LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-417

SYNTHESIS OF EFFICIENT DRINKING PHILOSOPHERS ALGORITHMS

Jennifer L. Welch
Nancy A. Lynch

November 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Synthesis of Efficient Drinking Philosophers Algorithms

Jennifer L. Welch
University of North Carolina
Chapel Hill, NC 27599

Nancy A. Lynch
MIT Laboratory for Computer Science
Cambridge, MA 02139

This work was supported in part 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.

Abstract: A variant of the drinking philosophers algorithm of Chandy and Misra is described and proved correct in a modular way, using the I/O automaton model of Lynch and Tuttle. 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 true 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). Formal definitions are given to distinguish the drinking and dining philosophers problems and to specify precisely varying degrees of concurrency.

Key words: dining philosophers, distributed algorithms, drinking philosophers, modularity, resource allocation, time complexity.

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. Formal definitions are given to distinguish the drinking and dining philosophers problems and to specify precisely varying degrees of concurrency.

The drinking philosophers problem is a dynamic variant due to Chandy and Misra (1984) of the dining philosophers problem, a much-studied resource allocation problem. In the original dining philosophers problem of Dijkstra (1971), 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 (1984). 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 true 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 roughly equal to the maximum waiting time for a dining philosopher to enter its critical region in the subroutine. Thus, by replacing the dining philosophers algorithm of Chandy and Misra (1984), which has waiting time $O(n)$, with a dining philosophers algorithm such as the one of Lynch (1981), which has waiting time $O(1)$, we obtain a more efficient drinking philosophers algorithm.

We provide definitions that distinguish the drinking and dining philosophers problem, and that specify precisely varying degrees of concurrency. We use the model of Lynch and Tuttle (1987), which is useful for stating properties that concern the infinite behavior of a system, such as no-deadlock and no-lockout, and which supports modular algorithm design and verification. This model, together with the particular definitions developed in this paper for expressing the safety and liveness properties for resource allocation problems, make possible a clear and precise proof of correctness for our construction.

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 Section 5 analyzes the performance of our algorithm with respect to various complexity measures. Section 6 contains our conclusions.

2. Problem Statement

There are n user processes in the system being modeled, and at various times, each one needs 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.

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 define two resource allocation problems, dining philosophers and drinking philosophers, as external schedule modules, that is, as sets of allowable interactions between inputs and outputs. (See Appendix A for a summary of the I/O automaton model of Lynch and Tuttle (1987).) 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 U of resources (with input actions $T_i(U)$ and $E_i(U)$ for each user i), decides which users are allowed to enter their critical and remainder regions at which times (with output actions $C_i(U)$ and $R_i(U)$). The automaton, then, represents the algorithm used to allocate the resources.

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.

We consider several versions of the dining and drinking philosophers problems, each satisfying successively stronger liveness properties. First we define the basic dining and drinking philosophers problems, which only satisfy safety conditions. Then the no-deadlock versions are defined, in which as long as some user is in its trying region, eventually some user enters its critical region. In the no-lockout versions, any user that enters its trying region eventually enters its critical region. The no-deadlock and no-lockout conditions assume that no user keeps resources forever.

A dining philosophers algorithm can be used to solve the drinking philosophers problem by treating each resource request as a request for the entire set of resources which that user will ever need. However, users may be blocked unnecessarily in such a scheme. A preferable solution would not rule out two users that share a resource from entering their critical regions simultaneously, if their current resource requirements are disjoint. We capture part of this intuition by defining the "more-concurrent" condition for the drinking philosophers problem — if a user requests a set of resources, none of which is currently being sought or used by another user, then the first user eventually enters its critical region, even if some other resources are never relinquished. (In our conclusions we discuss even stronger forms of the drinking philosophers problem.)

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 substantial. If a resource is shared by k users, then it can be represented by k choose 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*.

2.1 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.

For each i, let the set $\{T_i, C_i, E_i, R_i\}$ be denoted $F\text{-}TCER_i$. (The letter F stands for "fork.") T_i is the action by which user i enters its trying region, desiring the resources F_i, and analogously for the other actions and regions. Since each user i must request the same set F_i of forks each time, we do not explicitly include the set of forks in the action names. Let $F\text{-}TCER = \bigcup_{i=1}^{n} F\text{-}TCER_i$.

In order to specify the external schedule module for the dining philosophers problem, we define the following predicates on any sequence α. (Throughout this paper, Greek letters stand for sequences from a set, and Roman letters for single elements.) If α is a sequence from a set S and T is a subset of S, then $\alpha|T$ is defined to be the subsequence of α consisting of elements in T.

- α is *dining-well-formed* if for all i, the subsequence of α restricted to $F\text{-}TCER_i$ conforms to the pattern $T_iC_iE_iR_i\ldots$.

- α satisfies *(REL-F)* if for all i, if $\alpha|F\text{-}TCER_i$ is finite, then $\alpha|F\text{-}TCER_i$ does not end in C_i. (REL-F) states that every user eventually releases the resources it is granted, by leaving its critical region.

- α satisfies *(EX-F)* if for all i and j, $i \neq j$, if $\alpha = \beta_1 C_i \beta_2 C_j \beta_3$ and if $F_i \cap F_j \neq \emptyset$, then β_2 contains E_i. (EX-F) states that each user has exclusive access to a needed resource when it is in its critical region.

- α satisfies *(ND-F)* if for all i, if $\alpha|F\text{-}TCER$ is finite, then $\alpha|F\text{-}TCER_i$ ends in R_i or is empty. (ND-F) states that the system is not deadlocked, i.e., that users stop taking steps only if they are all in their remainder regions.

- α satisfies *(NL-F)* if for all i, if $\alpha|F\text{-}TCER_i$ is finite, then $\alpha|F\text{-}TCER_i$ ends in R_i or is empty. (NL-F) states that the system has no-lockout, i.e., that any individual user stops taking steps only if it is in its remainder region.

The *dining philosophers* problem for $\mathcal{F}$ is the external schedule module *DiPh* such that:

- $in(DiPh) = \{T_i, E_i : 1 \leq i \leq n\}$,
- $out(DiPh) = \{C_i, R_i : 1 \leq i \leq n\}$,
- *DiPh* preserves dining-well-formedness (see Appendix A for the definition of "preserves"), and
- *scheds(DiPh)* is the set of all sequences α of actions satisfying the following implication:
  Exclusion: If α is dining-well-formed, then α satisfies (EX-F).

The exclusion implication states that if the schedule is dining-well-formed, then no two users are in their critical regions at the same time with the same resource.

The *no-deadlock dining philosophers* problem for $\mathcal{F}$ is the external schedule module that is the same as the dining philosophers problem except that in addition to the exclusion implication, schedules must satisfy the following implication:

No-deadlock: If α is dining-well-formed and α satisfies (REL-F), then α satisfies (ND-F).

The no-deadlock implication states that if the schedule is dining-well-formed and no user keeps resources forever (by staying in its critical region forever), then eventually some user will enter its critical region.

The *no-lockout dining philosophers* problem for $\mathcal{F}$ is the external schedule module that is the same as the dining philosophers problem except that in addition to the exclusion implication, schedules must satisfy the following implication:

No-lockout: If α is dining-well-formed and α satisfies (REL-F), then α satisfies (NL-F).

The no-lockout implication states that if the schedule is dining-well-formed and no user keeps resources forever (by staying in its critical region forever), then eventually every user that wishes will enter its critical region.

2.2 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 3.1. Two new conditions, $(\text{NS-B})_i$ and $(\text{NOV-B})_i$, both indexed by user id i, are used to create implications to distinguish the drinking philosophers problem from the dining philosophers problem, as will be discussed.

For each i, let the set $\{T_i(B), C_i(B), E_i(B), R_i(B) : B \subseteq B_i \text{ and } B \neq \emptyset\}$ be denoted $B\text{-}TCER_i$. (The letter B stands for "bottles.") Let $B\text{-}TCER = \bigcup_{i=1}^{n} B\text{-}TCER_i$. The following predicates are defined for any sequence α.

- α is *drinking-well-formed* if for all i, the subsequence of α restricted to $B\text{-}TCER_i$ conforms to the pattern $T_i(B)C_i(B)E_i(B)R_i(B)T_i(B')C_i(B')E_i(B')R_i(B')\ldots$.

- α satisfies *(REL-B)* if for all i, if $\alpha|B\text{-}TCER_i$ is finite, then $\alpha|B\text{-}TCER_i$ does not end in $C_i(B)$ for any B. (REL-B) states that every user eventually releases all resources that it is granted, by leaving its critical region.

- α satisfies *(EX-B)* if for all i and j, $i \neq j$, if $\alpha = \beta_1 C_i(B)\beta_2 C_j(B')\beta_3$ and if $B \cap B' \neq \emptyset$, then β_2 contains $E_i(B)$. (EX-B) states that every user has exclusive access to needed resources when it is in its critical region.

- α satisfies *(ND-B)* if for all i, if $\alpha|B\text{-}TCER$ is finite, then $\alpha|B\text{-}TCER_i$ ends in $R_i(B)$ for some B or is empty. (ND-B) states that the system is not deadlocked, i.e., that users stop taking steps only if all users are in their remainder regions.

- α satisfies *(NL-B)* if for all i, if $\alpha|B\text{-}TCER_i$ is finite, then $\alpha|F\text{-}TCER_i$ ends in $R_i(B)$ for some B or is empty. (NL-B) states that the system has no-lockout, i.e., that any particular user stops taking steps only if it is in its remainder region.

The *drinking philosophers* problem for $\mathcal{B}$ is the external schedule module $DrPh$ such that:

- $in(DrPh) = \{T_i(B), E_i(B) : 1 \leq i \leq n, B \subseteq B_i \text{ and } B \neq \emptyset\}$,
- $out(DrPh) = \{C_i(B), R_i(B) : 1 \leq i \leq n, B \subseteq B_i \text{ and } B \neq \emptyset\}$,
- $DrPh$ preserves drinking-well-formedness, and
- $scheds(DrPh)$ is the set of all sequences α of actions satisfying the following implication:
  Exclusion: If α is drinking-well-formed, then α satisfies (EX-B).

The *no-deadlock drinking philosophers* problem for $\mathcal{B}$ is the external schedule module that is the same as the drinking philosophers problem except that in addition to the exclusion implication, schedules must satisfy the following implication:

No-deadlock: If α is drinking-well-formed and α satisfies (REL-B), then α satisfies (ND-B).

The *no-lockout drinking philosophers* problem for $\mathcal{B}$ is the external schedule module that is the same as the drinking philosophers problem except that in addition to the exclusion implication, schedules must satisfy the following implication:

No-lockout: If α is drinking-well-formed and α satisfies (REL-B), then α satisfies (NL-B).

The next two predicates are introduced to create an implication that will distinguish between the drinking and dining philosophers problems.

- (NOV-B)$_i$ $(1 \leq i \leq n)$ For all j and any B and B' with $B \cap B' \neq \emptyset$: (1) if $\alpha = \beta_1 T_i(B)\beta_2 T_j(B')\beta_3$, then β_2 contains $C_i(B)$; and (2) if $\alpha = \beta_1 T_j(B')\beta_2 T_i(B)\beta_3$, then β_2 contains $E_j(B')$. (NOV-B)$_i$ (NOV for "no overlap") states that whenever user i requests a resource, (1) no other user requests that resource until after user i enters its critical region, and (2) any other user that has previously requested that resource has already released it.

- (NS-B)$_i$ $(1 \leq i \leq n)$ If $\alpha|B\text{-}TCER_i$ is finite, then $\alpha|B\text{-}TCER_i$ does not end in $T_i(B)$ or $E_i(B)$ for any B. (NS-B)$_i$ (NS for "not stuck") states that user i is never stuck in its trying or exit regions.

The next problem statement requires a degree of concurrency in the drinking philosophers problem, concerning users being simultaneously in their critical regions, that cannot be obtained with a dining philosophers algorithm.

The *more-concurrent* drinking philosophers problem for $\mathcal{B}$ is the external schedule module that is the same as the drinking philosophers problem except that in addition to the exclusion implication, schedules must satisfy the following n implications:

More concurrent for i, $1 \leq i \leq n$: If α is drinking-well-formed and satisfies (NOV-B)$_i$, then α satisfies (NS-B)$_i$.

For each i, the implication "more concurrent for i" states that as long as the no-overlap condition is true for i, i will eventually enter its critical region, *even if*

some user j, with $B_i \cap B_j \neq \emptyset$, stays in its critical region forever (of course, j must have only resources not needed by i). Thus, simply using a dining philosophers algorithm for $\mathcal{B}$ would not satisfy this implication, since user i would be stuck in its trying region forever.

The no-lockout and more-concurrent drinking philosophers problems are incomparable in the sense that there is an algorithm for the first that does not solve the second and vice-versa.

3. Drinking Philosophers Automaton

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 several automata, to be described, and then hiding most of the actions, in order for the external actions to be consistent with the definition of the problem. The component automata are $D(i)$, for $1 \leq i \leq n$, and any automaton *Dine*($\mathcal{B}$) that solves the dining philosophers problem for $\mathcal{B}$. $D(i)$ represents the part of the drinking philosophers algorithm for user i; *Dine*($\mathcal{B}$) is the subroutine. First we describe the algorithm informally, then we present the $D(i)$ automata, and then we define *Drink*($\mathcal{B}$).

When drinker i enters its trying region needing a certain set of resources, it sends requests for those that it needs but lacks. Recipient j of a request satisfies the request unless j currently also wants the resource or is already using it. In the latter two cases, j defers the request and satisfies it once 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 drinker i is able to do so in its drinking trying region (without violating dining-well-formedness), it enters its dining trying region, that is, it tries to gain priority for all its adjacent resources. If 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 j of a demand must satisfy it even if j wants the resource, unless j is using the resource. In the latter case, j defers the request and satisfies it when j is through using the resource.

Once drinker 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. When i enters its drinking critical region, it relinquishes its dining critical region. The benefits of having the priorities are no longer needed. Doing so allows some extra concurrency: even if 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 old leftover demands to be in the system, so before satisfying or deferring a demand, the recipient must check that it has the resource.

Another point concerns when the actions of the dining subroutine should be performed. Drinker i's dining output actions are T_i and E_i and are enabled (using Boolean flags) in such a way as to preserve dining-well-formedness. Some drinkers could be locked out if drinker i never relinquishes the dining critical region. To avoid this situation, i cannot enter its drinking critical or remainder regions as long as E_i is enabled. The fairness assumption about the underlying model ensures that once E_i is enabled eventually i enters its dining exit region, after which it may enter the appropriate drinking region.

We now present the automaton $D(i)$.

The set of possible *messages* is $\{req(b), sat(b), dem(b) : b \in \mathcal{S}\}$.

The state of $D(i)$, $1 \leq i \leq n$, consists of values for the following variables: *drink-region*(i), *dine-region*(i), *deferred*(i), *bottles*(i), *req-bottles*(i), *buff*(i,j) for all $j \neq i$, *do-T*(i), and *do-E*(i). The *region*(i) variables take on the values T, C, E and R, and indicate which region the i^{th} dining and i^{th} drinking philosophers are in. The *deferred*(i) variable is a set of pairs (b, j), indicating that user j's request for bottle b has been deferred at user i. The *bottles*(i) and *req-bottles*(i) variables are sets of bottles, and indicate which bottles user i has and which it requires, respectively. For each $j \neq i$, the variable *buff*(i,j) is a FIFO queue of messages from $D(i)$ to send to $D(j)$, and is manipulated with operations enqueue and dequeue. The *do-T*(i) and *do-E*(i) variables are Booleans and control when the output actions T_i and E_i respectively are enabled. In the unique start state, the *region*(i)'s are R; *deferred*(i), *req-bottles*(i), and all the *buff*(i,j) are empty; *do-T*(i) and *do-E*(i) are false; and *bottles*(i) is an arbitrary subset of B_i. (We have actually defined a class of automata $D(i)$, with different start states, depending on the initial value of *bottles*(i). Later, we will require consistency between the $D(i)$'s.)

The actions of $D(i)$ are listed below, together with their preconditions and effects. (There are no internal actions.) First we define two macros, *SAT* and *DEFER*. *SAT* satisfies a request or demand from $D(j)$ for bottle b by sending the message $sat(b)$ to $D(j)$ and removing b from $bottles(i)$ and (b,j) from $deferred(i)$. *DEFER* defers a request or demand from $D(j)$ for bottle b, if b is currently required by $D(i)$, by adding (b,j) to $deferred(i)$; if b is not currently required, then the request or demand is satisfied.

$SAT(i,b,j)$ == enqueue($buff(i,j)$,$sat(b)$)
$bottles(i) \leftarrow bottles(i) - \{b\}$
$deferred(i) \leftarrow deferred(i) - \{(b,j)\}$

$DEFER(i,b,j)$ == if $b \in req\text{-}bottles(i)$ then $deferred(i) \leftarrow deferred(i) \cup \{(b,j)\}$
else $SAT(i,b,j)$

Input actions:

- $T_i(B)$, $B \subseteq B_i$
  Effect:
  $drink\text{-}region(i) \leftarrow T$
  $req\text{-}bottles(i) \leftarrow B$
  for all $j \neq i$ and $b \in req\text{-}bottles(i) \cap B_j$:
  if $b \notin bottles(i)$ then enqueue($buff(i,j)$,$req(b)$)
  if $dine\text{-}region(i) = R$ then $do\text{-}T(i) \leftarrow$ true

- $E_i(B)$, $B \subseteq B_i$
  Effect:
  $drink\text{-}region(i) \leftarrow E$
  for all $(b,j) \in deferred(i)$: $SAT(i,b,j)$

- $deliver(sat(b),j,i)$ for all $j \neq i$, $b \in B_i \cap B_j$
  Effects:
  $bottles(i) \leftarrow bottles(i) \cup \{b\}$

- $deliver(req(b),j,i)$ for all $j \neq i$, $b \in B_i \cap B_j$
  Effects:
  if $drink\text{-}region(i) = T$ or $drink\text{-}region(i) = C$
  then $DEFER(i,b,j)$
  else $SAT(i,b,j)$

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

Effect:
if $b \in bottles(i)$ then
if $drink\text{-}region(i) = C$ or ($drink\text{-}region(i) = T$ & $dine\text{-}region(i) = C$)
then $DEFER(i, b, j)$
else $SAT(i, b, j)$

- C_i

Effect:
$dine\text{-}region(i) \leftarrow C$
if $drink\text{-}region(i) = T$ then [
for all $j \neq i$ and $b \in req\text{-}bottles(i) \cap B_j$: if $b \notin bottles(i)$ then
enqueue($buff(i, j), dem(b)$)]
else $do\text{-}E(i) \leftarrow$ true

- R_i

Effect:
$dine\text{-}region(i) \leftarrow R$
if $drink\text{-}region(i) = T$ then $do\text{-}T(i) \leftarrow$ true

Output actions:

- $C_i(B)$, $B \subseteq B_i$

Precondition:
$drink\text{-}region(i) = T$
$B = req\text{-}bottles(i) \subseteq bottles(i)$
$do\text{-}E(i) =$ false
Effect:
$drink\text{-}region(i) \leftarrow C$
if $dine\text{-}region(i) = C$ then $do\text{-}E(i) \leftarrow$ true

- $R_i(B)$, $B \subseteq B_i$

Precondition:
$drink\text{-}region(i) = E$
$B = req\text{-}bottles(i)$
$do\text{-}E(i) =$ false
Effect:
$drink\text{-}region(i) \leftarrow R$

- $deliver(m, i, j)$ for all $j \neq i$, $m \in \{req(b), sat(b), dem(b) : b \in B_i \cap B_j\}$

Precondition:

m is at head of $buff(i,j)$
Effect:
dequeue($buff(i,j)$)

- T_i

  Precondition:
  $do\text{-}T(i) = \text{true}$
  Effect:
  $dine\text{-}region(i) \leftarrow T$
  $do\text{-}T(i) \leftarrow \text{false}$

- E_i

  Precondition:
  $do\text{-}E(i) = \text{true}$
  Effect:
  $dine\text{-}region(i) \leftarrow E$
  $do\text{-}E(i) \leftarrow \text{false}$

The output actions of $D(i)$ are partitioned into n classes, one for the delivery of messages in each $buff(i,j)$, and one for all the other actions. Formally, the subsets of the output actions are $\{C_i(B), R_i(B), T_i, E_i : B \subseteq B_i\}$, and for each $j \neq i$, $\{deliver(m,i,j) : m = req(b), dem(b), \text{or } sat(b), b \in B_i \cap B_j\}$. This partition guarantees that messages are eventually delivered in fair executions, since the message queues are FIFO. In essence, the *buff* variables are modeling separate pieces of hardware, the communication links.

A set of automata $\{D(i) : 1 \leq i \leq n\}$ is *resource-compatible* if for all i, and all b in B_i: b is in $bottles(j)$ in the start state of $D(j)$ for exactly one j. Let $Dine(\mathcal{B})$ be an automaton whose input actions are $\{T_i, E_i : 1 \leq i \leq n\}$ and whose output actions are $\{C_i, R_i : 1 \leq i \leq n\}$. The automaton $Drink(\mathcal{B})$ is formed by composing a resource-compatible set $\{D(i) : 1 \leq i \leq n\}$, and $Dine(\mathcal{B})$, and then hiding all actions except $\bigcup_{i=1}^{n} B\text{-}TCER_i$. See Figure 1.

4. Proof of Correctness

In Section 4.1, we show that $Drink(\mathcal{B})$ solves the drinking philosophers problem, that is, the safety properties are true, regardless of the behavior of the $Dine(\mathcal{B})$ subroutine. Section 4.2 consists of the proof that $Drink(\mathcal{B})$ solves the no-deadlock (resp., no-lockout) drinking philosophers problem if the $Dine(\mathcal{B})$ subroutine solves the no-deadlock (resp., no-lockout) dining philosophers problem. In Section 4.3

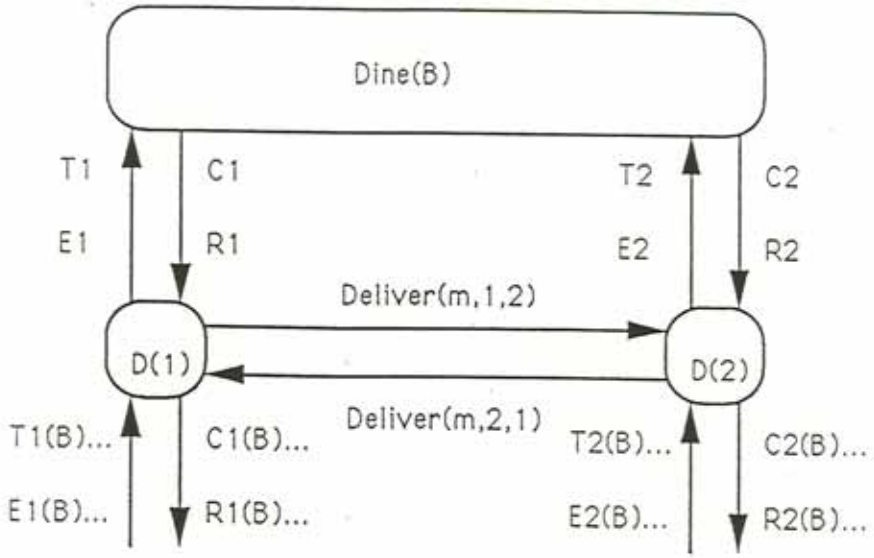

Figure 1: *Drink*($\mathcal{B}$), a system of two drinking philosophers.

we show that *Dine*($\mathcal{B}$) solves the more-concurrent drinking philosophers problem if the *Dine*($\mathcal{B}$) subroutine preserves dining-well-formedness. The proofs rely heavily on invariants about the states of the automata. The proofs of the invariants are relegated to Appendix B.

4.1 Drinking Philosophers

We show that *Drink*($\mathcal{B}$) solves the drinking philosophers problem, regardless of the behavior of *Dine*($\mathcal{B}$). That is, we show that correct exclusion is maintained by the algorithm, although no liveness properties are guaranteed. Three lemmas are used in the proof of the main result, Theorem 4. Lemma 1 states some simple relationships between states and actions in an execution, for example, *drink-region*(i) and *dine-region*(i) reflect the most recent drinking and dining actions at node i. Lemma 2 asserts that *Drink*($\mathcal{B}$) preserves drinking-well-formedness. Lemma 3 consists of several invariants needed to show the exclusion property.

Let $buff(i,j)|b$ be the subsequence of $buff(i,j)$ consisting exactly of $sat(b)$, $req(b)$ and $dem(b)$.

Lemma 1: *Let $e = s_0 a_1 s_1 \ldots$ be an execution of Drink($\mathcal{B}$). Choose any i and m, with $1 \leq i \leq n$ and s_m in e.*

(a) Let k be the largest integer such that $k \leq m$ and a_k is in B-TCER$_i$. (Let $k = 0$ if there is no such a_k.) If $k = 0$ or $a_k = R_i(B)$, then drink-region(i) = R

in s_m. *If* $a_k = T_i(B)$, *then drink-region*$(i) = T$ *in* s_m. *If* $a_k = C_i(B)$, *then drink-region*$(i) = C$ *in* s_m. *If* $a_k = E_i(B)$, *then drink-region*$(i) = E$ *in* s_m.

(b) Let k *be the largest integer such that* $k \leq m$ *and* $a_k = T_i(B)$ *for any* B. *(Let* $k = 0$ *if there is no such* a_k.*) If* $k = 0$, *then req-bottles*$(i) = \emptyset$ *in* s_m; *otherwise req-bottles*$(i) = B$ *in* s_m.

(c) Let k *be the largest integer such that* $k \leq m$ *and* a_k *is in F-TCER*$_i$. *(Let* $k = 0$ *if there is no such* a_k.*) If* $k = 0$ *or* $a_k = R_i$, *then dine-region*$(i) = R$ *in* s_m. *If* $a_k = T_i$, *then dine-region*$(i) = T$ *in* s_m. *If* $a_k = C_i$, *then dine-region*$(i) = C$ *in* s_m. *If* $a_k = E_i$, *then dine-region*$(i) = E$ *in* s_m.

(d) For all $j \neq i$, *buff*$(i,j)|b$ *is empty unless* b *is in* $B_i \cap B_j$, *in* s_m.

(e) If (b,j) *is in deferred*(i), *then* $j \neq i$ *and* b *is in* $B_i \cap B_j$, *for all* b *and* j, *in* s_m.

Proof: By an easy induction on m, inspecting the code. □

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

Proof: Suppose α is a schedule of *Drink*$(\mathcal{B})$, and βa is a prefix of α such that β is drinking-well-formed and a is a locally-controlled action of *Drink*$(\mathcal{B})$. We show that βa is drinking-well-formed.

Let e be any execution of *Drink*$(\mathcal{B})$ with schedule βa; let s be the state of e between β and a. There are two cases.

Case 1: $a = C_i(B)$ for some i and B. We must show that $\beta|B\text{-}TCER_i$ ends in $b = T_i(B)$. By precondition of $C_i(B)$, *drink-region*$(i) = T$ and $B =$ *req-bottles*(i) in s. By Lemma 1(a), $b = T_i(B')$ for some B', and by Lemma 1(b), $B' = B$.

Case 2: $a = R_i(B)$ for some i and B. We must show that $\beta|B\text{-}TCER_i$ ends in $b = E_i(B)$. By precondition of $R_i(B)$, *drink-region*$(i) = E$ and $B =$ *req-bottles*(i) in s. By Lemma 1(a), $b = E_i(B')$ for some B'. Since β is drinking-well-formed, $\beta|B\text{-}TCER_i$ ends in $T_i(B')C_i(B')E_i(B')$. By Lemma 1(b), $B' = B$. □

The following lemma states some invariants of the algorithm, that is, predicates true in every reachable state of *Drink*$(\mathcal{B})$. Recall that each bottle is in at most two B_i's.

Lemma 3: *Let* e *be an execution of Drink*$(\mathcal{B})$ *whose schedule is drinking-well-formed. Then in every state of* e, *the following are true, for all* i, j *and* b.

(A) If b is in $B_i \cap B_j$, $i \neq j$, then exactly one of the following is true: b is in bottles(i), or b is in bottles(j), or sat(b) is in buff(i,j), or sat(b) is in buff(j,i). If b is in B_i only, then b is in bottles(i).

(B) If (b,j) is in deferred(i), then
- *(a) b is in bottles(i),*
- *(b) drink-region$(j) = T$, and*
- *(c) b is in req-bottles(j).*

(C) If req(b) is at the head of buff$(i,j)|b$, then b is in bottles(j).

(D) If req(b) is in buff(i,j), then
- *(a) at most one req(b) is in buff(i,j),*
- *(b) no sat(b) follows it in buff(i,j),*
- *(c) (b,i) is not in deferred(j),*
- *(d) drink-region$(i) = T$,*
- *(e) b is in req-bottles(i), and*
- *(f) b is not in bottles(i).*

(E) If sat(b) is in buff(i,j), then
- *(a) at most one sat(b) is in buff(i,j),*
- *(b) no dem(b) immediately follows it in buff$(i,j)|b$,*
- *(c) drink-region$(j) = T$, and*
- *(d) b is in req-bottles(j).*

(F) If dem(b) is at the head of buff$(i,j)|b$ and b is in bottles(j), then (b,i) is in deferred(j).

(G) If drink-region$(i) = T$ and b is in req-bottles(i) and b is in B_j, $j \neq i$, then exactly one of the following is true: req(b) is in buff(i,j), or (b,i) is in deferred(j), or sat(b) is in buff(j,i), or b is in bottles(i).

(H) If b is in req-bottles(i) and drink-region$(i) = C$, then b is in bottles(i).

Proof: In Appendix B. □

Here is the main theorem.

Theorem 4: *Drink$(\mathcal{B})$ solves the drinking philosophers problem for $\mathcal{B}$.*

Proof: *Drink$(\mathcal{B})$* has the correct input and output actions by inspection and preserves drinking-well-formedness by Lemma 2.

Let e be a fair execution of *Drink*($\mathcal{B}$) with schedule α. We verify the exclusion implication.

Suppose α is drinking-well-formed. We must show α satisfies (EX-B). Suppose in contradiction that $\alpha = \alpha_1 C_i(B)\alpha_2 C_j(B')\alpha_3$ for some i and j, with $B \cap B' \neq \emptyset$, yet α_2 contains no $E_i(B)$. By drinking-well-formedness, $i \neq j$. Let s be the first state of α_3.

Since α is drinking-well-formed, $\alpha_1|B\text{-}TCER_i$ ends in $T_i(B)$, and $\alpha_2|B\text{-}TCER_i$ is empty. By Lemma 1(a) and (b), *drink-region*$(i) = C$, and $B =$ *req-bottles*(i) in s. Thus by Lemma 3 (H), $B \subseteq$ *bottles*(i) in s.

Again by drinking-well-formedness, $\alpha_1 C_i(B)\alpha_2|B\text{-}TCER_j$ ends in $T_j(B')$. By Lemma 1(a) and (b), *drink-region*$(j) = C$, and $B' =$ *req-bottles*(j) in s. Thus by Lemma 3 (H), $B' \subseteq$ *bottles*(j) in s.

But since $B \cap B' \neq \emptyset$, there is some b in $B \cap B'$, and thus in $B_i \cap B_j$, such that b is in *bottles*(i) and b is in *bottles*(j) in s, contradicting Lemma 3 (A). Thus α satisfies (EX-B).

We conclude that *Drink*($\mathcal{B}$) solves the drinking philosophers problem. □

4.2 No Deadlock and No Lockout

In this subsection we show that *Drink*($\mathcal{B}$) solves the no-deadlock (resp., no-lockout) drinking philosophers problem if *Dine*($\mathcal{B}$) solves the no-deadlock (resp., no-lockout) dining philosophers problem.

Lemma 5 consists of some invariants that are useful in doing the liveness proofs. Lemma 6 is a technical lemma relating to dining-well-formedness. Lemma 7 states that *Dine*($\mathcal{B}$) behaves properly in the composition, which means that the appropriate implications are true (e.g., exclusion and no-deadlock for dining, if *Dine*($\mathcal{B}$) solves the no-deadlock dining philosophers problem). Lemma 8 states that if all bottles are eventually released, then all forks are eventually released. The heart of Lemma 8 is showing that once a process in its drinking trying region enters its dining critical region, it subsequently enters its drinking critical region and releases its forks. Showing this depends on the dining exclusion implication (Lemma 7).

Lemma 9 is the key lemma and states that the no-deadlock implication for dining philosophers implies the no-deadlock implication for drinking philosophers (if all bottles are eventually released), and similarly for no-lockout. Lemma 9 is

proved as follows. Since all bottles are released, Lemma 8 implies that all forks are released. Then Lemma 7 implies that (ND-F) (or NL-F as appropriate) is true, which in turn implies that eventually the dining critical region is entered and the drinking critical region is entered. Theorems 10 and 11 put the pieces together.

Lemma 5: *Let e be an execution of $Drink(\mathcal{B})$ whose schedule is drinking-well-formed. Then in every state of e, the following are true, for all i.*

(A) If do-$T(i)$ is true, then dine-region$(i) = R$.

(B) If do-$E(i)$ is true, then dine-region$(i) = C$.

Proof: In Appendix B. □

Lemma 6: *Let e be an execution of $Drink(\mathcal{B})$ whose schedule α is drinking-well-formed. If $Dine(\mathcal{B})$ preserves dining-well-formedness, then*

(a) α is dining-well-formed, and

(b) for any i, if $\alpha|B\text{-}TCER_i$ is finite, then $\alpha|F\text{-}TCER_i$ is finite.

Proof: (a) We show α is dining-well-formed by induction on the length of its prefixes. The empty prefix is obviously dining-well-formed. Let βa be a prefix of α such that β is dining-well-formed. Let e be any execution of $Drink(\mathcal{B})$ with schedule βa; let s be the state of e between β and a.

Case 1: $a = T_i$ for some i. By precondition of T_i, do-$T(i)$ is true in s. By Lemma 5 (A), dine-region$(i) = R$ in s. By Lemma 1(c), $\beta|F\text{-}TCER_i$ either ends in R_i or is empty.

Case 2: $a = C_i$ for some i. Since $Dine(\mathcal{B})$ preserves dining-well-formedness, $\beta|F\text{-}TCER_i$ ends in T_i.

Case 3: $a = E_i$ for some i. By precondition of E_i, do-$E(i)$ is true in s. By Lemma 5 (B), dine-region$(i) = C$ in s. By Lemma 1(c), $\beta|F\text{-}TCER_i$ ends in C_i.

Case 4: $a = R_i$ for some i. Since $Dine(\mathcal{B})$ preserves dining-well-formedness, $\beta|F\text{-}TCER_i$ ends in E_i.

(b) Assume in contradiction that for some i, $\alpha|B\text{-}TCER_i$ is finite but $\alpha|F\text{-}TCER_i$ is infinite.

Case 1: $\alpha|B\text{-}TCER_i$ ends with $T_i(B)$ for some B. By Lemma 1, *drink-region*$(i) = T$ for the remainder of e. By dining-well-formedness, some C_i action occurs in e after the final $T_i(B)$. By dining-well-formedness and Lemmas 1 and 5 (B), *do-E*(i) is false when this C_i occurs. By the code, *do-E*(i) never becomes true after this point, since *drink-region*$(i) = T$ when the C_i occurs and no $C_i(B)$ action occurs subsequently. Thus there is no subsequent E_i action in e, contradicting dining-well-formedness.

Case 2: $\alpha|B\text{-}TCER_i$ ends with $C_i(B)$, $E_i(B)$ or $R_i(B)$ for some B. By Lemma 1, *drink-region*(i) is never equal to T in the remainder of e. By dining-well-formedness, some R_i action occurs in e after the final action in $B\text{-}TCER_i$. By dining-well-formedness and Lemmas 1 and 5 (A), *do-T*(i) is false when this R_i occurs. By the code, *do-T*(i) never becomes true after this point, since *drink-region*(i) is not T when the R_i occurs and no $T_i(B)$ action occurs subsequently. Thus there is no subsequent T_i action in e, contradicting dining-well-formedness. □

Lemma 7 shows that *Dine*$(\mathcal{B})$ behaves properly in the composition.

Lemma 7: *Let e be a fair execution of Drink$(\mathcal{B})$ whose schedule α is drinking-well-formed.*

(a) Suppose Dine$(\mathcal{B})$ solves the dining philosophers problem. Then α satisfies (EX-F).

(b) Suppose Dine$(\mathcal{B})$ solves the no-deadlock dining philosophers problem. If α satisfies (REL-F), then α satisfies (EX-F) and (ND-F).

(c) Suppose Dine$(\mathcal{B})$ solves the no-lockout dining philosophers problem. If α satisfies (REL-F), then α satisfies (EX-F) and (NL-F).

Proof: In all three cases, Lemma 6(a) implies that α is dining-well-formed. Let $e' = e|Dine(\mathcal{B})$ and $\alpha' = sched(e')$. Thus α' is also dining-well-formed, and if α satisfies (REL-F), then so does α'. By a result in [LT], e' is a fair execution of *Dine*$(\mathcal{B})$. Thus α' satisfies (EX-F) and either (ND-F) or (NL-F) (as appropriate), and so does α. □

Next we show that if all bottles are eventually released, then all forks are eventually released.

Lemma 8: *Let e be a fair execution of Drink$(\mathcal{B})$ whose schedule α is drinking-well-formed and satisfies (REL-B). If Dine$(\mathcal{B})$ solves the dining philosophers problem for $\mathcal{B}$, then α satisfies (REL-F).*

Proof: We must show that for all i, $\alpha|F\text{-}TCER_i$ does not end in C_i. Lemma 6(a) implies that α is dining-well-formed. By Lemma 7(a), α satisfies (EX-F). Suppose in contradiction that for some i, $e = e_1 C_i e_2$, where no action from $F\text{-}TCER_i$ occurs in e_2. By Lemma 1(c), *dine-region*$(i) = C$ throughout e_2. Let s be the last state of e_1.

Case 1: *drink-region*$(i) = C$, E or R in s. By the code, *do-E*(i) = true throughout e_2. Thus $C_i(B)$ and $R_i(B)$ are disabled for all B throughout e_2 and hence never occur in e_2. By assumption, T_i and E_i never occur in e_2. Yet E_i is enabled throughout e_2, contradicting e being fair.

Case 2: *drink-region*$(i) = T$ in s. Let *req-bottles*$(i) = B$ in s. If *do-E*(i) ever becomes true in e_2, then the same argument as in Case 1 gives a contradiction. Thus *do-E*(i) never becomes true in e_2. By the code, then, $C_i(B')$ never occurs in e_2 for any B', and by Lemma 1(a) and (b) and drinking-well-formedness, no $E_i(B')$ occurs, *drink-region*$(i) = T$, and *req-bottles*$(i) = B$ throughout e_2.

At the beginning of e_2, $D(i)$ sends *dem*(b) for all b in B that it is still missing. We now show that eventually every missing bottle will be in *bottles*(i). By fairness of e, each *dem*(b) is eventually received. Consider recipient $D(j)$.

Case 2.1: $b \notin$ *bottles*(j) when *dem*(b) is received by $D(j)$. Throughout e_2, $D(i)$ never adds *sat*(b) to *buff*(i,j), since requests and demands are deferred and no $E_i(B')$ occurs. Since the queues are FIFO, Lemma 3 (A) implies that the only possibilities when *dem*(b) is received are that b is in *bottles*(i) or *sat*(b) is in *buff*(j,i).

Case 2.2: $b \in$ *bottles*(j) when *dem*(b) is received by $D(j)$. By the code, there are only two situations in which *sat*(b) is not immediately added to *buff*(j,i).

Case 2.2.1: *drink-region*$(j) = C$ and $b \in$ *req-bottles*(j) when *dem*(b) is received by $D(j)$. By (REL-B), eventually some $E_j(B')$ occurs subsequently in e_2 and thus by the code *sat*(b) is added to *buff*(j,i) then.

Case 2.2.2: *drink-region*$(j) = T$ and *dine-region*$(j) = C$ and $b \in$ *req-bottles*(j) when *dem*(b) is received by $D(j)$. Since α satisfies (EX-F), *dine-region*(j) can never be C in e_2 by dining-well-formedness and Lemma 1(c), and this case cannot occur.

In both Cases 2.1 and 2.2, by fairness of e, the *sat*(b) message eventually arrives at $D(i)$ in e_2.

Since e_2 contains no $C_i(B)$ action, by drinking-well-formedness no $R_i(B')$ or $C_i(B')$ occurs in e_2 for any B'. Yet once any bottle in B is in *bottles*(i) in e_2, it

stays there for the rest of e_2. Thus after some point in e_2, $C_i(B)$ is continuously enabled, yet no action in that class of the partition occurs, contradicting e being fair. □

The next lemma states that no-deadlock for forks implies no-deadlock for bottles, and similarly for no-lockout.

Lemma 9: *Let e be a fair execution of Drink($\mathcal{B}$) whose schedule α is drinking-well-formed and satisfies (REL-B). If Dine($\mathcal{B}$) solves the no-deadlock (resp., no-lockout) dining philosophers problem, then α satisfies (ND-B) (resp., (NL-B)).*

Proof: By Lemma 6(a), α is dining-well-formed. If *Dine*($\mathcal{B}$) solves either the no-deadlock or the no-lockout dining philosophers problem, then *Dine*($\mathcal{B}$) obviously solves the dining philosophers problem, and by Lemma 8, α satisfies (REL-F).

Suppose in contradiction that α does not satisfy (ND-B) (resp., (NL-B)), i.e., there exists an i such that $\alpha|B\text{-}TCER$ (resp., $\alpha|B\text{-}TCER_i$) is finite and $\alpha|B\text{-}TCER_i$ ends in $E_i(B)$ or $T_i(B)$ for some B. (Ending in $C_i(B)$ is ruled out by (REL-B).)

We now show that $\alpha|F\text{-}TCER_i$ ends in R_i.

No-deadlock: Since $\alpha|B\text{-}TCER$ is finite, $\alpha|F\text{-}TCER$ is also finite by Lemma 6(b). By Lemma 7(b), α satisfies (ND-F), implying that $\alpha|F\text{-}TCER_i$ ends in R_i.

No-lockout: Since $\alpha|B\text{-}TCER_i$ is finite, $\alpha|F\text{-}TCER_i$ is also finite by Lemma 6(b). By Lemma 7(c), α satisfies (NL-F), implying that $\alpha|F\text{-}TCER_i$ ends in R_i.

We now show that both possibilities for the final action in $\alpha|B\text{-}TCER_i$ lead to a contradiction.

Case 1: $\alpha|B\text{-}TCER_i$ ends in $T_i(B)$ for some B. By Lemma 1(a), *drink-region*$(i) = T$ for the rest of e. Since $\alpha|F\text{-}TCER_i$ ends in R_i, *dine-region*$(i) = R$ for the rest of e by Lemma 1. If the final R_i occurs before the final $T_i(B)$, then *do-T*(i) is set to true when the $T_i(B)$ occurs. If the final R_i occurs after the final $T_i(B)$, then *do-T*(i) is set to true when the R_i occurs. In both cases, after some point, *do-T*(i) is true for the rest of e. Thus after some point in e, T_i is continuously enabled, yet no action from that class of the partition occurs, contradicting e being fair.

Case 2: $\alpha|B\text{-}TCER_i$ ends in $E_i(B)$ for some B. After this point, *drink-region*(i) remains E and *req-bottles*(i) remains B, by Lemma 1. Since $\alpha|F\text{-}TCER_i$ ends in

R_i, after some point in e *dine-region*(i) remains R by Lemma 1. Thus by Lemma 5 (B), *do-E*(i) remains false. So after some point in e, $R_i(B)$ is continuously enabled, yet no action in that class of the partition occurs, contradicting e being fair. □

The main theorems follow.

Theorem 10: *If Dine($\mathcal{B}$) solves the no-deadlock dining philosophers problem for $\mathcal{B}$, then Drink($\mathcal{B}$) solves the no-deadlock drinking philosophers problem for $\mathcal{B}$.*

Proof: *Drink*$(\mathcal{B})$ has the correct input and output actions by inspection and preserves drinking-well-formedness by Lemma 2.

Let e be a fair execution of *Drink*$(\mathcal{B})$. We verify the exclusion and no-deadlock implications. The exclusion implication is true by the same argument as in the proof of Theorem 4. The no-deadlock implication is true by Lemma 9. □

Theorem 11: *If Dine($\mathcal{B}$) solves the no-lockout dining philosophers problem for $\mathcal{B}$, then Drink($\mathcal{B}$) solves the no-lockout drinking philosophers problem for $\mathcal{B}$.*

Proof: Analogous to the proof of Theorem 10. □

4.3 Concurrent Drinking

In this subsection we show that *Drink*$(\mathcal{B})$ solves the more-concurrent drinking philosophers problem, regardless of the behavior of *Dine*$(\mathcal{B})$ (as long as it preserves dining-well-formedness). In essence, the condition $(\text{NOV-B})_i$ is so strong that the dining subroutine is not needed to arbitrate disputes. Lemma 12 proves several invariants about *dem*(b) messages and is used in the proof of the next lemma (as well as in the complexity analysis). Lemma 13 is the main one, stating that the no-overlap condition implies the never-stuck condition. Theorem 14 puts the pieces together.

A *dem*(b) message in *buff*(i,j) is *current* if one of the following is true: a *sat*(b) message precedes it in *buff*(i,j), or b is in *bottles*(j), or a *sat*(b) message is in *buff*(j,i).

Lemma 12: *Suppose Dine($\mathcal{B}$) preserves dining-well-formedness. Let e be an execution of Drink($\mathcal{B}$) whose schedule is drinking-well-formed. The following predicates are true in every state of e, for any i, j and b.*

(A) If there is a current dem(b) message in buff(i,j), then

(a) *drink-region*$(i) = T$,
(b) *dine-region*$(i) = C$,
(c) b *is in req-bottles*(i), *and*
(d) *do-E*(i) *is false.*

(B) There is at most one current dem(b) *message in buff*(i, j).

(C) There is at most one non-current dem(b) *message in buff*(i, j).

Proof: In Appendix B. □

Lemma 13: *Let e be a fair execution of Drink($\mathcal{B}$) whose schedule α is drinking-well-formed and satisfies (NOV-B)$_i$, for some fixed i. If Dine($\mathcal{B}$) preserves dining-well-formedness, then α satisfies (NS-B)$_i$.*

Proof: Recall that (NOV-B)$_i$ states that for all j and any B and B' with $B \cap B' \neq \emptyset$, the following two conditions hold: (1) if $\alpha = \beta_1 T_i(B) \beta_2 T_j(B) \beta_3$, then β_2 contains $C_i(B)$; and (2) if $\alpha = \beta_1 T_j(B') \beta_2 T_i(B) \beta_3$, then β_2 contains $E_j(B')$.

Suppose in contradiction to (NS-B)$_i$ that $\alpha | B\text{-}TCER_i$ ends in $T_i(B)$ or $E_i(B)$ for some B.

Case 1: $\alpha | B\text{-}TCER_i$ ends in $T_i(B)$. By (NOV-B)$_i$, drinking-well-formedness and Lemma 1(a), for all $j \neq i$, *drink-region*$(j) = E$ or R for the rest of e after the final $T_i(B)$. When the final $T_i(B)$ occurs, a request message for each bottle b in B that is not in *bottles*(i) is placed in the appropriate *buff*(i, j). Since e is fair, it is eventually delivered. By Lemma 3(c), b is in *bottles*(j) when the request is received and by the code $D(j)$ immediately satisfies the request. Since e is fair, the satisfy message is eventually delivered to $D(i)$.

We now show that once b is in *bottles*(i) after the final $T_i(B)$, it remains there. Since *drink-region*(j), $j \neq i$, is never equal to T after the final $T_i(B)$, Lemma 3 (D-d) implies that $D(i)$ never receives $req(b)$ after the final $T_i(B)$. Similarly, Lemma 11 (A-a) implies that $D(i)$ never receives a $dem(b)$ message for b in *bottles*(i) after the final $T_i(B)$. Thus there is a point in e after which every bottle in B is in *bottles*(i) and remains there.

By Lemma 6(b), $\alpha | F\text{-}TCER_i$ is finite. Consider the point in e after the latter of (1) the last action in $F\text{-}TCER_i$ and (2) the point after the final $T_i(B)$ when $B \subseteq$ *bottles*(i). If *do-E*(i) is true at this point, then E_i is continuously enabled for the rest of e, yet no action in that class of the partition occurs, contradicting e being

fair. If *do-E*(i) is false at this point, then $C_i(B)$ is continuously enabled for the rest of e, yet no action in that class of the partition occurs, contradicting e being fair.

Case 2: $\alpha|B\text{-}TCER_i$ ends in $E_i(B)$. By Lemma 6(b), $\alpha|F\text{-}TCER_i$ is finite. After the latter of the final action in $F\text{-}TCER_i$ and the final $E_i(B)$, *do-E*(i) is either true or false. If *do-E*(i) is true at this point, then E_i is continuously enabled for the rest of e, yet no action in that class of the partition occurs, contradicting e being fair. If *do-E*(i) is false at this point, then $C_i(B)$ is continuously enabled for the rest of e, yet no action in that class of the partition occurs, contradicting e being fair. □

Theorem 14: *If Dine*$(\mathcal{B})$ *preserves dining-well-formedness, then Drink*$(\mathcal{B})$ *solves the more-concurrent drinking philosophers problem for* $\mathcal{B}$.

Proof: *Drink*$(\mathcal{B})$ has the correct input and output actions by inspection and preserves drinking-well-formedness by Lemma 2.

Let e be a fair execution of *Drink*$(\mathcal{B})$ with schedule α. We verify the exclusion, and more-concurrent for i $(1 \leq i \leq n)$ implications. The exclusion implication is true by the same argument as in the proof of Theorem 4. The more-concurrent for i implications, $1 \leq i \leq n$, are true by Lemma 13. (Lemma 13 is applicable because Lemma 6(a) implies that α is dining-well-formed.) □

5. Complexity Analysis

In this section, we analyze the worst-case waiting time of our algorithm as well as evaluating it using the criteria listed in [CM]. The analysis of the worst-case waiting time shows that the limiting factor is the no-lockout dining philosophers subroutine. By replacing the $O(n)$ time subroutine of [CM] with an $O(1)$ time subroutine (for instance, that of [Ly]), 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. The following definitions provide a measure of time complexity for our model that is analogous to that in [PF], 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 timing definitions provide distinct upper bounds on process step time and on message delivery time.

Given an execution e of automaton A, where $e = s_0 a_1 s_1 a_2 \ldots$, a *timing function* for e is an increasing 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 a_i occurs; we rule out an infinite number of actions occurring before a finite real time.

Let f be a function mapping each class of the partition $part(A)$ to a positive real number. Execution e is *f-bounded* if the following condition is true for each class C of the partition $part(A)$. For each $i \geq 0$, either

(1) there exists $j > i$ such that a_j is in C and $t_e(j) - t_e(i) \leq f(C)$, or

(2) there exists $j \geq i$ such that no action of C is enabled in s_j and $t_e(j) - t_e(i) \leq f(C)$.

That is, starting at any point in the execution, within time $f(C)$ either some output action in C occurs, or else the automaton passes through a state in which no output action in C is enabled. Each class of the partition is considered separately, since each class corresponds, in some sense, to a distinct entity in a larger system.

Now we analyze the worst-case time behavior of the no-lockout drinking philosophers algorithm, automaton *Drink($\mathcal{B}$)*, which uses any no-lockout dining philosophers subroutine *Dine($\mathcal{B}$)* for $\mathcal{B}$. Let f map each class $\{C_i(B), R_i(B), T_i, E_i : B \subseteq B_i\}$ to some positive real c and each class $\{deliver(m,i,j) : m = req(b), dem(b)$ or $sat(b)\}$ to some positive real d. Thus, c is the upper bound on process step time and d is the upper bound on the message delay. Let $\mathcal{E}$ be the set of all fair f-bounded executions of *Drink($\mathcal{B}$)* whose schedules are drinking-well-formed and satisfy (REL-B).

Let try_{Drink} be the maximum time, over all i and all $B \subseteq B_i$, between any $T_i(B)$ action and the subsequent $C_i(B)$ action, in any execution in $\mathcal{E}$. Let $crit_{Drink}$ be the maximum time, over all i and all $B \subseteq B_i$, between any $C_i(B)$ action and the subsequent $E_i(B)$ action, in any execution in $\mathcal{E}$.

Let try_{Dine} be the maximum time over all i between any T_i action and the subsequent C_i action, in any execution in $\mathcal{E}$. Let $crit_{Dine}$ be the maximum time over all i between any C_i action and the subsequent E_i action, in any execution in $\mathcal{E}$. Let $exit_{Dine}$ be the maximum time over all i between any E_i action and the subsequent R_i action, in any execution in $\mathcal{E}$.

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

Theorem 16 gives an upper bound on try_{Drink}, the maximum time a user process must wait after requesting to enter its critical region until it is allowed to

do so. It is proved using Lemma 15, which bounds the number of messages in any $buff(i,j)$. The proof of Lemma 15 in turn uses Lemma 12.

First we show that there is a bounded number of messages in any *buff*. Let r be the maximum number of bottles shared by any two drinkers.

Lemma 15: *Suppose Dine($\mathcal{B}$) preserves dining-well-formedness. Let e be any execution of Drink($\mathcal{B}$) whose schedule is drinking-well-formed. Then in any state of e, there are at most $4r$ messages in $buff(i,j)$ for any i and j.*

Proof: Choose any i and j, $i \neq j$. Let s be any state in e. By Lemma 1(d), $buff(i,j)|b$ is empty unless b is in $B_i \cap B_j$. There are at most r bottles in $B_i \cap B_j$. Choose any such b. By (D-a) of Lemma 3, there is at most one $req(b)$ message in $buff(i,j)$ in s. By (E-a) of Lemma 3, there is at most one $sat(b)$ message in $buff(i,j)$ in s. By (B) of Lemma 12, there is at most one current $dem(b)$ message in $buff(i,j)$ in s. By (C) of Lemma 12, there is at most one non-current $dem(b)$ message in $buff(i,j)$ in s. Thus there are at most four messages in $buff(i,j)|b$. The result follows. □

The main theorem follows.

Theorem 16: $try_{Drink} \leq 3c + 8rd + exit_{Dine} + try_{Dine} + crit_{Drink}$.

Proof: Choose e in $\mathcal{E}$ and fix i. Suppose $T_i(B)$ occurs at time t, for some B. In the worst case, $dine\text{-}region(i) = C$ at time t. By time c later, E_i occurs, by time $exit_{Dine}$ later, R_i occurs, by time c later, T_i occurs, and by time try_{Dine} later, T_i occurs.

When this T_i occurs, $D(i)$ sends a $dem(b)$ message for all required and missing bottles. By Lemma 15, the demand is received by time $4rd$ later. As in the proof of Lemma 8 (Case 2.2.2), either the recipient immediately sends $sat(b)$ to $D(i)$ or else the recipient is in its drinking critical region and sends $sat(b)$ by time $crit_{Drink}$ later. By Lemma 15, the $sat(b)$ is received by time $4rd$ later. By time c later, $C_i(B)$ occurs. □

Since we assume that $crit_{Drink}$, $exit_{Dine}$, r, d and c 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} depends on $crit_{Dine}$. We now give an informal argument for an upper bound on $crit_{Dine}$. Once C_i occurs, E_i will not occur until after $D(i)$ has sent demands for needed bottles, these demands have been satisfied, and $D(i)$ has entered its drinking

critical region. The upper bound then is $2c + 8rd + crit_{Drink}$. Thus $crit_{Dine}$ is also a constant, under our assumptions.

The dining philosophers subroutine used by Chandy and Misra (1984) has try_{Dine} of $O(n)$. By replacing it with, for instance, the dining philosophers algorithm of Lynch (1981), which has worst-case waiting time of $O(1)$, we obtain a more efficient drinking philosophers algorithm. The algorithm of Lynch (1981) 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 satisfy messages took a set of bottles as arguments instead of a single bottle.

Five criteria for evaluating resource allocation algorithms are given by Chandy and Misra (1984) — 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 as long as the dining philosophers subroutine has it.

Symmetry means that each process runs the identical program. This property is true of our solution, as long as it is true of the subroutine.

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 $req(b)$ messages for all missing resource. It defers any $req(b)$ messages it receives when $drink\text{-}region(i) = T$, but yields to $dem(b)$ messages. When $dine\text{-}region(i)$ becomes C, it sends $dem(b)$ messages for any missing resources. Thus at most four messages ($req(b)$, $sat(b)$, $dem(b)$, $sat(b)$) are sent on behalf of any 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 certainly

true of our algorithm, since it satisfies the more-concurrent condition. More precise formulations of "concurrency" were given in our definitions (see Sections 3 and 6).

Boundedness means that the number of messages in any $buff(i,j)$ variable is bounded, and the size of each message is bounded. This is certainly true of our solution, by Lemma 15.

6. Conclusions

We have given precise definitions of several versions of the dining philosophers and drinking philosophers problems, each version satisfying different liveness and concurrency conditions. We described a modular drinking philosophers algorithm that used as a true 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 with a discussion of other versions of the drinking philosophers problem.

The version of the drinking philosophers problem specifying the most concurrency 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.) Unfortunately, neither the algorithm in this paper nor that of Chandy and Misra (1984) satisfies this conditions. An interesting problem would be to devise one that does.

The following situation shows that our algorithm does not solve the "most concurrent" drinking philosophers problem. (Essentially the same scenario shows that the algorithm of Chandy and Misra (1984) also does not.) 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 version of the drinking philosophers problem specifying a degree of concurrency intermediate between strongest and more-concurrent, that the algorithm of Chandy and Misra (1984) solves and ours does not. The informal description is 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. As in the previous scenario, 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 request is satisfied. Now 3 obtains d from 5. But 3 will never get c from 4, because it 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 Chandy and Misra (1984) 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. In other words, one can optimize to gain extra concurrency at the expense of violating modularity.

Acknowledgments

We thank Alan Fekete and Prasad Sistla for helpful discussions.

References

Chandy KM and Misra J (1984) The Drinking Philosophers Problem, *ACM Trans. on Programming Languages and Systems*, 6:632–646.

Dijkstra EW (1971) Hierarchical Ordering of Sequential Processes, *Acta Informatica*, 1:115–138.

Lynch NA (1981) Upper Bounds for Static Resource Allocation in a Distributed System, *JCSS*, 23:254–278.

Lynch NA and Tuttle MR (1987) Hierarchical Correctness Proofs for Distributed Algorithms, *Proc. 6^{th} Ann. ACM Symp. on Principles of Distributed Computing,*

pp. 137–151. (Also available as technical report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, 1987.)

Peterson GL and Fischer MJ (1977) Economical Solutions for the Critical Section Problem in a Distributed System, *Proc. 9^{th} Ann. ACM Symp. on Theory of Comp.*, pp. 91–97.

Appendix A

In this Appendix, we review the aspects of the model of Lynch and Tuttle (1987) that are relevant to this paper.

An *input-output automaton* A is defined by the following four components. (1) There is a (possibly infinite) set of *states* with a subset of *start states*. (2) There is a set of *actions*, associated with the state transitions. The actions are divided into three classes, *input*, *output*, and *internal*. Input actions are presumed to originate in the automaton's environment; consequently the automaton must be able to react to them no matter what state it is in. Output and internal actions (or, *locally-controlled* actions) are under the local control of the automaton; internal actions model events not observable by the environment. The input and output actions are the *external* actions of A, denoted $ext(A)$. (3) The transition relation is a set of (state, action, state) triples, such that for any state s' and input action π, there is a transition (s', π, s) for some state s. (4) There is an equivalence relation $part(A)$ partitioning the output and internal actions of A. The partition is meant to reflect separate pieces of the system being modeled by the automaton. Action π is *enabled* in state s' if there is a transition (s', π, s) for some state s.

An *execution* e of A is a finite or infinite sequence $s_0\pi_1 s_1 \ldots$ of alternating states and actions such that s_0 is a start state, (s_{i-1}, π_i, s_i) is a transition of A for all i, and if e is finite then e ends with a state. The *schedule* of an execution e is the subsequence of actions appearing in e.

We often want to specify a desired behavior using a set of schedules. Thus we define an *external schedule module* S to consist of a set of input actions, a set of output actions, and a set of schedules. Each schedule of S is a finite or infinite sequence of the actions of S. Internal actions are excluded in order to focus on the behavior visible to the outside world.

Let A be an automaton or schedule module and P be a predicate on sequences of actions of A. A *preserves* P if for every schedule βa of A such that P is true of β and a is a locally-controlled action of A, then P is also true of βa.

Automata can be composed to form another automaton, presumably modeling a system made of smaller components. Automata communicate by synchronizing on shared actions; the only allowed situations are for the output from one automaton to be the input to others, and for several automata to share an input. Thus, automata to be composed must have no output actions in common, and the internal actions of each must be disjoint from all the actions of the others. A state of the composite automaton is a tuple of states, one for each component. A start state of the composition has a start state in each component of the state. Any output action of a component becomes an output action of the composition, and similarly for an internal action. An input action of the composition is an action that is input for every component for which it is an action. In a transition of the composition on action π, each component of the state changes as it would in the component automaton if π occurred; if π is not an action of some component automaton, then the corresponding state component does not change. The partition of the composition is the union of the partitions of the component automata.

Given an automaton A and a subset Π of its actions, we define the automaton $Hide_{\Pi}(A)$ to be the automaton A' differing from A only in that each action in Π becomes an internal action. This operation is useful for hiding actions that model interprocess communication in a composite automaton, so that they are no longer visible to the environment of the composition.

An execution of a system is fair if each component is given a chance to make progress infinitely often. Of course, a process might not be able to take a step every time it is given a chance. Formally stated, execution e of automaton A is *fair* if for each class C of $part(A)$, the following two conditions hold. (1) If e is finite, then no action of C is enabled in the final state of e. (2) If e is infinite, then either actions from C appear infinitely often in e, or states in which no action of C is enabled appear infinitely often in e. Note that any finite execution of A is a prefix of some fair execution of A.

The following result from [LT] is very useful: If e is a fair execution of a composition of automata, and A is one of the components, then $e|A$ is a fair execution of A. (If $e = s_0\pi_1 s_1 \ldots$, we define $e|A$ to be the sequence obtained from e by deleting $\pi_i s_i$ if π_i is not an action of A, and replacing the remaining s_i with A's component.)

A *problem* is (specified by) an external schedule module. Automaton A *solves* the problem P if A and P have the same input and output actions, and if $\{\alpha | ext(A) :$ α is the schedule of a fair execution of $A\}$ is a subset of the set of schedules of P.

In other words, the behavior of A visible to the outside world is consistent with the behavior required in the problem specification.

Appendix B

This appendix contains the proofs of Lemmas 3, 5 and 12, all of which state that certain predicates are invariants.

Lemma 3: *Let e be an execution of Drink($\mathcal{B}$) whose schedule is drinking-well-formed. Then in every state of e, the following are true, for all i, j and b.*

(A) If b is in $B_i \cap B_j$, $i \neq j$, then exactly one of the following is true: b is in bottles(i), or b is in bottles(j), or sat(b) is in buff(i,j), or sat(b) is in buff(j,i). If b is in B_i only, then b is in bottles(i).

(B) If (b,j) is in deferred(i), then
- *(a) b is in bottles(i),*
- *(b) drink-region(j) $= T$, and*
- *(c) b is in req-bottles(j).*

(C) If req(b) is at the head of buff(i,j)$|b$, then b is in bottles(j).

(D) If req(b) is in buff(i,j), then
- *(a) at most one req(b) is in buff(i,j),*
- *(b) no sat(b) follows it in buff(i,j),*
- *(c) (b,i) is not in deferred(j),*
- *(d) drink-region(i) $= T$,*
- *(e) b is in req-bottles(i), and*
- *(f) b is not in bottles(i).*

(E) If sat(b) is in buff(i,j), then
- *(a) at most one sat(b) is in buff(i,j),*
- *(b) no dem(b) immediately follows it in buff(i,j)$|b$,*
- *(c) drink-region(j) $= T$, and*
- *(d) b is in req-bottles(j).*

(F) If dem(b) is at the head of buff(i,j)$|b$ and b is in bottles(j), then (b,i) is in deferred(j).

(G) If drink-region(i) $= T$ and b is in req-bottles(i) and b is in B_j, $j \neq i$, then exactly one of the following is true: req(b) is in buff(i,j), or (b,i) is in deferred(j), or sat(b) is in buff(j,i), or b is in bottles(i).

(H) If b is in req-bottles(i) and drink-region(i) = C, then b is in bottles(i).

Proof: Let $e = s_0 a_1 s_1 \ldots a_m s_m \ldots$. We proceed by induction on m, which indexes the states of e.

(A) through (H) are obviously true of s_0, since it is a start state of a composition of compatible automata. Assuming (A) through (H) are true of s_{m-1}, we show they are true of s_m. We consider every possible value of a_m.

Case 1: $a_m = T_i(B)$.

Claims about s_{m-1}:
1. *drink-region*(i) = R, by drinking-well-formedness and Lemma 1(a).
2. (b, i) is not in *deferred*(j), for all b and j, by Claim 1 and (B-b).
3. *sat*(b) is not in *buff*(j, i), for all b and j, by Claim 1 and (E-c).
4. *req*(b) is not in *buff*(i, j), for all b and j, by Claim 1 and (D-d).
5. If *sat*(b) is not in *buff*(i, j) and b is not in *bottles*(i), then b is in *bottles*(j), where b is in B_j, $j \neq i$, for all b, by Claim 3 and (A).
6. If *buff*(i, j) is empty and b is not in *bottles*(i), then b is in *bottles*(j), where b is in B_j, $j \neq i$, for all b, by Claim 5.

Claims about s_m:
7. *req*(b) is in *buff*(i, j) iff b is not in *bottles*(i) and b is in *req-bottles*(i) and b is in $B \cap B_j$, for all b and j, by Claim 4 and code.
8. If *req*(b) is at the head of *buff*(i, j) and b is not in *bottles*(i), then b is in *bottles*(j), for all b and j, by Claim 6 and code.
9. If *req*(b) is at the head of *buff*(i, j), then b is in *bottles*(j), for all b and j, by Claims 7 and 8.

(A) No relevant change.

(B) Only (B-c) is affected, for (b, i). By Claim 2 and code, no (b, i) is in *deferred*(j) in s_m, so the predicate is vacuously true.

(C) Only changes affect *req*(b) in *buff*(i, j); by Claim 9.

(D) Only changes affect *req*(b) in *buff*(i, j). (a) and (b) by Claim 4 and code. (c) by Claim 2 and code. (d) by code. (e) and (f) by Claim 7.

(E) Only (E-d) is affected, for *sat*(b) in *buff*(j, i). None by Claim 3 and code, so vacuously true.

(F) No relevant changes.

(G) Only changes involve i. Suppose b is in *req-bottles*(i) in s_m. By Claims 2 and 3 and code, we only need to show that $req(b)$ is in $buff(i,j)$ iff b is not in *bottles*(i), which is true by Claim 7.

(H) Only changes involve i. By code, *drink-region*$(i) = T$ in s_m, so vacuously true.

Case 2: $a_m = E_i(B)$.

Claims about s_{m-1}:
1. *drink-region*$(i) = C$, by drinking-well-formedness and Lemma 1(a).
2. (b,i) is not in *deferred*(j), for all b and j, by Claim 1 and (B-b).
3. $req(b)$ is not in $buff(i,j)$, for all b and j, by Claim 1 and (D-d).
4. $sat(b)$ is not in $buff(j,i)$, for all b and j, by Claim 1 and (E-c).
5. If (b,j) is in *deferred*(i), then b is in *bottles*(i), for all b and j, by (B-a).
6. If (b,j) is in *deferred*(i), then $j \neq i$, by Lemma 1(e).
7. If (b,j) is in *deferred*(i), then b is not in *bottles*(j), $sat(b)$ is not in $buff(i,j)$, and $sat(b)$ is not in $buff(j,i)$, for all b and j, by Claims 5 and 6 and (A).
8. If (b,j) is in *deferred*(i), then $req(b)$ is not in $buff(j,i)$, for all b and j, by Claim (D-c).
9. If (b,j) is in *deferred*(i), then *drink-region*$(j) = T$ and b is in *req-bottles*(j), for all b and j, by (B-b) and (B-c).
10. If (b,j) is in *deferred*(i), then $req(b)$ is not in $buff(j,i)$, for all b and j, by Claim 9 and (G).

(A) Only affects b such that (b,j) is in *deferred*(i) in s_{m-1}. By Claim 7 and code.

(B) Only affects *deferred*(i) and *deferred*(j). By Claim 2 and code, no (b,i) is in *deferred*(j), so vacuously true. By code, no (b,j) is in *deferred*(i) in s_m, so vacuously true.

(C) Only affects $buff(j,i)$, where (b,j) is in *deferred*(i) in s_{m-1}. By Claim 8 and code, no $req(b)$ is in $buff(j,i)$ in s_m, so vacuously true.

(D) Only affects $buff(i,j)$. By Claim 3 and code, no $req(b)$ is in $buff(i,j)$ in s_m, so vacuously true.

(E) Only affects $buff(i,j)$ such that (b,j) is in $deferred(i)$ in s_{m-1}, and $buff(j,i)$ for all j. By Claim 4 and code, no $sat(b)$ is in $buff(j,i)$ in s_m, so vacuously true. Suppose $sat(b)$ is added to $buff(i,j)$ in s_m, Then (b,j) is in $deferred(i)$ in s_{m-1}. (a) By Claim 7 and code. (b) By code. (c) and (d) By Claim 9 and code.

(F) Only affects i. Since (F) is true in s_{m-1} and by code b is removed from $bottles(i)$ if and only if b is removed from $deferred(i)$ in s_m, still true.

(G) Only affects j, where (b,j) is in $deferred(i)$ in s_{m-1}. By Claim 10 and code, $req(b)$ is not in $buff(j,i)$ in s_m. By Claim 7 and code, b is not in $bottles(j)$ in s_m. By code, (b,j) is not in $deferred(i)$ and $sat(b)$ is in $buff(j,i)$ in s_m.

(H) Only affects i. By code, $drink\text{-}region(i) = E$ in s_m, so vacuously true.

Case 3: $a_m = deliver(sat(b), j, i)$.

Claims about s_{m-1}:

1. $sat(b)$ is at the head of $buff(j,i)$, by precondition.
2. b is in $B_i \cap B_j$, by Claim 1 and Lemma 1(d).
3. b is not in $bottles(i)$, by Claims 1 and 2 and (A).
4. b is not in $bottles(j)$, by Claims 1 and 2 and (A).
5. $sat(b)$ is not in $buff(i,j)$, by Claims 1 and 2 and (A).
6. At most one $sat(b)$ is in $buff(j,i)$, by Claim 1 and (E-a).
7. No $dem(b)$ immediately follows $sat(b)$ in $buff(j,i)$, by Claim 1 and (E-b).
8. $drink\text{-}region(i) = T$, by Claim 1 and (E-c).
9. b is in $req\text{-}bottles(i)$, by Claim 1 and (E-d).
10. $req(b)$ is not in $buff(i,j)$, by Claims 1, 8 and 9 and (G).
11. (b,i) is not in $deferred(j)$, by Claims 1, 8 and 9 and (G).
12. b is not in $bottles(i)$, by Claims 1, 8 and 9 and (G).

(A) Only affects b. By Claims 4, 5 and 6 and code.

(B) No relevant change.

(C) Only affects $buff(j,i)|b$. By code, since b is added to $bottles(i)$.

(D) Only affects b. By Claim 10 and code, no $req(b)$ is in $buff(i,j)$, so vacuously true.

(E) No relevant change.

(F) Only affects $buff(j,i)$. By Claim 7 and code, $dem(b)$ is not at head of $buff(j,i)$, so vacuously true.

(G) Only affects b and i. By Claims 6, 10 and 11 and code.

(H) No relevant change.

Case 4: $a_m = deliver(req(b), j, i)$.

Claims about s_{m-1}*:*
1. $req(b)$ is at the head of $buff(j,i)$, by precondition.
2. b is in $B_i \cap B_j$, by Claim 1 and Lemma 1(d).
3. b is in $bottles(i)$, by Claim 1 and (C).
4. b is not in $bottles(j)$, by Claims 2 and 3 and (A).
5. $sat(b)$ is not in $buff(i,j)$, by Claims 2 and 3 and (A).
6. $sat(b)$ is not in $buff(j,i)$, by Claims 2 and 3 and (A).
7. Exactly one $req(b)$ is in $buff(j,i)$, by Claim 1 and (D-a).
8. $drink\text{-}region(j) = T$, by Claim 1 and (D-d).
9. b is in $req\text{-}bottles(j)$, by Claim 1 and (D-e).
10. $req(b)$ is not in $buff(i,j)$, by Claim 3 and (D-f).

(A) Only affects b. By Claims 4, 5 and 6 and code.

(B) Only affects (b,j). (a) by code. (b) by Claim 8. (c) by Claim 9.

(C) Only affects $buff(j,i)$. By Claim 7 and code, no $req(b)$ is in $buff(j,i)$, so vacuously true.

(D) Only affects $buff(i,j)$ and $buff(j,i)$. By Claims 7 and 10 and code, no $req(b)$ is in either $buff$, so vacuously true.

(E) Only affects $buff(i,j)$ if $sat(b)$ is added. (a) by Claim 5 and code. (b) by code. (c) by Claim 8 and code. (d) by Claim 9 and code.

(F) Only affects $buff(i,j)|b$. By code, b is removed from $bottles(i)$ if and only if (b,j) is removed from $deferred(i)$.

(G) Only affects b and j. By Claim 7 and code, no $req(b)$ is in $buff(j,i)$ in s_m. By Claim 4 and code, b is not in $bottles(j)$ in s_m. By Claim 5 and code, $sat(b)$ is in $buff(i,j)$ if and only if (b,j) is not in $deferred(i)$ in s_m.

(H) Only affects b and i. By Claim 3 and code.

Case 5: $a_m = deliver(dem(b), j, i)$. If b is not in $bottles(i)$ in s_{m-1}, then no relevant changes are made. Assume b is in $bottles(i)$ in s_{m-1}.

Claims about s_{m-1}:
1. b is in $bottles(i)$, by assumption.
2. $dem(b)$ is at the head of $buff(j,i)$, by precondition.
3. b is in $B_i \cap B_j$, by Claim 2 and Lemma 1(d).
4. b is not in $bottles(j)$, by Claims 1 and 3 and (A).
5. $sat(b)$ is not in $buff(i,j)$, by Claims 1 and 3 and (A).
6. $sat(b)$ is not in $buff(j,i)$, by Claims 1 and 3 and (A).
7. (b,j) is in $deferred(i)$, by Claims 1 and 2 and (F).
8. $drink\text{-}region(j) = T$, by Claim 7 and (B-b).
9. b is in $req\text{-}bottles(j)$, by Claim 7 and (B-c).
10. $req(b)$ is not in $buff(j,i)$, by Claim 7 and (D-c).
11. $req(b)$ is not in $buff(i,j)$, by Claim 1 and (D-f).

(A) Only affects b. By Claims 4, 5 and 6 and code.

(B) Only affects (b,j). By Claims 1, 8 and 9 and code.

(C) Only affects $buff(j,i)|b$. By Claim 10 and code, vacuously true.

(D) Only affects $buff(j,i)$ and $buff(i,j)$. By Claims 10 and 11, vacuously true.

(E) Suppose $sat(b)$ is added to $buff(i,j)$. (Nothing else is affected.) (a) By Claim 5 and code. (b) by code. (c) by Claim 8 and code. (d) by Claim 9 and code.

(F) Only affects $buff(i,j)|b$. By code, if b remains in $bottles(i)$, then (b,j) is in $deferred(i)$ in s_m.

(G) Only affects j and b. By Claim 10, no $req(b)$ is in $buff(j,i)$ in s_m. By Claim 4, b is not in $bottles(j)$ in s_m. By code, (b,j) is in $deferred(i)$ if and only if $sat(b)$ is not in $buff(i,j)$ in s_m.

(H) By Claim 1 and code.

Case 6: $a_m = C_i$.

Claims about s_{m-1}: If *drink-region*$(i) \neq T$ in s_{m-1}, then no relevant changes occur. Suppose otherwise. Only b in *req-bottles*(i) and not in *bottles*(i) is affected.
1. *drink-region*$(i) = T$, by assumption.
2. b is in *req-bottles*$(i) \cap B_j$, $j \neq i$, by assumption.
3. b is not in *bottles*(i), by assumption.
4. *req*(b) is in *buff*(i,j), or (b,i) is in *deferred*(i), or *sat*(b) is in *buff*(j,i), by Claims 1, 2 and 3 and (G).
5. If *sat*(b) is in *buff*(i,j), then no *sat*(b) is in *buff*(j,i) and b is not in *bottles*(j), by (A).
6. If *sat*(b) is in *buff*(i,j), then (b,i) is not in *deferred*(j), by Claim 5 and (B-a).
7. If *sat*(b) is in *buff*(i,j), then *req*(b) is in *buff*(i,j), by Claims 4, 5 and 6.
8. If *sat*(b) is in *buff*(i,j), then *req*(b) follows it in *buff*(i,j), by Claim 7 and (D-b).
9. If *buff*$(i,j)|b$ is empty and b is in *bottles*(j), then no *sat*(b) is in *buff*(j,i), by Claim 2 and (A).
10. If *buff*$(i,j)|b$ is empty and b is in *bottles*(j), then (b,i) is in *deferred*(j), by Claims 4 and 9.

(E-b) by Claim 8.

(F) by Claim 10.

Rest are not affected.

Case 7: $a_m = C_i(B)$. By Lemma 2, *sched*(e) is drinking-well-formed; thus in *sched*$(e)|B$-$TCER_i$, a_m is immediately preceded by $T_i(B)$. By Lemma 1(b), *req-bottles*$(i) = B$ in s_{m-1}.

Claims about s_{m-1}:
1. *drink-region*$(i) = T$, by precondition.
2. If b is in *req-bottles*(i), then b is in *bottles*(i), for all b, by precondition.
3. If b is in *req-bottles*(i), then b is not in *bottles*(j), where b is in B_j, $j \neq i$, for all b, by Claim 2 and (A).
4. If (b,i) is in *deferred*(j), then $i \neq j$ and b is in $B_i \cap B_j$, for all b and j, by Lemma 1(e).
5. (b,i) is not in *deferred*(j), for all b and j, by Claims 3 and 4 and (B-a) and (B-c).
6. *req*(b) is not in *buff*(i,j), for all b and j, by Claim 2 and (D-e) and (D-f).

7. If b is in *req-bottles*(i), then *sat*(b) is not in *buff*(j,i), where b is in B_j, $j \neq i$, for all b, by Claim 2 and (A).
8. If b is not in *req-bottles*(i), then *sat*(b) is not in *buff*(j,i), where b is in B_j, $j \neq i$, for all b, by (E-d).
9. *buff*$(j,i)|$*sat*(b) is empty for all b not in B_j, by Lemma 1(d).

(B-b) vacuously true by Claim 5.

(D-d) vacuously true by Claim 6.

(E-c) vacuously true by Claims 7, 8 and 9.

The rest are unaffected.

Case 8: $a_m = R_i(B)$. The only change is that *drink-region*(i) becomes R in s_m. By Lemma 2, *sched*(e) is drinking-well-formed; thus in *sched*$(e)|B\text{-}TCER_i$, a_m is immediately preceded by $E_i(B)$. By Lemma 1(a), *drink-region*$(i) = E$ in s_{m-1}. Thus (B-b), (D-d) and (E-c) are still true in s_m.

Case 9: $a_m = R_i$, T_i, or E_i. None of the changes affects any of the predicates. □

Lemma 5: *Let e be an execution of Drink($\mathcal{B}$) whose schedule is drinking-well-formed. Then in every state of e, the following are true, for all i.*

(A) If do-T(i) is true, then dine-region$(i) = R$.

(B) If do-E(i) is true, then dine-region$(i) = C$.

Proof: Let $e = s_0 a_1 s_1 \ldots a_m s_m \ldots$. We proceed by induction on m, which indexes the states of e.

(A) and (B) are obviously true of s_0, since it is a start state. Assuming (A) and (B) are true of s_{m-1}, we show they are true of s_m. We need only consider the following values for a_m.

Case 1: $a_m = T_i(B)$.

(A) If *dine-region*$(i) = R$ in s_{m-1}, then by code. If *dine-region*$(i) \neq R$ in s_{m-1}, then by induction hypothesis for (A), *do-*T_i is false in s_{m-1}; since by code it is still false in s_m, we are done.

(B) By the induction hypothesis, since there is no relevant change.

Case 2: $a_m = C_i$. First note that $a_1 \ldots a_{m-1} | F\text{-}TCER_i$ ends in T_i, by dining-well-formedness.

Claims about s_{m-1}*:*
1. *dine-region*$(i) = T$, by above note and Lemma 1(c).
2. *do-*$T(i)$ = false, by Claim 1 and (A).

(A) by Claim 2 and code, vacuous.

(B) by code.

Case 3: $a_m = R_i$. First note that $a_1 \ldots a_{m-1} | F\text{-}TCER_i$ ends in E_i by dining-well-formedness.

Claims about s_{m-1}*:*
1. *dine-region*$(i) = E$, by above note and Lemma 1(c).
2. *do-*$E(i)$ = false, by Claim 1 and (B).

(A) by code.

(B) by Claim 2 and code, vacuous.

Case 4: $a_m = C_i(B)$.

(A) and (B) by induction hypothesis and code.

Case 5: $a_m = T_i$.

(A) by code.

(B) By (A) and precondition, *drink-region*$(i) = R$ in s_{m-1}. By (B), *do-*$E(i)$ = false in s_{m-1}, and still in s_m.

Case 6: $a_m = E_i$.

(A) By (B) and precondition, *drink-region*$(i) = C$ in s_{m-1}. By (A), *do-*$T(i)$ = false in s_{m-1}, and still in s_m.

(B) by code. □

Lemma 12: *Suppose Dine($\mathcal{B}$) solves the dining philosophers problem. Let e be an execution of Drink($\mathcal{B}$) whose schedule is drinking-well-formed. The following predicates are true in every state of e, for any i, j and b.*

(A) If there is a current dem(b) message in buff(i,j), then
- *(a) drink-region(i) = T,*
- *(b) dine-region(i) = C,*
- *(c) b is in req-bottles(i), and*
- *(d) do-E(i) is false.*

(B) There is at most one current dem(b) message in buff(i,j).

(C) There is at most one non-current dem(b) message in buff(i,j).

Proof: Let $e = s_0 a_1 s_1 \ldots a_m s_m \ldots$. We proceed by induction on m, which indexes the states of e.

(A) through (C) are obviously true of s_0, since it is a start state. Assuming (A) through (C) are true of s_{m-1}, we show that they are true of s_m. We consider every possible value of a_m. By Lemma 6(a), *sched*(e) is dining-well-formed.

Case 1: $a_m = T_i(B)$. Only messages in buff(i,j), for all j, are affected.

Remark: By drinking-well-formedness, $a_1 \ldots a_{m-1}|B$-it TCER_i ends in $R_i(B')$ for some B', or is empty.

Claims about s_{m-1}:
1. drink-region(i) = R, by Remark and Lemma 1(a).
2. No current dem(b) is in buff(i,j) for any b and j, by Claim 1 and (A-a).
3. At most one non-current dem(b) is in buff(i,j) for any b and j, by (C).

Claims about s_m:
4. No current dem(b) is in buff(i,j) for any b and j, by Claim 2 and code.
5. At most one non-current dem(b) is in buff(i,j) for any b and j, by Claim 3 and code.

(A) By Claim 4, vacuously true for buff(i,j) for all j.

(B) By Claim 4 for buff(i,j) for all j.

(C) By Claim 5 for $buff(i,j)$ for all j.

Case 2: $a_m = E_i(B)$. Only $dem(b)$ messages in $buff(i,j)$ and $buff(j,i)$ are affected, where (b,j) is in $deferred(i)$ in s_{m-1}. Fix such a b and j.

Remark: By drinking-well-formedness, $a_1 \ldots a_{m-1}|B\text{-}TCER_i$ ends in $C_i(B)$.

Claims about s_{m-1}*:*

1. $drink\text{-}region(i) = C$, by Remark and Lemma 1(a).
2. No current $dem(b)$ is in $buff(i,j)$, by Claim 1 and (A-a).
3. At most one non-current $dem(b)$ is in $buff(i,j)$, by (C).
4. b is in $bottles(i)$, by choice of b and Lemma 3 (B-a).
5. If $dem(b)$ is in $buff(j,i)$, then $dem(b)$ is current, by Claim 4.
6. At most one $dem(b)$ is in $buff(j,i)$, by Claim 5 and (B).

Claims about s_m*:*

7. No current $dem(b)$ is in $buff(i,j)$, by Claim 2 and code.
8. At most one non-current $dem(b)$ is in $buff(i,j)$, by Claim 3 and code.
9. At most one $dem(b)$ is in $buff(j,i)$, by Claim 6 and code.

(A) By Claim 7 for $buff(i,j)$. No relevant change for $buff(j,i)$.

(B) By Claim 7 for $buff(i,j)$. By Claim 9 for $buff(j,i)$.

(C) By Claim 8 for $buff(i,j)$. By Claim 9 for $buff(j,i)$.

Case 3: $a_m = deliver(sat(b),j,i)$. The only messages affected are $dem(b)$ in $buff(i,j)$ or $buff(j,i)$.

Claims about s_{m-1}*:*

1. $sat(b)$ is at the head of $buff(j,i)$, by precondition.
2. If $dem(b)$ is in $buff(j,i)$, then it is current, by Claim 1.
3. At most one $dem(b)$ is in $buff(j,i)$, by Claim 2 and (B).
4. If $dem(b)$ is in $buff(i,j)$, then it is current, by Claim 1.
5. At most one $dem(b)$ is in $buff(i,j)$, by Claim 4 and (B).

Claims about s_m*:*

6. At most one $dem(b)$ is in $buff(j,i)$, by Claim 3 and code.

7. At most one $dem(b)$ is in $buff(i,j)$, by Claim 5 and code.

   (A) No relevant changes are made.

   (B) By Claims 6 and 7.

   (C) By Claims 6 and 7.

Case 4: $a_m = deliver(req(b), j, i,)$. If the request is deferred, there is no relevant change. Suppose the request is satisfied, i.e., $sat(b)$ is added to $buff(i,j)$. The only messages affected are $dem(b)$ in $buff(i,j)$ or $buff(j,i)$.

Claims about s_{m-1}*:*
1. $req(b)$ is at the head of $buff(j,i)$, by precondition.
2. b is in $bottles(i)$, by Claim 1 and Lemma 3 (C).
3. If $dem(b)$ is in $buff(j,i)$, then it is current, by Claim 2.
4. At most one $dem(b)$ is in $buff(j,i)$, by Claim 3 and (B).
5. b is in $B_i \cap B_j$, by Claim 1 and Lemma 1(d).
6. If $dem(b)$ is in $buff(i,j)$, then it is not current, by Claims 2 and 5 and Lemma 3 (A).
7. At most one $dem(b)$ is in $buff(i,j)$, by Claim 6 and (C).

Claims about s_m*:*
8. At most one $dem(b)$ is in $buff(j,i)$, by Claim 4 and code.
9. At most one $dem(b)$ is in $buff(i,j)$, by Claim 7 and code.

   (A) No relevant change.

   (B) By Claims 8 and 9.

   (C) By Claims 8 and 9.

Case 5: $a_m = deliver(dem(b), j, i)$. If b is not in $bottles(i)$ in s_{m-1}, then there is no relevant change. Suppose b is in $bottles(i)$ in s_{m-1}. The only messages affected are $dem(b)$ in $buff(i,j)$ or $buff(j,i)$.

Claims about s_{m-1}*:*
1. $dem(b)$ is at the head of $buff(j,i)$, by precondition.

2. b is in $bottles(i)$, by assumption.
3. If $dem(b)$ is in $buff(j,i)$, then it is current, by Claim 1.
4. There is exactly one $dem(b)$ in $buff(j,i)$, by Claims 1 and 3.
5. b is in $B_i \cap B_j$, by Claim 1 and Lemma 1(d).
6. If $dem(b)$ is in $buff(i,j)$, then it is non-current, by Claim 2 and Lemma 3 (A).
7. There is at most one $dem(b)$ in $buff(i,j)$, by Claim 6 and (C).

Claims about s_m:
8. There is no $dem(b)$ in $buff(j,i)$, by Claim 4 and code.
9. There is at most one $dem(b)$ in $buff(i,j)$, by Claim 7 and code.
10. If $dem(b)$ is in $buff(i,j)$, then it is non-current, by Claim 6 and code (i.e., $sat(b)$ is added to the end of $buff(i,j)$, if it is added at all).

(A) By Claims 8 and 10.

(B) By Claims 9 and 10.

(C) By Claims 9 and 10.

Case 6: $a_m = C_i$. First, suppose *drink-region*$(i) \neq T$ in s_{m-1}. Then by (A-d), no current $dem(b)$ message is in $buff(i,j)$, for any b and j, in s_{m-1}. Thus, setting *do-E*(i) to true in s_m does not falsify (A-d). There is no relevant change for the rest of the invariants.

Now suppose *drink-region*$(i) = T$ in s_{m-1}. We need only consider a $dem(b)$ added to some $buff(i,j)$ in s_m. Fix such a b and j.

Remark: By dining-well-formedness, $a_1 \ldots a_{m-1} | F\text{-}TCER_i$ ends in T_i.

Claims about s_{m-1}:
1. *dine-region*$(i) = T$, by Remark and Lemma 1(c).
2. If $dem(b)$ is in $buff(i,j)$, then it is non-current, by Claim 1 and (A-b).
3. At most one $dem(b)$ is in $buff(i,j)$, by Claim 2 and (C).
4. b is not in $bottles(i)$, by code and choice of b.
5. $sat(b)$ is in $buff(i,j)$, or b is in $bottles(j)$, or $sat(b)$ is in $buff(j,i)$, by Claim 4 and Lemma 3 (A).
6. *drink-region*$(i) = T$, by assumption.
7. b is in *req-bottles*(i), by choice of b.
8. *do-E*(i) is false, by Claim 1 and Lemma 5 (B).

Claims about s_m:

9. The *dem*(b) message added to *buff*(i,j) is current, by Claim 5 and code.

10. *drink-region*$(i) = T$, *dine-region*$(i) = C$, b is in *req-bottles*(i), and *do-E*(i) is false, by Claims 6, 7 and 8 and code.

11. One current *dem*(b) message is in *buff*(i,j), by Claims 2 and 9 and code.

(A) By Claim 10.

(B) By Claim 11.

(C) No relevant change.

Case 7: $a_m = R_i$. The only relevant change is to *dine-region*(i), affecting (A-b) for i. By dining-well-formedness, $a_1 \ldots a_{m-1} | F\text{-}TCER_i$ ends in E_i. By Lemma 1(c), *dine-region*$(i) = E$ in s_{m-1}, so by (A-b) there is no current *dem*(b) in *buff*(i,j), for any b and j in s_{m-1}. By code, the same is true in s_m, so (A-b) for i is vacuously true in s_m.

Case 8: $a_m = C_i(B)$. The only relevant change is to *do-E*(i), affecting (A-d) for i. By precondition (*req-bottles*(i) a subset of *bottles*(i)) and (A-c), there is no current *dem*(b) in *buff*(i,j), for any b and j. By code, the same is true in s_m, so (A-d) for i is vacuously true in s_m.

Case 9: $a_m = R_i(B)$. The only change is to *drink-region*(i), affecting (A-a) for i. By precondition, *drink-region*$(i) = E$ in s_{m-1}, so by (A-a), there is no current *dem*(b) in *buff*(i,j), for any b and j. By code, the same is true in s_m, so (A-a) for i is vacuously true in s_m.

Case 10: $a_m = T_i$. The only relevant change is to *dine-region*(i), affecting (A-b) for i. By precondition and Lemma 5 (A), *dine-region*$(i) = R$ in s_{m-1}, so by (A-b), there is no current *dem*(b) in *buff*(i,j), for any b and j. By code, the same is true in s_m, so (A-b) for i is vacuously true in s_m.

Case 11: $a_m = E_i$. The only changes are to *dine-region*(i) and *do-E*(i), affecting (A-b) and (A-d) for i. By precondition and (A-b), there is no current *dem*(b) in *buff*(i, j), for any b and j. By code, the same is true in s_m, so (A-b) and (A-d) for i are vacuously true in s_m. □