Stabilization-Preserving Atomicity Refinement (Full Version)

Mikhail Nesterenko[1] ⋆ and Anish Arora[2] ⋆⋆

[1] Mathematics and Computer Science,
Kent State University, Kent, OH 44242 USA,
mikhail@mcs.kent.edu.

[2] Department of Computer and Information Science,
Ohio State University
Columbus, OH 43210 USA,
anish@cis.ohio-state.edu

Abstract. Program refinements from an abstract to a concrete model empower designers to reason effectively in the abstract and architects to implement effectively in the concrete. For refinements to be useful, they must not only preserve functionality properties but also dependability properties. In this paper, we focus our attention on refinements that preserve the property of stabilization.

We distinguish between two types of stabilization-preserving refinements — atomicity refinement and semantics refinement — and study the former. Specifically, we present a stabilization-preserving atomicity refinement from a model where a process can atomically access the state of all its neighbors and update its own state, to a model where a process can only atomically access the state of any one of its neighbors or atomically update its own state. (Of course, correctness properties, including termination and fairness, are also preserved.)

Our refinement is based on a low-atomicity, bounded-space, stabilizing solution to the dining philosophers problem. It is readily extended to: (a) solve stabilization-preserving semantics refinement, (b) solve the drinking philosophers problem, and (c) allow further refinement into a message-passing model.

1 Introduction

Concurrent programming involves reasoning about the interleaving of the execution of multiple processes running simultaneously. On one hand, if the grain of atomic (indivisible) actions of a concurrent program is assumed to be coarse, the number of possible interleavings is kept small and the program design is made

⋆ This research was supported in part by a grant from the Division of Research and Graduate Studies, Kent State University

⋆⋆ This research was supported in part by an Ameritech Faculty Fellowship and grants from the National Science Foundation, the National Security Agency, and Microsoft Research.

simple. On the other hand, if the program is to be efficiently implemented, its atomic actions must be fine-grain. This motivates the need for refinements from *high-atomicity* programs to *low-atomicity* programs.

Atomicity refinement must preserve the correctness of the high-atomicity program. In other words, the safety (e.g., invariants) and the liveness (e.g., termination and fairness) properties of that program must also hold in the corresponding low-atomicity program. But it is also important to preserve the non-functional properties of the high-atomicity program. In this paper, we concentrate on refinements that, in addition to preserving functionality, preserve the property of stabilization.

Informally speaking, stabilization of a program with respect to a set of legitimate states implies that upon starting from an arbitrary state, every execution of the program eventually reaches a legitimate state and thereafter remains in legitimate states. It follows that a stabilizing program does not necessarily have to be initialized and is able to recover from transient failures.

To be fair, our notion of stabilization-preservation atomicity refinement should be distinguished from what we call stabilization-preservation semantics refinement:

- *Atomicity refinement.* In this case, the atomicity of program actions is refined from high to low, but the semantics of concurrency in program execution is not. For instance, both the high- and low-atomicity programs may be executed in *interleaving* semantics, where only one atomic operation may be executed at a time. Alternatively, both programs may be executed in *power-set* semantics, where any number of processes may each execute an atomic action at a time, or both in *partial-order* semantics, etc.
- *Semantics refinement.* In this case, the semantics of concurrency in program execution is refined, but the program atomicity is not. For instance, a program in interleaving semantics may be refined to execute (with identical actions) in power-set semantics [2]. The program is more easily reasoned about in the former semantics, but more easily implemented in the latter.

An elegant solution for a semantics refinement problem has been proposed by Gouda and Haddix [12]. Their solution does not however achieve atomicity refinement. In this paper, by way of contrast, we focus on an atomicity refinement problem. But, as an aside, we demonstrate that our solution is applicable for semantics refinement as well.

Specifically, we consider atomicity refinement from a model where a process can atomically access the state of all its neighbors and update its own state, to a model where a process can only atomically access the state of any one of its neighbors or atomically update its own state. (We also address further refinement to a message-passing model.) In all models, concurrent execution of actions of processes is in interleaving semantics.

As can be expected, the straightforward division of high-atomicity actions into a sequence of low-atomicity actions does not suffice because each sequence may not execute in isolation. A simple strategy for refinement, therefore, is to execute each sequence in a mutually exclusive manner. Of course, the mechanism for achieving mutual exclusion has to be (i) itself stabilizing, in order for the refinement to be stabilization-preserving, (ii) in low-atomicity, since the refined program is in low atomicity, and (iii) bounded space, to be implemented reasonably.

This simple strategy unfortunately suffers from loss of concurrency, since no two processes can execute sequences concurrently even if these sequences operate on completely disjoint state spaces. We are therefore led to solving the problem of dining philosophers, which requires mutual exclusion only between "neighboring" processes, and thus allows more concurrency.

Although there are a number of stabilizing mutual exclusion programs in the literature [4, 5, 11, 15, 18], none of them is easily generalized to solve dining philosophers. Mizuno and Nesterenko [17] consider dining philosophers in order to solve a problem that has a flavor of atomicity refinement, but their solution uses infinite variables. It is well-known that bounding the state of stabilizing programs is often challenging [3]. This motivates a new solution to the dining philosopher's problem which satisfies the requirements (i)-(iii) above.

Other notable characteristics of our refinement include: It is *sound* and *complete*; i.e. every computation of the low-atomicity program corresponds to a unique computation of the high-atomicity program, and for every computation of the high-atomicity program there is a computation of the low-atomicity program that corresponds to it. It is *fixpoint-preserving*; i.e., terminating computations of the high-atomicity program correspond only to terminating computations of the low atomicity program. It is *fairness-preserving*; i.e., weak-fairness of action execution is preserved, which intuitively implies that the refinement includes a stabilizing, low-atomicity weakly-fair scheduler.

We describe further refinement into a message-passing model. An (unbounded space) transformation from high-atomicity model into message-passing model is presented in [16]. Our solution has bounded space complexity.

The rest of the paper is organized as follows. We define the model, syntax, and semantics of the programs we use in Section 2. We then present a low-atomicity bounded-space dining philosophers program and prove its correctness and stabilization properties, in Section 3. Next, in Section 4, we demonstrate how a high-atomicity program is refined using our dining philosophers program, and show the relationship between the refined program and the original high-atomicity program in terms of soundness, completeness, and fixpoint- and fairness- preservation. We summarize our contribution and discuss extensions of our work in Section 5.

2 Model, Syntax, and Semantics

Model. A *program* consists of a set of processes and a binary reflexive symmetric relation N between them. The processes are assumed to have unique identifiers

1 through n. Processes P_i and P_j are called *neighbor processes* iff $(P_i, P_j) \in N$. Each process in the system consists of a set of variables, set of parameters, and a set of guarded commands (GC).

Syntax of high-atomicity programs. The syntax of a process P_i has the form:

> **process** P_i
> **par** ⟨declarations⟩
> **var** ⟨declarations⟩
>
> ∗[
>   ⟨guarded command⟩ [] ... [] ⟨guarded command⟩
>]

Declarations is a comma-separated list of items, each of the form:

$$\langle list\ of\ names \rangle : \langle domain \rangle$$

A variable can be updated (written to) only by the process that contains the variable. A variable can be read by the process that contains the variable or by a neighbor process. We refer to a variable v that belongs to process P_i as v_i.

A parameter is used to define a set of variables and a set of guarded commands as one parameterized variable and guarded command respectively. For example, let a process P_i have parameter j ranging over values 2,5, and 9; then a parameterized variable $x.j$ defines a set of variables $\{x.j \mid j \in \{2, 5, 9\}\}$ and a parameterized guarded command $GC.j$ defines the set of GCs:

$$GC.(j := 2)\ []\ GC.(j := 5)\ []\ GC.(j := 9)$$

A guarded command has the syntax:

$$\langle guard \rangle \longrightarrow \langle command \rangle$$

A guard is a boolean expression containing local and neighbor variables. A command is a finite comma separated sequence of assignment statements updating local variables and branching statements. An assignment statement can be simple or quantified. A quantified assignment statement has the form:

$$(\| \langle range \rangle : \langle assignments \rangle)$$

quantification is a bound variable and the values it contains. Assignments is a comma separated list of assignment statements containing the bound variable. Similar to parameterized GC, a quantified statement represents a set of assignment statements where each assignment statement is obtained by replacing every occurrence of the bound variable in the assignments by its instance from the specified range.

Syntax of low-atomicity programs. The syntax for the low-atomicity program is the same as for the high atomicity program with the following restrictions. The variable declaration section of a process has the following syntax:

var
  private ⟨declarations⟩
  public ⟨declarations⟩

A variable declared as private can be read only by the process that contains this variable. A public variable can also be read by a neighbor processes. A guarded command can be either *synch* or *update*. A synch GC mentions the public variables of one neighbor process and local private variables only. An update GC mentions local private and public variables.

Let v_i be a private variable of P_i and v_j a public variable of P_j. We say that v_i is an *image* of v_j if there is a synch guard of process P_i that is enabled when $v_i \neq v_j$ and which assigns $v_i := v_j$ and v_i is not updated otherwise. The variable which value is copied to the image variable is called a *source* of the image.

Semantics. The semantics of high- and low-atomicity programs is the same (cf. [1]). An assignment of values to variables of all processes in the concurrent program is a *state* of this program. A GC whose guard is **true** at some state of the program is *enabled* at this state. A *computation* is a maximal fair sequence of steps such that for each state s_i the state s_{i+1} is obtained by executing the command of some GC that is enabled at s_i. The maximality of a computation means that no computation can be a proper prefix of another computation and the set of all computations is suffix-closed. That is a computation either terminates in a state where none of the GCs are enabled or the computation is infinite. The fairness of a computation means that no GC can be enabled in infinitely many consequent states of the computation. A boolean variable is *set* in some state s if the value of this variable is **true** in s, otherwise the variable is *cleared*.

A *state predicate* (or just predicate) is a boolean expression on the state of a program. A state *conforms* to some predicate if this predicate has value **true** at this state. Otherwise, the state *violates* the predicate. By this definition every state conforms to predicate **true** and none conforms to **false**.

Let $\mathcal{P}$ be a program and R and S be state predicates on the states of $\mathcal{P}$. R is *closed* if every state of the computation of $\mathcal{P}$ that starts in a state conforming R also conforms to R. R *converges* to S in $\mathcal{P}$ if R is closed in $\mathcal{P}$, S is closed in $\mathcal{P}$, and any computation starting from a state conforming to R contains a state conforming to S. If **true** converges to R, we say that R just converges. $\mathcal{P}$ stabilizes to R iff **true** converges to R in $\mathcal{P}$. In the rest of the paper we omit the name of the program whenever it is clear from the context.

3 Dining Philosophers Program

3.1 Description

process P_i
par $j : (P_i, P_j) \in N$
var
  public
    $ready_i$: boolean,
    $a_i.j, c_i.j$: (0..3)
  private
    $\texttt{request}_i$: boolean,
    $r_i.j, y_i, j$: boolean,
    $b_i.j, d_i.j$: (0..3)

$*[$
(dp1) $\texttt{request}_i \wedge \neg ready_i \wedge (\forall k : a_i.k = d_i.k) \wedge (\forall k > i : \neg y_i.k) \longrightarrow$
    $ready_i :=$ **true**,
    $(\| k > i : y_i.k := r_i.k,\ a_i.k := (a_i.k + 1)$ **mod** $4)$
  $[]$
(dp2) $ready_i \wedge (\forall k : a_i.k = d_i.k) \wedge (\forall k < i : \neg r_i.k) \longrightarrow$
    / * critical section * /
    $ready_i :=$ **false**,
    $(\| k < i : a_i.k := (a_i.k + 1)$ **mod** $4)$
  $[]$
(dp3) $c_i.j \neq b_i.j \longrightarrow$
    $c_i.j := b_i.j$
  $[]$
(dp4) $r_i.j \neq ready_j \vee (b_i.j \neq a_j.i) \vee (d_i.j \neq c_j.i) \vee (j > i \wedge \neg ready_j \wedge y_i.j) \longrightarrow$
    $r_i.j := ready_j$,
    $b_i.j := a_j.i$,
    $d_i.j := c_j.i$,
    if $j > i \wedge \neg ready_j \wedge y_i.j$ **then** $y_i.j :=$ **false** **fi**
$]$

Fig. 1. Dining philosophers process

The dining philosophers problem was first stated in [8]. Any process in the system can request the access to a certain portion of code called *critical section*(CS). The objective of the algorithm is to ensure that the following two properties hold:

safety no two neighbor processes have guarded commands that execute CS enabled in one state;
liveness a process requesting to execute CS is eventually allowed to do so.

This section describes a program $\mathcal{DP}$ that solves the dining philosophers problem. Every process P_i of $\mathcal{DP}$ is shown in Figure 1. To refer to a guarded command executed by some process we attach the process identifier to the name of the guarded command shown in Figure 1. For example, guarded command $dp1_i$ sets variable $ready_i$. We sometimes use GCs identifiers in state predicates. For example, $dp1_i$ used in a predicate means that the guard of this GC is enabled.

Every P_i has the following variables:

- $\mathtt{request}_i$ - abstracts the reaction of the environment. It is a read-only variable which is used in program composition in later sections. P_i *wants to enter* its CS if $\mathtt{request}_i$ is set.
- $ready_i$ - indicates if P_i tries to execute its CS. P_i is *in CS contention* if $ready_i$ is set.
- $r_i.j$ - records whether P_j is in CS contention, it is an image of $ready_j$.
- $y_i.j$ - records if P_j requests CS and needs to be allowed to access it before P_i can request its own CS again. It is maintained for each P_j such that $j > i$; it is called the *yield variable*.
- $a_i.j, b_i.j, c_i.j, d_i.j$ - used for synchronization between neighbor processes; they are called *handshake variables*.

The basic idea of the program is: among the neighbor processes in CS contention the one with the lowest identifier is allowed to proceed. To ensure fairness, when a process joins CS contention it records the neighbors in CS contention with ids greater than its own; after the process exits its CS it is not allowed to request CS again until the recorded neighbors enter their CS.

Let us consider neighbor processes P_i and P_j and the following sequence of handshake variables $H_{ij} = \langle a_i.j,\ b_j.i,\ c_j.i,\ d_i.j\rangle$. We say that $a_i.j$ has a token if $a_i.j$ is equal to $d_i.j$. We say that any of the other variables has a token if it is *not* equal to the variable preceding it in H_{ij}. $\mathcal{DP}$ is constructed so that H_{ij} forms a ring similar to the a ring used in K-state stabilizing protocol described in [9].

Every P_i has the following GCs:

$dp1_i$ - update GC. When P_i wants to enter CS, it is not in CS contention, for every neighbor P_j, $a_i.j$ has a token, and yield variables for processes with identifiers greater than i are not set; then P_i sets $ready_i$ joining CS contention; P_i also sets yield variables for processes who are in CS contention and increments $a_i.j$ for every P_j with identifier greater than i passing the token in the handshake sequence.

Note, that when $a_i.j$ collects the token again P_j is informed of P_i's joining CS contention, that is $r_j.i$ is set. This ensures safety of the program.

$dp2_i$ - update GC. When P_i is in CS contention, every $a_i.j$ has the token and processes with smaller identifiers are not in CS contention then it is safe for P_i to execute its CS. Note, that critical section is put in comments in Figure 1 since no actual CS is executed. P_i clears $ready_i$ and increments $a_i.j$ for every P_j with identifier less than i, again passing the tokens.

Note, that when the tokens are collected every P_j is informed that P_i exited CS and yield variable $y_j.i$ is cleared. This ensures liveness of the program: P_i cannot repeatedly enter CS while P_j is stuck with $y_j.i$ set.

$dp3_i$ - update GC. It sets $c_i.j$ equal to $b_i.j$ thus passing the token from c to d.

$dp4_i$ - synch GC. It passes the tokens from $b_i.j$ to $c_i.j$ and from $d_i.j$ to $a_i.j$. $dp4_i$ also copies the value of $ready_j$ to it's image $r_i.j$ and clears the yield variable $y_i.j$ when P_j is not in CS contention.

3.2 Proof of Correctness of $\mathcal{DP}$

Stabilization

Let P_u and P_v be any two neighbor processes.

Proposition 1. $\mathcal{DP}$ stabilizes to the following predicate:

$$\text{there can be one and only one token in } H_{uv} \qquad (R_1)$$

This proposition is proven in the Appendix A.

Lemma 1. $\mathcal{DP}$ stabilizes to the following predicates:

$$((u < v) \wedge (a_u.v = b_v.u) \wedge ready_u) \Rightarrow r_v.u \qquad (R_2)$$

$$((u > v) \wedge (a_u.v = b_v.u) \wedge \neg ready_u) \Rightarrow \neg r_v.u \qquad (R_3)$$

Proof: By Proposition 1, $\mathcal{DP}$ stabilizes to R_1. To prove the lemma we need to demonstrate that R_2 and R_3 are closed when R_1 holds and that R_1 converges to R_2 and R_3. We prove convergence and closure for R_2 only. The stabilization of R_3 can be proven similarly.

We show the closure first. Out of the eight GCs of processes P_u and P_v only $dp1_u$, $dp2_u$ and $dp4_v$ affect the variables of the predicate. $dp1_u$ is executed only when $a_u.v = d_u.v$. That is when variable $a_u.v$ has the token. Since R_1 holds there can be no other tokens in the handshake variables. Thus, $a_u.v = b_v.u$ when $dp1_u$ is executed. Therefore $dp1_u$ sets the antecedent of our predicate to **false** by clearing $ready_u$. GC $dp2_u$ also sets the antecedent of our predicate to **false**. If $dp4_v$ sets the antecedent to **true** (by setting $a_u.v = b_v.u$), then the consequent is also set to **true** (since $r_v.u = ready_u$ after the execution of $dp4_v$).

To demonstrate convergence we note that when R_2 does not hold $dp4_v$ is enabled. When $dp4_v$ is executed the predicate holds. □

Lemma 2. $\mathcal{DP}$ stabilizes to the following predicate:

$$((u > v) \wedge (a_u.v = b_v.u) \wedge \neg ready_u) \Rightarrow \neg y_v.u \qquad (R_4)$$

We show closure first. The guarded commands that affect our predicate are: $dp1_u$, $dp2_u$, $dp1_v$ and $dp4_v$. $dp1_u$ and $dp2_u$ set the antecedent to **false** and therefore do not violate the predicate. $dp1_v$ affects only the consequent of our predicate. By R_3 when the antecedent of our predicate is **true**, then $r_v.u =$ **false**. Therefore, when the antecedent is **true** and $dp1_v$ is executed, variable $y_v.u$ remains **false** and our predicate is not violated.

$dp4_v$ can set the consequent of our predicate to **true** only. Also, if $dp4_v$ sets the antecedent of our predicate to **true** (by setting $a_u.v = b_v.u$ while $ready_u =$ **false**) the consequent of our predicate must also be **true** after the execution of $dp4_v$ (if $ready_u$ is cleared before the execution of $dp4_v$, $y_v.u$ is set to **false** by $dp4_v$).

Similar to Lemma 1 we demonstrate convergence by pointing out that when our predicate does not hold $dp4_v$ is enabled. When $dp4_v$ is executed the predicate holds. □

We now define a predicate I_{DP} (which stands for invariant of $\mathcal{DP}$) such that every computation of $\mathcal{DP}$ that starts at a state conforming to I_{DP} satisfies safety and liveness. I_{DP} is: for every pair of neighbor processes $R_1 \wedge R_2 \wedge R_3 \wedge R_4$. In other words in every state conforming to I_{DP}, every pair of neighbor processes conforms to every predicate from the above list.

Theorem 1. $\mathcal{DP}$ stabilizes to I_{DP}.

Thus every execution of $\mathcal{DP}$ eventually reaches a state conforming to I_{DP}. In the next two subsections we show that every computation that starts from a state conforming to I_{DP} satisfies safety and liveness properties.

Safety

Theorem 2 (Safety). In a state conforming to I_{DP}, no two neighbor processes have their guarded commands that execute critical section enabled.

Proof: Let us assume that P_u and P_v are neighbors and $u < v$. If $dp2_u$ is enabled then $a_u.v = d_u.v$. By R_1 this means that $a_u.v = b_v.u$. If $dp2_u$ is enabled then $ready_u$ is set. Therefore (byR_2) $r_v.u$ is also set. When $r_v.u$ is set $dp2_v$ cannot be enabled. □

Liveness

For a process P_u and its neighbor P_v the value of the variable $a_u.v$ is changed only when all a variables of process P_u have their tokens. The following observation can be made on the basis of Proposition 1.

Proposition 2. All a variables of a process eventually get the tokens. That is a state conforming to: $\exists v : (P_v, P_u) \in N : a_u.v \neq d_u.v$ is eventually followed by a state where: $\forall v : (P_v, P_u) \in N : a_u.v = d_u.v$

Lemma 3. If a process P_u is in CS contention it is eventually allowed to execute its CS.

Proof: To prove the lemma we need to show that for any P_u, if $ready_u$ is set then $dp2_u$ eventually gets enabled and stays enabled until executed.

The proof is by induction on the process identifiers. Suppose a process P_1 has $ready_1$ set in some state of a computation. By Proposition 2 all tokens are eventually collected at a_us variables and $dp2_1$ gets enabled. When $ready_1$ is set the only command that can manipulate the tokens is $dp2_1$. Therefore, a_us do not give up the tokens unless $dp2_1$ is executed and $dp2_1$ stays enabled until executed. Thus, the lemma holds for P_1.

Suppose now that the lemma holds for processes with identifiers smaller than u and P_u has $ready_u$ set at some state of a computation. Again, by Proposition 2, for any neighbor P_v, $a_u.v$ gets the token. To demonstrate that $dp2_u$ eventually becomes and stays enabled until it is executed we need to show that for any neighbor P_v such that $v < u$, $r_u.v$ is eventually cleared and never set until $dp2_u$ is executed. There can be two cases:

- $ready_v$ is set in infinitely many states of the computation. By assumption the lemma holds for P_v. Therefore, every such state is eventually followed by a state where $ready_v$ is cleared. After such a state $ready_v$ is set again. Thus $dp1_v$ and $dp2_v$ are executed infinitely many times during the computation. If $ready_u$ is set, eventually $r_v.u$ is set as well. When $dp1_v$ is executed in a state where $r_v.u$ set, this command sets $y_v.u$. $y_v.u$ is not cleared while $ready_u$ (and subsequently $r_v.u$) is set. When $y_v.u$ is set $dp1_v$ cannot be executed and $ready_v$ remains cleared while $ready_u$ is set. If $ready_v$ is cleared, eventually $r_u.v$ is cleared as well. Thus, $r_u.v$ stays cleared until $dp2_u$ is executed.
- $ready_v$ is set in only finitely many states of the computation. In this case there is a suffix of the computation where $ready_v$ is not set in any of the states. Thus eventually $r_u.v$ is cleared and remains cleared for the rest of the computation.

Thus $dp2_u$ becomes enabled and it stays enabled until executed. □

Lemma 4. If a process P_u wants to enter its CS it eventually joins CS contention.

Proof: To prove the lemma we need to show that if $\texttt{request}_u$ is enabled then $dp1_u$ (that sets $ready_u$) is eventually executed. Let us assume that $\texttt{request}_u$ holds unless $ready_u$. By Proposition 2 for any P_u's neighbor P_v, $a_u.v$ eventually gets the token. When $ready_u$ is cleared $a_u.v$ never gives up the token unless $dp1_u$ is executed. Therefore, to demonstrate that the $dp1_u$ gets enabled we need to show that for any neighbor P_v such that $v > u$, $y_u.v$ eventually gets cleared. Note that $y_u.v$ is set only when $dp1_u$ is executed.

There can be only two cases:

- $ready_v$ is set in infinitely many states. By Lemma 3 this implies that $ready_v$ is also cleared in infinitely many states as well. Therefore, $dp1_v$ is executed infinitely many times. By Predicate R_4, $dp1_v$ can be executed only in a state where $y_u.v$ is cleared.

– $ready_v$ is set in only finitely many states. In this case there is a suffix of the execution where $ready_v$ is never set. If $y_u.v$ is set then $dp4_u$ is enabled. When $dp4_u$ is executed $y_u.v$ is cleared.

□

The following theorem unifies Lemmas 3 and 4.

Theorem 3 (Liveness). If I_{DP} holds, a process that wants to enter its CS is eventually allowed to do so.

4 The Refinement

4.1 High-Atomicity Program

process P_i
var x_i

$*[$

(h1) $\quad g_i(x_i,\ \langle x_k \mid (P_i, P_k) \in N \rangle) \longrightarrow x_i := f_i(x_i,\ \langle x_k \mid (P_i, P_k) \in N \rangle)$

$]$

Fig. 2. High-atomicity process

Each process P_i of high-atomicity program ($\mathcal{H}$) is shown in Figure 2. To simplify the presentation we assume that P_i contains only one GC. We provide the generalization to multiple GCs later in the section. Each P_i of $\mathcal{H}$ contains a variable x_i which is updated by $h1_i$. The type of x_i is arbitrary. The guard of this GC is a predicate g_i that depends on the values of x_i and variables of neighbors processes. The command of $h1_i$ assigns a new value to x_i. The value is supplied by a function f_i which again depends on the previous value of x_i as well as on the values of the variables of the neighbors. Recall, that unlike low-atomicity program such as $\mathcal{DP}$, a GC of $\mathcal{H}$ can read any variable of the neighbor process and update its own variable in one GC.

4.2 Composing $\mathcal{DP}$ and $\mathcal{H}$

To produce the refinement $\mathcal{C}$ of $\mathcal{H}$ we *superpose* additional commands on the GCs of $\mathcal{DP}$ and demonstrate that $\mathcal{C}$ is equivalent to $\mathcal{H}$. Superposition is a type of program composition that preserves safety and liveness properties of the underlying program ($\mathcal{DP}$). $\mathcal{C}$ consists of $\mathcal{DP}$, superposition variables, superposition commands and superposition GCs. The superposition variables are disjoint from variables of $\mathcal{DP}$. Each superposition command has the following form:

$$\langle GC\ of\ \mathcal{DP} \rangle \parallel \langle command \rangle$$

The type of combined GC (synch or update) is the same as the type of the GC of $\mathcal{DP}$. The superposition commands and GCs can read but cannot update the variables of $\mathcal{DP}$. They can update the superposed variables. Operationally speaking a superposed command executes in parallel (synchronously) with the GC of $\mathcal{DP}$ it is based upon, and a superposed GC executes independently (asynchronously) of the other GCs. Refer to [7] for more details on superposition. Superposition preserves liveness and safety properties of the underlying program. In particular, if R is stabilizing for $\mathcal{DP}$ it is also stabilizing for $\mathcal{C}$. Thus, I_{DP} is also an invariant of $\mathcal{C}$.

process P_i
par $j : (P_i, P_j) \in N$
var
  public x_i
  private
    $x_i.j$,
    $\mathtt{request}_i$: boolean
$*[$
(c1) $dp1$
$[]$
(c2) $dp2 \parallel \left(\begin{array}{l} \textbf{if}\ \ g_i(x_i, \langle x_i.k \mid (P_i, P_k) \in N \rangle)\ \ \textbf{then}\ \ x_i := f_i(x_i,\ \langle x_i.k \mid (P_i, P_k) \in N \rangle)\ \ \textbf{fi} \\ \mathtt{request}_i := \textbf{false} \end{array} \right)$
$[]$
(c3) $dp3$
$[]$
(c4) $dp4 \parallel (\ \textbf{if}\ \ x_i.j \neq x.j\ \textbf{then}\ \ x_i.j := x.j,\ \mathtt{request}_i := \textbf{true}\ \ \textbf{fi}\ )$
$[]$
(c5) $x_i.j \neq x.j \quad \longrightarrow \quad x_i.j := x.j,\ \mathtt{request}_i := \textbf{true}$
$[]$
(c6) $g_i(x_i, \langle x_i.k \mid (P_i, P_k) \in N \rangle) \wedge \neg \mathtt{request}_i \quad \longrightarrow$
    $\mathtt{request}_i := \textbf{true}$
$]$

Fig. 3. Refined process

Each process P_i of the composed program ($\mathcal{C}$) is shown in Figure 3. For brevity, we only list the superposed variables in the variable declaration section. Besides the x_i we add $x_i.j$ which is an image of x_j for every neighbor P_j. Superposed variable $\mathtt{request}_i$ is read by $\mathcal{DP}$. Yet it does not violate the liveness and safety properties of $\mathcal{DP}$ since no assumptions about this variable was used when these properties were proven.

The GCs of $\mathcal{DP}$ are shown in abbreviated form. We superpose the execution of $h1$ on $dp2$. Note that $c2$ is an update GC. Therefore, the superposed command cannot read the value of x_j of a neighbor P_j directly as $h1$ does. The image $x_i.j$ is used instead. We superpose copying of the value of x_j into $x_i.j$ on $dp4$. Thus, the images of neighbor variables of $\mathcal{H}$ are equal to the sources when $h1$ is executed by $\mathcal{C}$. We add a superposition GC $c5$ that copies the value of x_j into $x_i.j$. This GC ensures that no deadlock occurs when an image is not equal to its source. $\mathtt{request}_i$ is set when one of the images of the superposed variables is found to be different from the sources or when the guard of $h1$ evaluates to **true** ($c6$). $\mathtt{request}_i$ is cleared after $h1$ is executed.

So far we assumed that $\mathcal{H}$ has only one GC. The refined program can be extended to multiple GCs. In this case, $c2$ has to select one of the enabled GCs of $\mathcal{H}$ and execute it. $c6$ has to be enabled when at least one of the GCs of $\mathcal{H}$ is enabled. We prove the correctness of $\mathcal{C}$ assuming that $\mathcal{H}$ has only one GC. In a straightforward manner, our argument can be extended to encompass multiple GCs.

4.3 Correctness of the Refinement

Throughout this section we assume that P_u and P_v are neighbor processes.

Lemma 5. $\mathcal{C}$ stabilizes to the following predicates:

$$((u < v) \wedge (a_u.v = d_u.v) \wedge ready_u) \Rightarrow (x_u.v = x_v) \qquad (R_5)$$

$$((u > v) \wedge \neg r_u.v) \Rightarrow (x_u.v = x_v) \qquad (R_6)$$

Proof: To demonstrate the stabilization of these predicates we show that they are closed under the assumption that I_{DP} holds and that they converge.

We show the closure of R_5 first. Of the twelve guarded commands of P_u and P_v, the following GCs affect R_5: $c1_u$, $c2_u$, $c4_u$, $c5_u$, and $c2_v$. $c1_u$ and $c2_u$ set the antecedent to **false**; $c4_u$ and $c5_u$ set the consequent to **true**. If $c2_v$ holds at certain state then $\neg r_v.u$. By R_2 this implies that at this state the antecedent of R_5 is false. Therefore, the execution of $c2_v$ does not violate R_5.

To show the closure of R_6 we note that only $c4_u$, $c5_u$, and $c2_v$ affect R_6. Again both $c4_u$ and $c5_u$ set the consequent to **true**. Holding of $c2_v$ implies that the antecedent of R_6 is false by R_2.

To demonstrate convergence of both predicates we observe that when either of them does not hold $c5_u$ is enabled and it remains enabled until executed. After $c5_u$ is executed the predicates hold. □

The following corollary can be deduced from the lemma.

Corollary 1. If I_{DP} holds, $c2$ is executed only when the images of the neighbor variables are equal to the sources. That is:

$$\forall (P_u, P_v) \in N : c2_u \Rightarrow (x_u.v = x.v)$$

Lemma 6. $\mathcal{C}$ stabilizes to the following predicates:

$$((u < v) \wedge (a_u.v = b_v.u) \wedge \neg ready_u) \Rightarrow (x_u = x_v.u) \qquad (R_7)$$
$$((u > v) \wedge (a_u.v = b_v.u) \wedge ready_u) \Rightarrow (x_u = x_v.u) \qquad (R_8)$$

Proof: We prove the stabilization of R_7. The stabilization of R_8 can be shown likewise. Similarly to Lemma 5 we demonstrate the stabilization of the R_7 by showing that it is closed under the assumption that I_{DP} holds and that it converges.

We show the closure first. Of the twelve guarded commands of P_u and P_v, the following GCs affect R_7: $c1_u$, $c2_u$, $c4_v$, $c5_v$. $c1_u$ and $c2_u$ set the antecedent to **false**; $c4_u$ and $c5_u$ set the consequent to **true**. Therefore, R_7 is not violated.

To demonstrate convergence we observe that when R_7 does not hold $c5_u$ is enabled and it remains enabled until executed. After $c5_u$ is executed the predicate holds. □

We define the invariant for $\mathcal{C}$ (denoted I_C) to be the conjunction of I_{DP}, R_5, R_6, R_7, and R_8. On the basis of Theorem 1, Lemma 5, and Lemma **??** we can conclude:

Theorem 4. $\mathcal{C}$ stabilizes to I_C.

Recall, that a global state is by definition an assignment of values to all the variables of a concurrent program. If a program is composed of several component programs, then a *component projection* of a global state s is a part of s consisting of the assignment of values to the variables used only in one of the components of the program. *Stuttering* is a sequence of identical states. A component projection of a computation is a sequence of corresponding component projections of all the states of the computation with finite stuttering eliminated. Note, that projection of a computation does not eliminate an infinite sequence of identical states. When we discuss a projection (of a computation or a state) of $\mathcal{C}$ onto $\mathcal{H}$ we omit $\mathcal{H}$ and just say a projection of $\mathcal{C}$. A *fixpoint* is a state where none of the GCs of the program are enabled. Thus, a computation either ends in a fixpoint or it is infinite.

Proposition 3. Let s be a fixpoint of $\mathcal{C}$. The following is true in s:

- $a_u.v = b_v.u = c_v.u = d_u.v$
- $r_u.v = ready_v$
- $ready_u$ is cleared;
- if $u < v$ then $y_u.v$ is cleared;
- $x_u.v = x_v$

Theorem 5 (Fixpoint preservation). When I_C holds, a projection of a fixpoint of $\mathcal{C}$ is a fixpoint of $\mathcal{H}$; and if a computation of $\mathcal{C}$ starts from a state which projection is a fixpoint of $\mathcal{H}$ then this computation ends in a fixpoint.

Proof: Let s be a fixpoint of $\mathcal{C}$. By Proposition 3, at state s, $ready_u$ and $y_u.v$ (if $u < v$) are cleared, and $a_u.v$ has the token. Since s is a fixpoint, $c1_u$ is not enabled, therefore, $\texttt{request}_u$ is cleared at s.

Since $c6_u$ is not enabled and $\texttt{request}_u$ is cleared, $h1_u$ is not enabled either. By Proposition 3, $x_u.v$ is equal to x_v at s. Therefore the projection of s does not have $h1$ enabled and this projection is a fixpoint.

We now show that the computation that starts at a state s_1 such that the projection of s_1 is a fixpoint this computation ends in a fixpoint. By Corollary 1 if I_C holds, $x_u.v = x_v$ when $c2_u$ is executed. Thus, if the projection of the initial state of the computation is a fixpoint, $h1_u$ is not executed during this computation. Therefore the projection of every state of this computation is a fixpoint of $\mathcal{H}$.

Since the projection of every state is a fixpoint, eventually there is a state s_1 such that $x_u.v = x_v$. After this, if $\texttt{request}_u$ is cleared it is never set. Also, $c5_u$ and $c6_u$ cannot be enabled after s_1. If $\texttt{request}_u$ is set, by Theorem 3, $c2_u$ is eventually executed which clears $ready_u$ and $\texttt{request}_u$. After $\texttt{request}_u$ is cleared $c1_u$ cannot be enabled. Therefore $ready_u$ is cleared throughout the rest of the computation. Thus, $c2_u$ cannot enabled. If $c1_u$ and $c2_u$ are never executed then eventually $a_u.v = b_v.u = c_v.u = d_u.v$, $r_u.v = ready_v$ and if $u < v$ then $y_u.v$ is cleared. Thus $c3_u$ and $c4_u$ are disabled and $\mathcal{C}$ reaches a fixpoint. $\Box$

Let σ_C and σ_H be computations of $\mathcal{C}$ and $\mathcal{H}$ respectively.

Lemma 7. If I_C holds and $h1_u$ continually enabled in the projection of σ_C, then $h1_u$ is eventually executed in σ_C.

Proof: Let s be a state of σ_C such that $h1_u$ is enabled in the projection of s. If $\texttt{request}_u$ is set in s then (by Theorem 3) $c2_u$ is eventually executed. Let s_1 be the state when $c2_u$ is executed. By Corollary 1, $x_u.v = x_v$ at s_1. Then, since $h1_u$ is enabled in the projection of s_1 at it is also enabled at s_1. Thus, $h1_u$ is executed at s_1.

If $\texttt{request}_u$ is not set in s there can be two cases:

- every P_v executes $h1_v$ only finitely many times during σ_C. Let s_2 be the state after P_v executed $h1_v$ for the last time. If for some neighbor $x_u.v \neq x_v$ in s_2, then either $c4_u$ or $c5_u$ are eventually executed which sets $\texttt{request}_u$. By Theorem 3 $h1_u$ is executed in σC. Let us consider $x_u.v = x_v$ for every neighbor P_v in s_2. Since $h1_u$ is enabled in the projection of s_2 and $x_u.v = x_v$ for every neighbor P_v the $h1_u$ is also enabled in s_2. since $\texttt{request}_u$ is cleared in s_2 and $h1_u$ is enabled, then $c6_u$ is enabled and remains enabled until executed. $c6_u$ sets $\texttt{request}_u$ which leads to eventual execution of $h1_u$.
- A neighbor P_v of P_u executes $h1_v$ infinitely many times. This implies that $c1_v$ and $c2_v$ are executed infinitely often. If $u < v$ by R_7, $x_v = x_u.v$ when $c1_v$ is enabled. Therefore P_u must execute either $c4_u$ or $c5_u$ infinitely often. Also, when $c4_u$ is executed $x_v \neq x_u.v$. Therefore, $\texttt{request}_u$ gets enabled which leads to eventual execution of $h1_u$.
  Similar argument applies to the case of $u > v$ and R_8.

□

Theorem 6 (Soundness). If a computation of $\mathcal{C}$, σ_C starts at a state where I_C holds, then the projection of σ_C, σ_H is a computation of $\mathcal{H}$.

Proof: By Corollary 1, when $c2_u$ is executed the images of the variables used in $\mathcal{H}$ are equal to their respective sources. Therefore, the projection of the application of $c2_u$ to a state of $\mathcal{C}$ is equivalent to the application of $h1_u$ to the projection of this state. Therefore the projection of σ_C is a sequence of states of $\mathcal{H}$ such that each consequent state is produced by an application of some $h1$ to the previous state (Recall that finite stuttering is eliminated by the definition of a projection). Therefore, to prove that the projection of σ_C is a computation of $\mathcal{H}$ we need to show that this projection is maximal and fair.

There can be two cases. If σ_C is finite, by Theorem 5 the projection of this computation ends in a fixpoint. Therefore, the projection is maximal. Since any finite computation is fair, this projection is a computation of σ_H.

If σ_C is infinite, by Theorem 5 the projection of this computation cannot end in a fixpoint. Lemma 7 implies that this computation is going to be fair. □

We call a state s of $\mathcal{C}$ *clean* if for any process P_u, $ready_u$ is cleared and the only guard that is possibly enabled at s is $c6_u$. Let $u < v$. In a clean state only $c6_u$ be enabled in P_u. Thus the following should also be true in every clean state:

- the token is held by $a_u.v$, that is: $a_u.v = b_v.u = c_v.u = d_u.v$;
- since $ready_v$ is cleared, $r_u.v$ and $y_u.v$ are also cleared;
- $x_u.v = x_v$;
- $\texttt{request}_u$ is cleared.

Theorem 7 (Completeness). For every computation σ_H there exists a computation of σ_C the projection of which is σ_H.

Proof: Let $s_0, s_1, s_2, \ldots$ be σ_H. We prove the theorem by constructing σ_C such that the projection of σ_C is σ_H.

Let t_0 be a clean state of σ_C such that for every P_u the value of x_u is the same as in s_0. Thus the projection of t_0 is s_0. In a clean state for the value of $x_u.v$ is the same as the source x_v. Therefore, if $g_u.k$ is enabled in t_0, it also evaluates to true in s_0. Therefore, $c6_u$ is enabled in t_0. The execution of $c6_u$ sets $\texttt{request}_u$ and, therefore, enables $c1_u$.

Let s_1 be a state produced by executing of $g_u.k$ at s_0. The execution of $c1_u$ increments $a_u.v$ for every neighbor P_v such that $u < v$. Thus $a_u.v$ relinquishes the token. If $a_u.v$ does not have the token there is an enabled GC such that the execution of this GC passes the token further until $a_u.v$ re-acquires the token.

Let us assume that after t_1, σ_C contains the sequence of states such that every state of this sequence is produced by executing a GC that passes the token as described in the previous paragraph. This sequence ends in a state t_i where P_v has $ready_u$ set and for every neighbor P_v, $a_u.v = d_u.v$. Furthermore, $r_v.u$ is cleared for every neighbor P_v. Thus $c2_u$ is enabled at t_i.

Note that the sequence of states $t_0, \dots, t_i$ does not execute any GC of $\mathcal{H}$. Therefore the projection of this sequence produces just one state - s_0.

Let t_{i+1} be the state produced by executing $c2_u$ at t_i. The execution of $c2_u$ at t_{i+1} increments $a_u.v$ for every neighbor P_v such that $u > v$. Similar to the argument above we can construct a sequence which leads to the state t_j where $a_u.v = d_u.v$ for all neighbor P_v. Note that $t_{i+1}, \dots, t_j$ does not execute any GC of $\mathcal{H}$. Therefore the projection of this sequence onto $\mathcal{H}$ produces just one state - s_1.

Note also that t_j is a clean state. Similarly we can attach a sequence of states to σ_C that projects to s_2. Continuing in this manner we can construct a sequence of states σ_C such that the projection of it produces σ_H. If σ_H is finite then σ_C ends in a clean state where no $c6$ is enabled (*i.e.* a fixpoint). If σ_H is infinite, so is σ_C.

It remains to be proven that σ_C is fair. Note, that all GC that got enabled between t_0 and t_j were executed except for the GCs that correspond to GCs enabled in s_0. Note also that the GC that changes s_0 into s_1 was executed between t_0 and t_j. Since σ_H is fair and all enabled GCs are eventually executed, σ_C is also fair. $\square$

5 Extensions and Concluding Remarks

In this paper we presented a technique for stabilization-preserving atomicity refinement of concurrent programs. The refinement enables design of stabilizing programs in simple but restrictive model and implementation in a more complex but efficient model. Our refinement is based on a stabilizing, bounded-space, dining philosopher program in the more complex model. It is sound and complete, and fixpoint- and fairness-preserving.

In conclusion, we discuss three notable extensions of our refinement.

5.1 Semantics Refinement

Consider the semantics refinement problem where the abstract model employs interleaving semantics and the concrete model employs power-set semantics. We show how our refinement can be used to solve this problem. To demonstrate that our atomicity refinement is applicable to semantics refinement for power-set semantics we show that for a low-atomicity program a power-set computation is equivalent to an interleaving computation.

Two computations are *equivalent* if in both computations every P_u executes the same sequence of GCs and when a GC executes the values of the variables it reads are the same. Recall that in a computation under interleaving semantics (interleaving computation) each consequent state is produced by the execution of one of the GC that is enabled in the preceding state. In a computation under power-set semantics (power-set computation) each consequent state is produced by the execution of any number of GCs that are enabled in the preceding state.

Theorem 8. For every power-set computation of a low-atomicity program there is an equivalent interleaving computation.

Proof: To prove the theorem it is sufficient to demonstrate that for every pair of consequent states (s_1, s_2) of power-set computation there is an equivalent sequence of states of interleaving computation. The GCs executed in s_1 are either synchs or updates. Clearly, if the synchs are executed one after another and followed by the updates the resulting interleaving sequence is equivalent to the pair (s_1, s_2). □

5.2 Generalization to Drinking Philosophers Problem

Our refinement solution can be generalized to drinking philosophers problem [6] to further increase concurrency of the computation of the program. In the argument below we assume that $\mathcal{H}$ has multiple GCs. GCs of $\mathcal{H}$ *conflict* (affect each other) if one of them writes (updates) the variables the other GC reads. $\mathcal{DP}$ enforces MX of execution of GCs of $\mathcal{H}$ among neighbor processes. This is done regardless of whether these GCs actually conflict.

In $\mathcal{DP}$ to ensure MX among neighbor processes every pair of neighbors P_u and P_v maintains a sequence of handshake variables H_{uv}. Sending a token along this handshake sequence is used to inform the neighbor if the process is entering or exiting its CS. In a similar manner, P_u and P_v can have a sequence of handshake variables for every pair of of conflicting guarded commands. Then if a GC of $\mathcal{H}$ gets enabled the tokens are sent along each sequence to prevent the conflicting guarded commands from executing concurrently. [1] In the meantime non-conflicting GCs can execute concurrently.

5.3 Extension to Message-Passing Systems

Our refinement is further extended into message-passing model where the processes communicate via finite capacity lossy channels. To do so the underlying $\mathcal{DP}$ has to be modified so as it works in this model as follows.

The sequence of handshake variables H_{uv} between a pair of neighbors P_u and P_v is used in $\mathcal{DP}$ for process P_u to pass some information to P_v and get an acknowledgment that this information has been received. In message-passing systems an alternating-bit protocol (ABP) can be used for the same purpose. A formal model of dealing with lossy bounded channels in message-passing systems as well as a stabilizing ABP is presented in [14].

In this case P_u sends the value of a handshake variable (together with the rest of its state) to its neighbor in a message. If the message is lost it is retransmitted by a timeout. When P_u receives the message it copies the state of P_v (including the handshake variable) into its image variables and sends a reply back to P_u. When P_u gets the reply it knows that P_v got the original message. It has been

[1] More precisely: the tokens are sent to the processes with higher ids before the GC is executed and to the processes with smaller ids afterwards.

proven that the ABP stabilizes when the range of the handshake variables is greater than the sum of the capacity of the channels between P_u and P_v and in the opposite direction [14].

When $\mathcal{H}$ reaches a fixpoint the values of the variables of processes of $\mathcal{C}$ extended to message passing system do not change. Thus $\mathcal{C}$ is in a quiescent state. It is well-known [13] that a stabilizing message-passing program cannot reach a fixpoint. Therefore the extension of $\mathcal{DP}$ to message-passing systems no longer fixpoint-preserving: the timeout has to be executed even if the projection of the program has reached a fixpoint.

References

1. A. Arora and M. G. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.
2. A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.
3. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Bounding the unbounded (distributed computing protocols). In *Proceedings IEEE INFOCOM 94 The Conference on Computer Communications*, pages 776–783, 1994.
4. G. M. Brown, M. G. Gouda, and C. L. Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38:845–852, 1989.
5. J. E. Burns. Self-stabilizing rings without demons. Technical Report GIT-ICS-87/36, Georgia Tech, 1987.
6. K. M. Chandy and J. Misra. The drinking philosopher's problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.
7. K. M. Chandy and J. Misra. *Parallel Program Design : a Foundation*. Addison-Wesley, Reading, Mass., 1988.
8. E. W. Dijkstra. *Hierarchical Ordering of Sequential Processes*, pages 72–93. Academic Press, 1972.
9. E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.
10. S. Dolev, M. G. Gouda, and M. Schneider. Memory requirements for silent stabilization. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.
11. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
12. M. G. Gouda and F. Haddix. The alternator. In *Proceedings of the Fourth Workshop on Self-Stabilizing systems*, 1999. To appear.
13. M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.
14. R. R. Howell, M. Nesterenko, and M. Mizuno. Finite-state self-stabilizing protocols in message passing systems. In *Proceedings of the Fourth Workshop on Self-Stabilizing systems*, 1999. To appear.
15. L. Lamport. The mutual exclusion problem: part ii-statement and solutions. *Journal of the Association of the Computing Machinery*, 33:327–348, 1986.
16. M. Mizuno and H. Kakugawa. A timestamp based transformation of self-stabilizing programs for distributed computing environments. In *WDAG96 Distributed Algorithms 10th International Workshop Proceedings, Springer-Verlag LNCS:1151*, pages 304–321, 1996.

17. M. Mizuno and M. Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 1998.

18. M. Mizuno, M. Nesterenko, and H. Kakugawa. Lock-based self-stabilizing distributed mutual exclusion algorithms. In *Proceedings of the Sixteenth International Conference on Distributed Computing Systems*, pages 708–716, 1996.

Appendix

A Ring stabilization

Theorem 9. The disjunction of the following predicates is stabilizing for $\mathcal{DP}$.

$$\begin{aligned} &\forall(P_u, P_v) \in N : \\ &(a_u.v = b_v.u = c_v.u = d_v.u) \vee && (R_9) \\ &(a_u.v \neq b_v.u = c_v.u = d_v.u) \vee && (R_{10}) \\ &(a_u.v = b_v.u \neq c_v.u = d_v.u) \vee && (R_{11}) \\ &(a_u.v = b_v.u = c_v.u \neq d_v.u) && (R_{12}) \end{aligned}$$

Proof: We show closure first. Out of ten GCs of P_u and P_v only $dp1_u$, $dp2_u$, $dp4_v$, $dp3_v$, and $dp4_u$ can affect the predicates. When R_9 holds, only $dp1_u$ or $dp2_u$ can be enabled (depending on whether v is greater or smaller than u). When R_{10}, R_{11}, R_{12} hold only $dp4_v$, $dp3_v$ and $dp4_u$ are enabled respectively. Note that the execution of the enabled GC brings the program to a state where one of the predicates holds.

We demonstrate convergence by reduction of the program to Dijkstra K-state token circulation algorithm [9]. The difference between our handshake sequence and Dijkstra's algorithm is that $dp1_u$ (or $dp2_u$) may or may not be enabled when $a_u.v = d_v.u$. If the execution of the $dp1_u$ (or $dp2_u$) is weakly fair wrt $a_u.v = d_v.u$ in some computation σ_C. Then H_{uv} behaves like Dijkstra's algorithm and stabilizes. If execution of the $dp1_u$ (or $dp2_u$) is not weakly fair wrt $a_u.v = d_v.u$ in $a_u.v = d_v.u$ then there is an infinite suffix of the computation where $a_u.v = d_v.u$ and $a_u.v$ is not incremented. Clearly then, the program stabilizes to R_9. $\square$

B Acronyms and Notation

Acronyms	
MX	mutual exclusion problem
DP	dining philosophers problem
CS	critical section
GC	guarded command
ABP	alternating-bit protocol
Notation	
$\mathcal{P}$	a program
$\mathcal{DP}$	dining philosophers program
$\mathcal{H}$	high-atomicity program
$\mathcal{C}$	refined program
P	process
N	neighbor relation between processes
i, j, k, u, v	process identifiers
P_u, P_v	neighbor processes
R, S	predicates
a, b, c, d	handshake variables
H_{uv}	sequence of handshake variables between P_u and P_v
s, t	program states
I_{DP}	invariant of $\mathcal{DP}$
I_C	invariant of $\mathcal{C}$
σ_H	computation of $\mathcal{H}$
σ_C	computation of $\mathcal{C}$