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 powerset 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 powerset 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 fixedpoint-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 fixedpoint- 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:

\[
\begin{align*}
\text{process } P_i \\
\text{par } (\text{declarations}) \\
\text{var } (\text{declarations}) \\
\star[ \langle \text{guarded command} \rangle \ldots \langle \text{guarded command} \rangle ]
\end{align*}
\]

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

\[
\langle \text{list of names} \rangle : \langle \text{domain} \rangle
\]

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 \( x \cdot j \) defines a set of variables \{\( x \cdot j \mid j \in \{2, 5, 9\} \}\} and a parameterized guarded command \( GC \cdot j \) defines the set of GCs:

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

A guarded command has the syntax:

\[
\langle \text{guard} \rangle \rightarrow \langle \text{command} \rangle
\]

A guard is a boolean expression containing local and neighbor variables. A command is a finite comma expression 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:

\[
(||\langle \text{range} \rangle : \langle \text{assignments} \rangle)
\]

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

\[
\text{process } P_i \\
\text{par } j : (P_j, P_i) \in N \\
\var \\
\text{public} \\
\quad \text{ready}_i : \text{boolean}, \\
\quad a_i, j, c_i, j : (0..3) \\
\text{private} \\
\quad \text{request}_i : \text{boolean}, \\
\quad r_i, j, y_i, j : \text{boolean}, \\
\quad b_i, j, d_i, j : (0..3)
\]

\[
\begin{align*}
\text{(d1)} & \quad \text{request}_i \land \neg \text{ready}_i \land (\forall k : a_i, k = d_i, k) \land (\forall k : i : \neg y_k, k) \rightarrow \\
& \hspace{1cm} \text{ready}_i := \text{true}, \\
& \quad ([k > i : y_k, k := r_i, k, a_i, k := (a_i, k + 1) \mod 4]) \\
\text{(d2)} & \quad \text{ready}_i \land (\forall k : a_i, k = d_i, k) \land (\forall k < i : \neg r_k, k) \rightarrow \\
& \hspace{1cm} / \ast \text{critical section} \ast / \\
& \hspace{1cm} \text{ready}_i := \text{false}, \\
& \quad ([k < i : a_i, k := (a_i, k + 1) \mod 4]) \\
\text{(d3)} & \quad c_i, j \neq b_i, j \rightarrow \\
& \hspace{1cm} c_i, j := b_i, j \\
\text{(d4)} & \quad r_i, j \neq \text{ready}_j \lor (b_i, j \neq a_j, i) \lor (d_i, j \neq c_j, i) \lor (j > i \land \neg \text{ready}_j \land y_k, j) \rightarrow \\
& \hspace{1cm} r_i, j := \text{ready}_j, \\
& \hspace{1cm} b_i, j := a_j, i, \\
& \hspace{1cm} d_i, j := c_j, i, \\
& \hspace{1cm} \text{if } j > i \land \neg \text{ready}_j \land y_k, j \text{ then } y_k, j := \text{false} \text{ fi}
\]

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 $\text{ready}_i$. 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:

- 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 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.
- $a_i, b_i, j, c_i, d_i, j$ - 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 $H_{ij} = \langle a_i, b_i, c_i, d_i, j \rangle$. 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$_j$ 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$_j$ 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 $\text{ready}_{j}$ to it’s image $\text{r}_{i,j}$ 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:

\[
\text{there can be one and only one token in } H_{uv} \quad (R_1)
\]

This proposition is proven in the Appendix A.

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

\[
((u < v) \land (a_{u,v} = b_{u,v}) \land \text{ready}_{u}) \Rightarrow r_{v,u} \quad (R_2)
\]
\[
((u > v) \land (a_{u,v} = b_{u,v}) \land \neg \text{ready}_{u}) \Rightarrow \neg r_{v,u} \quad (R_3)
\]

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 $dp1_{u,v}$, $dp2_{u,v}$ and $dp4_{u,v}$ affect the variables of the predicate. $dp1_{u,v}$ is executed only when $a_{u,v} = b_{u,v}$. 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, $a_{u,v} = b_{u,v}$ when $dp1_{u,v}$ is executed. Therefore $dp1_{u,v}$ sets the antecedent of our predicate to false by clearing $\text{ready}_{u}$. GC $dp2_{u,v}$ also sets the antecedent of our predicate to false. If $dp4_{u,v}$ sets the antecedent to true (by setting $a_{u,v} = b_{u,v}$), then the consequent is also set to true (since $r_{v,u} = \text{ready}_{u}$ after the execution of $dp4_{u,v}$).

To demonstrate convergence we note that when $R_2$ does not hold $dp4_{u,v}$ is enabled. When $dp4_{u,v}$ is executed the predicate holds. \hfill \Box

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

\[
((u > v) \land (a_{u,v} = b_{u,v}) \land \neg \text{ready}_{u}) \Rightarrow \neg y_{v,u} \quad (R_4)
\]

We show closure first. The guarded commands that affect our predicate are: $dp1_{u,v}$, $dp2_{u,v}$, $dp1_{v}$ and $dp4_{u}$. $dp1_{u,v}$ and $dp2_{u,v}$ set the antecedent to false and therefore do not violate the predicate. $dp1_{u,v}$ affects only the consequent of our
predicate. By $R_3$ when the antecedent of our predicate is \textbf{true}, then $r_v.u = \textbf{false}$. Therefore, when the antecedent is \textbf{true} and $dp1_v$ is executed, variable $y_v.u$ remains \textbf{false} and our predicate is not violated.

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

We now define a predicate $I_{DP}$ (which stands for invariant of $D\mathcal{P}$) such that every computation of $D\mathcal{P}$ that starts at a state conforming to $I_{DP}$ satisfies safety and liveness. $I_{DP}$ is: for every pair of neighbor processes $R_1 \land R_2 \land R_3 \land R_4$. 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.** $D\mathcal{P}$ stabilizes to $I_{DP}$.

Thus every execution of $D\mathcal{P}$ 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 (by $R_2$) $r_v.u$ is also set. When $r_v.u$ is set $dp2_v$ cannot be enabled. \hfill $\square$

**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 $a_v.s$ variables and $dp2_1$ gets enabled. When $ready_1$ is set the only command that can manipulate the tokens is $dp2_1$. Therefore, $a_v.s$ 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 $P_v$, $a_v.v$ 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:

- $r_{u,v}$ 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 $ready_u$ is cleared. After such a state $ready_u$ is set again. Thus $dp1_v$ and $dp2_v$ are executed infinitely many times during the computation. If $ready_u$ is set, eventually $r_{u,v}$ is set as well. When $dp1_v$ is executed in a state where $r_{u,v}$ sets, this command sets $y_{u,v}$. $y_{u,v}$ is not cleared while $ready_u$ (and subsequently $r_{u,v}$) is set. When $y_{u,v}$ is set $dp1_v$ cannot be executed and $ready_u$ remains cleared while $ready_u$ is set. If $ready_u$ is cleared, eventually $r_{u,v}$ is cleared as well. Thus, $r_{u,v}$ stays cleared until $dp2_u$ is executed.

- $r_{u,v}$ is set in only finitely many states of the computation. In this case there is a suffix of the computation where $ready_u$ 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. \qed

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 $request_u$ is enabled then $dp1_u$ (that sets $ready_u$) is eventually executed. Let us assume that $request_u$ holds unless $ready_u$. By Proposition 2 for any $P_u$’s neighbor $P_v$, $a_v.v$ eventually gets the token. When $ready_u$ is cleared $a_v.v$ never gives up the token unless $dp1_v$ 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:

- $r_{u,v}$ is set in infinitely many states. By Lemma 3 this implies that $ready_u$ is also cleared in infinitely many states as well. Therefore, $dp1_u$ is executed infinitely many times. By Predicate $R_4$, $dp1_v$ can be executed only in a state where $y_{u,v}$ is cleared.
- \textit{ready}_v is set in only finitely many states. In this case there is a suffix of the execution where \textit{ready}_v is never set. If \( y_i.v \) is set then \( dp4_i \) is enabled. When \( dp4_i \) is executed \( y_i.v \) is cleared.

\[ \square \]

The following theorem unifies Lemmas 3 and 4.

\textbf{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

\begin{verbatim}
process \( P_i \)
var \( x_i \)

\[ s \]
\[ g_i(x_i, \langle x_k \mid (P_i, P_k) \in \mathcal{N} \rangle) \rightarrow x_i := f_i(x_i, \langle x_k \mid (P_i, P_k) \in \mathcal{N} \rangle) \]
\end{verbatim}

\textbf{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 \( DP \), a GC of \( \mathcal{H} \) can read any variable of the neighbor process and update its own variable in one GC.

4.2 Composing \( DP \) and \( \mathcal{H} \)

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

\[ \langle GC \text{ of } \mathcal{DP} \rangle \parallel \langle \text{command} \rangle \]

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, $I_{\mathcal{DP}}$ is also an invariant of $\mathcal{C}$.

\[
\text{process } P_i
\]
\[
\text{par } j : (P_i, P_j) \in N
\]
\[
\text{var }
\]
\[
\text{public } x_i
\]
\[
\text{private } x_{i,j}
\]
\[
\text{request}_i : \text{boolean}
\]
\[
[]
\]
\[
(1) \quad dp1
\]
\[
[]
\]
\[
(2) \quad dp2 \parallel \left( \text{ if } g_i(x_i, \{x_i, k | (P_i, P_k) \in N\}) \text{ then } x_i := f_i(x_i, \{x_i, k | (P_i, P_k) \in N\}) \text{ fi } \right)
\]
\[
[]
\]
\[
(3) \quad dp3
\]
\[
[]
\]
\[
(4) \quad dp4 \parallel (\text{ if } x_i, j \neq x, j \text{ then } x_i, j := x, j, \text{ request}_i := \text{true} \text { fi })
\]
\[
[]
\]
\[
(5) \quad x_i, j \neq x, j \rightarrow x_i, j := x, j, \text{ request}_i := \text{true}
\]
\[
[]
\]
\[
(6) \quad g_i(x_i, \{x_i, k | (P_i, P_k) \in N\}) \land \lnot \text{ request}_i \rightarrow
\]
\[
\text{ request}_i := \text{true}
\]
\]

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 $\text{request}_i$ is read by $\mathcal{DP}$. Yet it does not violate the liveness and safety properties of $\mathcal{DP}$ since no assumptions about this variable were used when these properties were proven.
The GCs of $\mathcal{DP}$ are shown in abbreviated form. We superpose the execution of $h_1$ on $d_p2$. Note that $c_2$ is an update GC. Therefore, the superposed command cannot read the value of $x_j$ of a neighbor $P_j$ directly as $h_1$ does. The image $x_i, j$ is used instead. We superpose copying of the value of $x_j$ into $x_i, j$ on $d_p4$. Thus, the images of neighbor variables of $\mathcal{H}$ are equal to the sources when $h_1$ is executed by $\mathcal{C}$. We add a superposition GC $c_5$ 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. 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 $h_1$ evaluates to true (c6). request$_i$ is cleared after $h_1$ 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, $c_2$ 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:

\[
((u < v) \land (a_u, v = d_u, v) \land ready_u) \Rightarrow (x_u, v = x_v) \quad (R_5)
\]

\[
((u > v) \land \neg r_u, v) \Rightarrow (x_u, v = x_v) \quad (R_6)
\]

**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 $R_5$: $c_{1u}, c_{2u}, c_{4u}, c_{5u}, \text{ and } c_{2v}$. $c_{1u}$ and $c_{2u}$ set the antecedent to false; $c_{4u}$ and $c_{5u}$ set the consequent to true. If $c_{2u}$ holds at a certain state then $\neg r_u, u$. By $R_2$ this implies that at this state the antecedent of $R_5$ is false. Therefore, the execution of $c_{2u}$ does not violate $R_5$.

To show the closure of $R_6$ we note that only $c_{4u}, c_{5u}, \text{ and } c_{2v}$ affect $R_6$. Again both $c_{4u}$ and $c_{5u}$ set the consequent to true. Holding of $c_{2v}$ 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 $c_{5u}$ is enabled and it remains enabled until executed. After $c_{5u}$ is executed the predicates hold. $\square$

The following corollary can be deduced from the lemma.

**Corollary 1.** If $I_{DP}$ holds, $c_2$ is executed only when the images of the neighbor variables are equal to the sources. That is:

\[
\forall (P_u, P_v) \in N : c_{2u} \Rightarrow (x_u, v = x_v)
\]
Lemma 6. \( C \) stabilizes to the following predicates:

\[(u < v) \land (a_u.v = b_v.u) \land \neg \text{ready}_u \Rightarrow (x_u = x_v.u) \quad (R_7)\]
\[((u > v) \land (a_u.v = b_v.u) \land \text{ready}_u) \Rightarrow (x_u = x_v.u) \quad (R_8)\]

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 \( R_7 \): \( c_1_u, c_2_u, c_4_u, c_5_u \). \( c_1_u \) and \( c_2_u \) set the antecedent to \( \text{false} \); \( c_4_u \) and \( c_5_u \) set the consequent to \( \text{true} \). Therefore, \( R_7 \) is not violated.

To demonstrate convergence we observe that when \( R_7 \) does not hold \( c_5_u \) is enabled and it remains enabled until executed. After \( c_5_u \) is executed the predicate holds.

We define the invariant for \( C \) (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. \( C \) 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 \( C \) onto \( \mathcal{H} \) we omit \( \mathcal{H} \) and just say a projection of \( C \). 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 \( C \). The following is true in \( s \):

- \( a_u.v = b_v.u = c_v.u = d_v.u \)
- \( r_u.v = \text{ready}_v \)
- \( \text{ready}_u \) 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 \( C \) is a fixpoint of \( \mathcal{H} \); and if a computation of \( C \) 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 $C$. By Proposition 3, at state $s$, $\text{ready}_u$ and $y_u, v$ (if $u < v$) are cleared, and $a_u, v$ has the token. Since $s$ is a fixpoint, $c_1_u$ is not enabled, therefore, $\text{request}_u$ is cleared at $s$.

Since $c_6_u$ is not enabled and $\text{request}_u$ is cleared, $h_1_u$ is not enabled either. By Proposition 3, $x_u, v$ is equal to $x_u$ at $s$. Therefore the projection of $s$ does not have $h_1$ enabled and this projection is a fixpoint.

We now show that the computation that starts at 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 $c_2_u$ is executed. Thus, if the projection of the initial state of the computation is a fixpoint, $h_1_u$ is not executed during this computation. Therefore the projection of every state of this computation is a fixed point of $H$.

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 $\text{request}_u$ is cleared it is never set. Also, $c_5_u$ and $c_6_u$ cannot be enabled after $s_1$. If $\text{request}_u$ is set, by Theorem 3, $c_2_u$ is eventually executed which clears $\text{ready}_u$ and $\text{request}_u$. After $\text{request}_u$ is cleared $c_1_u$ cannot be enabled. Therefore $\text{ready}_u$ is cleared throughout the rest of the computation. Thus, $c_2_u$ cannot enabled. If $c_1_u$ and $c_2_u$ are never executed then eventually $a_u, v = b_u, u = c_u, u = d_u, v, r_u, v = \text{ready}_u$ and if $u < v$ then $y_u, v$ is cleared. Thus $c_3_u$ and $c_4_u$ are disabled and $C$ reaches a fixpoint. □

Let $\sigma_C$ and $\sigma_H$ be computations of $C$ and $H$ respectively.

Lemma 7. If $I_C$ holds and $h_1_u$ continually enabled in the projection of $\sigma_C$, then $h_1_u$ is eventually executed in $\sigma_C$.

Proof: Let $s$ be a state of $\sigma_C$ such that $h_1_u$ is enabled in the projection of $s$. If $\text{request}_u$ is set in $s$ then (by Theorem 3) $c_2_u$ is eventually executed. Let $s_1$ be the state when $c_2_u$ is executed. By Corollary 1, $x_u, v = x_v$ at $s_1$. Then, since $h_1_u$ is enabled in the projection of $s_1$ at it is also enabled at $s_1$. Thus, $h_1_u$ is executed at $s_1$.

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

- every $P_u$ executes $h_1_u$ only finitely many times during $\sigma_C$. Let $s_2$ be the state after $P_u$ executed $h_1_u$ for the last time. If for some neighbor $x_u, v \neq x_v$ in $s_2$, then either $c_4_u$ or $c_5_u$ are eventually executed which sets $\text{request}_u$. By Theorem 3 $h_1_u$ is executed in $\sigma_C$. Let us consider $x_u, v = x_v$ for every neighbor $P_u$ in $s_2$. Since $h_1_u$ is enabled in the projection of $s_2$ and $x_u, v = x_v$ for every neighbor $P_u$ the $h_1_u$ is also enabled in $s_2$. Since $\text{request}_u$ is cleared in $s_2$ and $h_1_u$ is enabled, then $c_6_u$ is enabled and remains enabled until executed. $c_6_u$ sets $\text{request}_u$ which leads to eventual execution of $h_1_u$.

- A neighbor $P_u$ of $P_u$ executes $h_1_u$ infinitely many times. This implies that $c_1_u$ and $c_2_u$ are executed infinitely often. If $u < v$ by $R_7$, $x_u = x_v$ when $c_1_u$ is enabled. Therefore $P_u$ must execute either $c_4_u$ or $c_5_u$ infinitely often. Also, when $c_4_u$ is executed $x_u \neq x_u, v$. Therefore, $\text{request}_u$ gets enabled which leads to eventual execution of $h_1_u$.

Similar argument applies to the case of $u > v$ and $R_8$. 
Theorem 6 (Soundness). If a computation of $C$, $\sigma_C$ starts at a state where $I_C$ holds, then the projection of $\sigma_C$, $\sigma_H$ is a computation of $H$.

Proof: By Corollary 1, when $c_{2u}$ is executed the images of the variables used in $H$ are equal to their respective sources. Therefore, the projection of the application of $c_{2u}$ to a state of $C$ is equivalent to the application of $h_{1u}$ to the projection of this state. Therefore the projection of $\sigma_C$ is a sequence of states of $H$ such that each consequent state is produced by an application of some $h_1$ to the previous state (Recall that finite stuttering is eliminated by the definition of a projection). Therefore, to prove that the projection of $\sigma_C$ is a computation of $H$ we need to show that this projection is maximal and fair.

There can be two cases. If $\sigma_C$ 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 $\sigma_H$.

If $\sigma_C$ 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 $C$ clean if for any process $P_u$, $ready_u$ is cleared and the only guard that is possibly enabled at $s$ is $c_6_u$. Let $u < v$. In a clean state only $c_6_u$ 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;
- $x_u.v = x_v$;
- $request_u$ is cleared.

Theorem 7 (Completeness). For every computation $\sigma_H$ there exists a computation of $\sigma_C$ the projection of which is $\sigma_H$.

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

Let $t_0$ be a clean state of $\sigma_C$ 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 $x_v.v$ 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, $c_6_u$ is enabled in $t_0$. The execution of $c_6_u$ sets $request_u$ and, therefore, enables $c_1_u$.

Let $s_1$ be a state produced by executing of $g_u.k$ at $s_0$. The execution of $c_1_u$ 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 $t_1$, $\sigma_C$ 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, $r_v.a$ is cleared for every neighbor $P_v$. Thus $c_{2u}$ is enabled at $t_i$. □
Note that the sequence of states $t_0, \ldots, t_i$ 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 $c_{2u}$ at $t_i$. The execution of $c_{2u}$ 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 $t_{i+1}, \ldots, t_j$ 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 $\sigma_C$ that projects to $s_2$. Continuing in this manner we can construct a sequence of states $\sigma_C$ such that the projection of it produces $\sigma_H$. If $\sigma_H$ is finite then $\sigma_C$ ends in a clean state where no $\sigma$ is enabled (i.e. a fixpoint). If $\sigma_H$ is infinite, so is $\sigma_C$.

It remains to be proven that $\sigma_C$ 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 $\sigma_H$ is fair and all enabled GCs are eventually executed, $\sigma_C$ is also fair. □

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 $P_i$ 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 \((s_1, s_2)\).

\(\square\)

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. D\(\mathcal{P}\) enforces MX of execution of GCs of \(\mathcal{H}\) among neighbor processes. This is done regardless of whether these GCs actually conflict.

In D\(\mathcal{P}\) 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 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 D\(\mathcal{P}\) 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 D\(\mathcal{P}\) 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 AAB P stabilizes when the range of the handshake variables is greater than the sum of the capacity of the channels between \( P_a \) and \( P_o \) and in the opposite direction [14].

When \( H \) reaches a fixpoint the values of the variables of processes of \( C \) extended to message passing system do not change. Thus \( C \) is in a quiescent state. It is well-known [13] that a stabilizing message-passing program cannot reach a fixpoint. Therefore the extension of \( DP \) 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


Appendix

A Ring stabilization

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

$$
\forall (P_u, P_v) \in \mathcal{N} :
(a_u.v = b_v.u = c_v.u = d_v.u) \lor 
(a_u.v \neq b_v.u = c_v.u = d_v.u) \lor 
(a_u.v = b_v.u \neq c_v.u = d_v.u) \lor 
(a_u.v = b_v.u = c_v.u \neq d_v.u)
$$

**Proof:** We show closure first. Out of ten GCs of $P_u$ and $P_v$ only $dp1_u$, $dp2_u$, $dp4_u$, $dp3_u$, and $dp4_u$ can affect the predicates. When $R_0$ 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 $dp4_u$, $dp3_u$, 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 $\sigma_C$. 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 $R_0$. \qed
## B  Acronyms and Notation

<table>
<thead>
<tr>
<th>Acronyms</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MX</td>
<td>mutual exclusion problem</td>
</tr>
<tr>
<td>DP</td>
<td>dining philosophers problem</td>
</tr>
<tr>
<td>CS</td>
<td>critical section</td>
</tr>
<tr>
<td>GC</td>
<td>guarded command</td>
</tr>
<tr>
<td>ABP</td>
<td>alternating-bit protocol</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>a program</td>
</tr>
<tr>
<td>$DP$</td>
<td>dining philosophers program</td>
</tr>
<tr>
<td>$H$</td>
<td>high-atomicity program</td>
</tr>
<tr>
<td>$C$</td>
<td>refined program</td>
</tr>
<tr>
<td>$P$</td>
<td>process</td>
</tr>
<tr>
<td>$N$</td>
<td>neighbor relation between processes</td>
</tr>
<tr>
<td>$i, j, k, u, v$</td>
<td>process identifiers</td>
</tr>
<tr>
<td>$P_u, P_v$</td>
<td>neighbor processes</td>
</tr>
<tr>
<td>$R, S$</td>
<td>predicates</td>
</tr>
<tr>
<td>$a, b, c, d$</td>
<td>handshake variables</td>
</tr>
<tr>
<td>$H_{uv}$</td>
<td>sequence of handshake variables between $P_u$ and $P_v$</td>
</tr>
<tr>
<td>$s, t$</td>
<td>program states</td>
</tr>
<tr>
<td>$I_{DP}$</td>
<td>invariant of $DP$</td>
</tr>
<tr>
<td>$I_C$</td>
<td>invariant of $C$</td>
</tr>
<tr>
<td>$\sigma_H$</td>
<td>computation of $H$</td>
</tr>
<tr>
<td>$\sigma_C$</td>
<td>computation of $C$</td>
</tr>
</tbody>
</table>