

[11] A.K. Singh, *Ranking in Distributed Systems,* Ph.D. dissertation, Department of Computer Sciences, University of Texas at Austin, December 1989.

[12] A.K. Singh and M.G. Gouda, "Rankers: A Classification of Synchronization Problems," manuscript.

[13] J. L. Welch, *Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms,* Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1988.

[14] J. L. Welch and N. A. Lynch, "Synthesis of Efficient Drinking Philosophers Algorithms," Technical Memorandum MIT/LCS/TM-417, Laboratory for Computer Science, Massachusetts Institute of Technology, November 1989.