Verification of a Concurrent Deque Implementation

Robert D. Blumofe   C. Greg Plaxton   Sandip Ray
Department of Computer Science, University of Texas at Austin
{`rdb`, `plaxton`, `sandip`}`@cs.utexas.edu`

June 1999

Abstract

We prove the correctness of the concurrent deque component of a recent implementation of the work-stealing algorithm. Specifically, we prove that this concurrent deque implementation is synchronizable. Synchronizability is a weaker condition than the more traditional notion of serializability. Our concurrent deque implementation is not serializable, but its synchronizability makes it sufficient for use in the work-stealing algorithm. Whereas serializability requires that concurrent method invocations appear as if they are executed atomically in some serial order, synchronizability allows some invocations to appear as if they are executed atomically at exactly the same time.

1 Introduction

In this paper we prove the correctness of the concurrent deque implementation given in [1] as a component of the work-stealing thread-scheduling algorithm. This implementation is nonblocking, meaning that slow or preempted processes cannot prevent other processes from making progress [2]. No mutual exclusion is used. This nonblocking property makes this implementation ideal for use in multiprogrammed multiprocessors in which processes can be preempted at arbitrary times by the operating system kernel.

A ***deque***, or double-ended queue, is a data structure that maintains a finite sequence of items and supports insertion or removal of an item at either end of the sequence. We refer to these two ends as bottom and top. A deque implementation is a set of methods, one for each of the four deque operations. We refer to these methods as `pushBottom`, `popBottom`, `pushTop`, and `popTop`. One or more processes manipulate the deque by invoking these methods. A nonblocking concurrent deque allows the execution of two or more method invocations to be arbitrarily interleaved.

The nonblocking concurrent deque implementation of [1] does not provide a true concurrent deque as defined in the preceding paragraph, as it only specifies methods for three of the four deque operations, and it restricts the set of processes allowed to invoke each of these methods. Specifically, this concurrent deque implementation is subject to the following assumptions and limitations:

1. The set of processes allowed to access the deque consists of a single ***owner*** and some number of ***thieves***.

2. The owner invokes the `pushBottom` and `popBottom` methods only.

3. Thieves invoke the `popTop` method only.

For the sake of brevity, in the rest of the paper we will use the term deque to refer to a concurrent deque subject to the above restrictions.

This research is supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0150 from the U.S. Air Force Research Laboratory. In addition, Greg Plaxton is supported by the National Science Foundation under Grant CCR–9504145. Multiprocessor computing facilities were provided through a generous donation by Sun Microsystems.

We introduce "synchronizability", a new correctness criteria that is weaker than the more traditional notion of serializability [3], and we prove that our nonblocking deque implementation is synchronizable. Whereas serializability requires that concurrent method invocations appear as if they are executed atomically in some serial order, synchronizability allows some invocations to appear as if they are executed atomically at exactly the same time. The semantics of these method invocations are defined by a synchronous specification. In the case of our deque, multiple `popTop` invocations can appear to occur at exactly the same time, and when they do, our synchronous specification dictates that if the deque is nonempty, then one of these invocations returns the topmost item and all others return `NIL`, even if the deque contains more than one item. This synchronizability turns out to be sufficient for the work-stealing algorithm [1]. Our deque implementation is not serializable, and we are not aware of a serializable nonblocking deque implementation that is either as simple or as fast as our synchronizable one.

The remainder of this paper is organized as follows. Section 2 introduces some basic terminology. Section 3 gives two specifications of a deque: a serial specification and a synchronous specification. Section 4 presents the nonblocking deque implementation of [1]. Section 5 proves the correctness of this implementation with respect to the synchronous specification of Section 3.

2 Basic Terminology

In this section, we define the terms that provide the framework for our proof. We begin by defining a "program" as a set of "methods." In our proof, we shall be concerned with a program whose methods are `pushBottom`, `popBottom`, and `popTop`. We then define an "execution" as an interleaving of method invocations by the various processes. Next, we define a "behavior" as the observable method calls and returns in an execution. We then define what it means for a program to be "correct" in terms of its behaviors. Finally, we provide two lemmas that are used in our proof.

Before defining programs and methods, we first define a "system" as a set of processes and a set of states. Each state is broken into a single shared state and one or more private states — one private state for each process. Formally, a ***system*** Φ is a set of ***processes*** $processes(\Phi)$, a set of ***shared-states*** $shared(\Phi)$, and a set of functions from $processes(\Phi)$ to $private(\Phi)$, where $private(\Phi)$ is a set of ***private-states***. Such a function assigns a private state to each process. A ***state*** of Φ is a shared-state and a function from $processes(\Phi)$ to $private(\Phi)$. Throughout the remainder of the paper, we assume a fixed system Φ.

A program is a set of methods, and we define a method by breaking it into a set of "actions." Each action corresponds to a contiguous sequence of one or more instructions in the method's implementation code. An "event" is the execution of an action by a process, and we assume that each event is atomic. A sequence of events constitutes a program execution. In addition to a set of actions, a method has several other components. A set of "start" states specifies the states in which the method can be invoked. An "enabling" relation specifies which actions can be executed by which processes in which states. A "transition" function specifies how the state is updated when an action is executed. Finally, a method has a set of argument sequences that specify the allowable arguments.

Formally, a ***program*** Π is a set of methods $methods(\Pi)$, and a ***method*** $\pi \in \Pi$ is a set of ***start states*** $start(\pi)$, a set of ***actions*** $actions(\pi)$, a set of ***argument sequences*** $args(\pi)$, an ***enabling relation*** $enabled(\pi)$, and a ***transition function*** $trans(\pi)$. The enabling relation is a relation over $actions(\pi) \times logs(\Pi) \times processes(\Phi)$. The set $logs(\Pi)$ will be defined and the interpretation of the enabling relation will be explained shortly. The transition function is a function from $actions(\pi) \times args(\pi)$ to the set of functions over $private(\Phi) \times shared(\Phi)$. The transition function specifies how the state is updated when a process executes an action with an appropriate argument sequence. Specifically, when a process executes an action with an appropriate argument sequence, the transition function, when applied to that action and argument sequence, gives a function over $private(\Phi) \times shared(\Phi)$. This function is then applied to the private-state of the process and the shared-state to generate a new private-state for that process and a new shared-state.

One action in $actions(\pi)$ is designated as the ***input action*** of π. A second action is designated as the ***output action*** of π. The input action takes a specified number of arguments; all other actions take zero arguments. The output action may return a value; no other action returns a value. (Remark: We assume that the return value of the output action, if any, is encoded in the state via the transition function.)

To understand the enabling relation, we first define logs, events, and executions. The set of all ***logs*** of a program Π, denoted $logs(\Pi)$, is the set of all pairs (u, α) such that u is a state and α is an event sequence. An ***event*** is a tuple (π, ψ, p, η, r) where π is a method of Π, ψ is an action of π, p is a process, η is an argument sequence of π, and r is an optional return value. Such an event denotes the execution of action ψ from method π by process p. In addition, if the action is the input action, then η specifies its arguments, and if the action is the output action, then r specifies its return value (if any). An event associated with an input (resp., output) action is a ***prologue*** (resp., ***epilogue***). Any event that is neither a prologue nor an epilogue is a ***burst***. A log is a state and a sequence of events. Some logs are executions.

Informally, a log (u, α) is an execution of a program if the state u is a start state of the program, and α is an event sequence that can be generated by processes executing the program. Formally, we define start states, executions, and final states as follows. For any program Π, we define the associated set of start states, denoted $start(\Pi)$, as the intersection over all π in $methods(\Pi)$ of $start(\pi)$. We now inductively define both the set of ***executions*** of a program Π, denoted $execs(\Pi)$, as well as the ***final state*** of each execution σ in $execs(\Pi)$, denoted $final(\Pi, \sigma)$.

- Every log $\sigma = (u, \epsilon)$, where u belongs to $start(\Pi)$ and ϵ denotes the empty sequence, is an execution. For such an execution $final(\Pi, \sigma) = u$.

- For all executions $\sigma = (u, \alpha)$ and all actions ψ such that (ψ, σ, p) belongs to $enabled(\pi)$ for some method π in $methods(\Pi)$, the log (u, β) is an execution, where β is the event sequence obtained by appending to the event sequence α any event of the form $x = (\pi, \psi, p, \eta, r)$, where η is a valid argument sequence (i.e., if ψ is an input action then η belongs to $args(\pi)$; otherwise, η is the empty sequence) and r is an optional return value (i.e., r is present if and only if ψ is an output action that produces a return value, in which case the value of r is determined by ψ and the current state). The state $final(\Pi, \sigma)$ is determined by updating the private-state of p and the shared-state according to the transition function of ψ.

We can now understand the meaning of the enabling relation. For an action ψ, an execution $\sigma = (u, \alpha)$, and a process p, if we have $(\psi, \sigma, p) \in enabled(\pi)$, then ψ can be executed by process p in state $final(\Pi, \sigma)$. In other words, if π is the method to which ψ belongs, then for an event $x = (\pi, \psi, p, \eta, r)$, the log $(u, \alpha x)$ is an execution. Note that we allow the enabling relation of a method to depend on the entire history of the execution. Of course, in practice, the enabling relation of a method depends only on the current state.

A method π is defined to be ***feasible*** if and only if, for any action ψ, execution σ, and process p, membership of the triple (ψ, σ, p) in $enabled(\pi)$ depends only on ψ, p, and the private-state of p in $final(\Pi, \sigma)$. A program Π is ***feasible*** if and only if each method in $methods(\Pi)$ is feasible. In Section 5 of this paper we prove the correctness of a feasible program by reasoning about a collection of related programs that are not feasible.

A program is "nonblocking" if every process always has an enabled action. Formally, a program Π is ***non-blocking*** if and only if, for all executions σ in $execs(\Pi)$, at least one action is enabled for each process in the state $final(\Pi, \sigma)$.

A "trace" is an event sequence that is part of some execution. Formally, the set of ***traces*** of a program Π, denoted $traces(\Pi)$, is defined as the set of all event sequences α such that $execs(\Pi)$ contains an execution of the form $(u, \beta\alpha)$. Given two events x and y in some trace α, we say that y is the ***successor*** of x (resp., x is the ***predecessor*** of y) if and only if x and y have the same associated process p and x immediately precedes y in the subsequence of α consisting of all events associated with process p. A trace is ***closed*** if and only if the first event associated with any process is a prologue and the last event associated with any process is an epilogue. An execution (u, α) is ***closed*** if and only if α is closed.

We now define the "futures" of an execution and the "histories" of a trace. Informally, the futures of an execution σ are those executions that extend σ with more events, and the histories of a trace α are those executions that can precede α. Formally, for any program Π and any execution $\sigma = (u, \alpha)$ in $execs(\Pi)$, we define the ***futures*** of σ, denoted $futures(\Pi, \sigma)$, as the set of all executions $(u, \alpha\beta)$ in $execs(\Pi)$. For any program Π and any trace α in $traces(\Pi)$, we define the ***histories*** of α, denoted $histories(\Pi, \alpha)$, as the set of all executions (u, β) such that $(u, \beta\alpha)$ belongs to $execs(\Pi)$.

We shall focus our attention on programs that are "well-formed" in the sense that their executions have the property that the events of each process occur in a reasonable order. Formally, an execution is ***well-formed*** if and only if the following conditions hold: the predecessor of each epilogue is either a burst or a prologue with the same associated method; the predecessor of each burst is either a prologue or a burst with the same associated method; the predecessor of each prologue is either an epilogue or does not exist. A program Π is ***well-formed*** if and only if each execution in $\mathit{execs}(\Pi)$ is well-formed.

We now define a behavior as the observable events — that is, the prologues and epilogues — in an execution. Formally, for any execution $\sigma = (u, \alpha)$, we define the ***behavior*** of σ, denoted $\mathit{behavior}(\sigma)$, as the sequence of all prologues and epilogues in α. For any program Π, we define the ***behaviors*** of Π, denoted $\mathit{behaviors}(\Pi)$, as the set $\{\mathit{behavior}(\sigma) : \sigma \in \mathit{execs}(\Pi)\}$. A behavior is ***well-formed*** if and only if the predecessor of each epilogue is a prologue with the same associated method and the predecessor of each prologue is either an epilogue or does not exist. A behavior is ***closed*** if and only if it is well-formed and each prologue has a successor.

The two types of behaviors with which we are most concerned are serial behaviors and synchronous behaviors. In a serial behavior, there is no interleaving of methods. Each prologue is followed immediately by its successor epilogue. In a synchronous behavior, multiple invocations can be "nested." We interpret nested invocations as occurring at the same time, hence the term "synchronous." Formally, we define these behaviors as follows. A behavior is an ***invocation*** if and only if it is closed and has length 2. We inductively define the set of all ***nested*** behaviors as follows: The empty sequence is a nested behavior, and any behavior of the form $x\alpha y$, where x is a prologue, α is a nested behavior, and y is the successor of x, is a nested behavior. A behavior is ***serial*** (resp., ***synchronous***) if and only if it is the concatenation of a number of invocations (resp., nested behaviors).

We define program correctness using "serializability" (resp., "synchronizability") which is defined as the ability to transform any behavior into a correct serial (resp., synchronous) behavior via interchanging events. The interchange of two adjacent events x and y (not necessarily from the same process) in a well-formed behavior constitutes a ***valid transposition*** if and only if the resulting event sequence is a well-formed behavior and either x is a prologue or y is an epilogue. Note that an interchange in which x is an epilogue and y is a prologue is not a valid transposition, because we do not want to change non-overlapping method invocations into overlapping ones. A ***serial (resp., synchronous) specification*** defines the set of ***correct*** serial (resp., synchronous) behaviors. A behavior is ***serializable*** (resp., ***synchronizable***) if and only if it can be transformed to a correct serial (resp., synchronous) behavior via a sequence of valid transpositions.

A program is ***correct*** with respect to a given serial (resp., synchronous) specification if and only if it is well-formed and for every execution σ in $\mathit{execs}(\Pi)$ there is an execution τ in $\mathit{futures}(\Pi, \sigma)$ such that $\mathit{behavior}(\tau)$ is serializable (resp., synchronizable).

For any epilogue occurring in some well-formed execution, we define the ***running time*** of the method invocation associated with the epilogue as the least i such that the ith iterated predecessor of the epilogue is a prologue. For any well-formed program Π and any method π in $\mathit{methods}(\Pi)$, we define method π to be ***constant-time*** if and only if there is a constant exceeding the running time associated with any epilogue of π occurring in any execution in $\mathit{execs}(\Pi)$. A program Π is ***constant-time*** if and only if it is well-formed and every method in $\mathit{methods}(\Pi)$ is constant-time. The proof of the following lemma is straightforward.

Lemma 1 *A constant-time nonblocking program Π is correct with respect to a given serial (resp., synchronous) specification if every closed behavior in $\mathit{behaviors}(\Pi)$ is serializable (resp., synchronizable).*

For any program Π and any pair of executions σ and τ in $\mathit{execs}(\Pi)$, we say that σ and τ are ***congruent*** with respect to Π, denoted $\sigma \cong \tau$, if and only if

$$\{\mathit{behavior}(\sigma') : \sigma' \in \mathit{futures}(\Pi, \sigma)\} \quad = \quad \{\mathit{behavior}(\tau') : \tau' \in \mathit{futures}(\Pi, \tau)\}.$$

For any program Π, any trace α in $\mathit{traces}(\Pi)$, and any set of traces X contained in $\mathit{traces}(\Pi)$, we say that α is ***subsumed*** by X with respect to Π, denoted $\alpha \to X$, if and only if for every execution (u, γ) in $\mathit{histories}(\Pi, \alpha)$ there is some β in X such that (u, γ) belongs to $\mathit{histories}(\Pi, \beta)$ and $(u, \gamma\alpha) \cong (u, \gamma\beta)$. (Remark: We define $\alpha \to \beta$ as a shorthand for $\alpha \to \{\beta\}$.) We make extensive use of the following basic lemma.

Lemma 2 *For any program Π, any traces α, β, and γ in $traces(\Pi)$, and any set of traces X in $traces(\Pi)$ such that $\beta \to X$, we have $\alpha\beta\gamma \to \{\alpha\delta\gamma : \delta \in X\}$.*

3 Deque Specification

A ***deque*** is a program with three methods: `pushBottom`, `popBottom`, and `popTop`. The `pushBottom` method takes a single non-`NIL` argument and does not return a value. The `popBottom` and `popTop` methods both take zero arguments and return a value. One process is designated as the ***owner*** of the deque; the owner invokes the `popBottom` and `pushBottom` methods only. Every other process is a ***thief***; thieves invoke the `popTop` method only.

We now give an inductive definition of the set of ***correct*** serial behaviors of a deque. In the following, the variables α and β denote serial behaviors.

1. The empty serial behavior is correct.
2. A serial behavior of the form $\alpha\mu$, where μ is a `pushBottom` invocation, is correct if and only if α is correct.
3. A serial behavior of the form $\mu\alpha$, where μ is a `popBottom` or `popTop` invocation, is correct if and only if the return value of μ is `NIL` and α is correct.
4. A serial behavior of the form $\alpha\mu\nu\beta$, where μ is a `pushBottom` invocation and ν is a `popBottom` invocation, is correct if and only if the return value of ν is equal to the argument of μ and $\alpha\beta$ is correct.
5. A serial behavior of the form $\mu\nu\alpha$, where μ is a `pushBottom` invocation and ν is a `popTop` invocation, is correct if and only if the return value of ν is equal to the argument of μ and α is correct.
6. A serial behavior of the form $\alpha\mu\nu\xi\beta$, where μ and ν are `pushBottom` invocations and ξ is a `popTop` invocation, is correct if and only if $\alpha\mu\xi\nu\beta$ is correct.

We now give an inductive definition of the set of ***correct*** synchronous behaviors of a deque.

1. Any correct serial behavior is a correct synchronous behavior.
2. A synchronous behavior of the form $\alpha x\mu y\beta$, where α and β are behaviors, x is a `popTop` prologue, μ is a `popTop` invocation returning a non-`NIL` value, and y is the successor of x, is correct if and only if the return value associated with y is `NIL` and the synchronous behavior $\alpha\mu\beta$ is correct.

4 A Deque Implementation

The deque implementation of [1] is given in Figures 1 and 2. Figure 1 shows the instance variables, and Figure 2 shows the method implementations. All instance variables reside in shared memory. The items are stored in an array `deq` that is indexed from 0 and is assumed to be infinite in size. The index of the top item and the index below the bottom item are stored in the variables `top` and `bot` respectively. An additional variable `tag` is a "uniquifier" and is required for correct operation. The `tag` and `top` variables are implemented as fields of a structure `age`, and this structure is assumed to fit within a single shared-memory cell that can be operated on atomically with load, store, and compare-and-swap instructions. The compare-and-swap instruction is described below.

In addition to the shared memory, the implementation assumes that each process has a private memory (e.g., a register file). A standard set of atomic machine instructions is assumed to be available for manipulating the contents of the private memories.

The implementation assumes that the following atomic instructions are available to operate on shared memory: load, store, and compare-and-swap. The compare-and-swap instruction, `cas`, operates as follows. It takes three operands. The first operand is a private-memory cell `addr` that holds the address of a shared-memory cell. The second and third operands are private-memory cells, `old` and `new`, holding arbitrary values. Let `[addr]` denote the shared-memory cell addressed by `addr`. The instruction `cas (addr, old, new)` compares the value stored

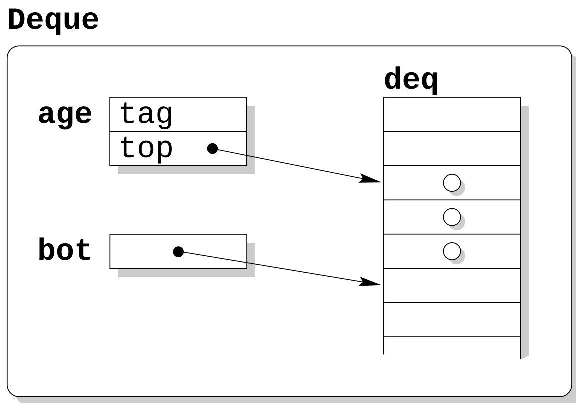

Figure 1: A deque object contains an array deq of items, a variable bot that is the index below the bottom item, and a variable age that contains two fields: top, the index of the top item, and tag, a “uniquifier” needed to ensure correct operation. All of these instance variables reside in shared memory. The variable age fits in a single cell of shared memory that can be operated on with atomic load, store, and compare-and-swap instructions.

```
void pushBottom (Item item)
load localBot ← bot
store item → deq[localBot]
localBot ← localBot + 1
store localBot → bot
```

```
Item popTop()
load oldAge ← age
load localBot ← bot
if localBot ≤ oldAge.top
    return NIL
load item ← deq[oldAge.top]
newAge ← oldAge
newAge.top ← newAge.top + 1
cas (age, oldAge, newAge)
if oldAge = newAge
    return item
return NIL
```

```
Item popBottom()
load localBot ← bot
if localBot = 0
    return NIL
localBot ← localBot − 1
store localBot → bot
load item ← deq[localBot]
load oldAge ← age
if localBot > oldAge.top
    return item
store 0 → bot
newAge.top ← 0
newAge.tag ← oldAge.tag + 1
if localBot = oldAge.top
    cas (age, oldAge, newAge)
    if oldAge = newAge
        return item
store newAge → age
return NIL
```

Figure 2: The three deque methods. The deque’s instance variables, age, bot, and deq, reside in shared memory; the remaining variables in this code reside in the process’s private memory. The load, store, and cas instructions operate atomically. Each return statement is assumed to assign the return value to a private variable returnValue.

in [addr] with the value stored in old, and if they are equal, [addr] is swapped with new. In this case, we say the cas ***succeeds***. Otherwise, it loads [addr] into new without modifying [addr]. In this case, we say the cas ***fails***. This whole operation — comparing and then either swapping or loading — is performed atomically with respect to all other memory operations. We can detect whether the cas fails or succeeds by comparing the value stored in old with the value stored in new after the cas. If they are equal, then the cas succeeded; otherwise, it failed.

The start states are those states in which bot $=$ age.top ≥ 0 and no process has an outstanding deque method invocation.

Color	*Thief Program Counter*	*Levels*
S	No outstanding `popTop` invocation	$[0, 16]$
A	Line 1 of `popTop`	$[0, 14]$
B	Line 2 of `popTop`	$[0, 0]$
C	Line 5 of `popTop`	$[0, 0]$
D	Line 8 of `popTop`	$[0, 0]$
E	Epilogue of `popTop`	$[0, 0]$

Table 1: Colors corresponding to particular thief program counter values.

5 Proof of Correctness

Let Δ_0 denote the deque of Section 4. The goal of the present section is to prove the correctness of Δ_0 with respect to the synchronous specification of Section 3. It is straightforward to prove that Δ_0 is constant-time and nonblocking. Thus, by Lemma 1, it remains only to prove that every closed behavior in $behaviors(\Delta_0)$ is synchronizable.

The sequence of events corresponding to the execution of any method consists of a prologue followed by a sequence of bursts followed by an epilogue. The bursts execute the method body. For each method in $methods(\Delta_0)$, there is an action corresponding to each individual instruction of the method body, that is, each burst executes a single instruction. The fine-grained nature of the actions of Δ_0 allows for a large number of possible interleavings of concurrent method invocations.

Instead of reasoning directly about the deque Δ_0, we find it convenient to define a sequence of "new" deques Δ_ℓ, $1 \le \ell \le 16$, each of which is based on the code of Figure 2, but where the granularity of the actions associated with each successive Δ_ℓ increases as a function of ℓ. Our proof then proceeds in two stages. In the first stage, we show that every closed behavior of deque Δ_ℓ is a closed behavior of $\Delta_{\ell+1}$, $0 \le \ell < 16$. In the second stage, we prove the correctness of Δ_{16}. The second stage is straightforward due to the appropriately large-grained atomicity of Δ_{16}.

For the sake of brevity, we refer to the actions, events, traces and executions associated with Δ_ℓ as ***ℓ-actions***, ***ℓ-events***, ***ℓ-traces***, and ***ℓ-executions***, respectively, $0 \le \ell \le 16$. We use the term ***ℓ-congruent*** (resp., ***ℓ-subsumed***) as a shorthand for the phrase "congruent (resp., subsumed) with respect to Δ_ℓ."

5.1 Process colors in a 0-execution

For any thief p and execution $\sigma = (u, \alpha)$ in $execs(\Delta_0)$, we now inductively define the ***count*** of p with respect to σ. If α is the empty trace, then the count of p with respect to σ is -1. Otherwise, α is of the form βx for some trace β and event x and, letting i' (resp., i) denote the count of p with respect to (u, α) (resp., (u, β)), i' is determined from i as follows: if $i = -1$ and x is a burst associated with p that executes the load instruction on line 1 of `popTop`, then $i' = 0$; if $i \ge 0$ and x writes the shared variable `age`, then $i' = i + 1$; if x is an epilogue associated with p, then $i' = -1$; otherwise, $i' = i$.

We now define a set of ***colors***. Table 1 (resp., Table 2) defines a set of colors corresponding to particular thief (resp., owner) program counter values. Table 3 (resp., Table 4) defines the remaining thief (resp., owner) colors; note that each of the latter color symbols corresponds (by removal of the subscript) to a unique thief (resp., owner) program counter value. Furthermore, each of the colors in Tables 3 and 4 has an associated list of assertions. (See Table 5 for the definitions of these assertions.)

Table 5 defines the state predicates P_i, $0 \le i \le 15$, and the execution predicates Q_0 and Q_1. In general, we say that a state predicate P holds for a process p with respect to a given 0-execution σ if and only if P holds for p in state $final(\Delta_0, \sigma)$.

Note that the assertions associated with thief colors are the state predicates P_i, $0 \le i \le 8$, and the execution predicate Q_0. We say that $Q_0(i)$ holds for a thief p with respect to a given 0-execution σ if and only if the count of p with respect to σ is equal to i. For any thief p and 0-execution σ, we say that p has color λ with respect to σ if and only if the program counter of p is consistent with λ, and any assertions associated with color λ hold for p with

Color	*Owner Program Counter*	*Interval*
S	No outstanding `popBottom` or `pushBottom` invocation	$[0, 0]$
A	Line 1 of `popBottom`	$[0, 0]$
B	Line 6 of `popBottom`	$[0, 0]$
C	Line 10 of `popBottom`	$[0, 0]$
D	Line 13 of `popBottom`	$[0, 0]$
E	Line 17 of `popBottom`	$[0, 0]$
F	Epilogue of `popBottom`	$[0, 0]$
G	Line 1 of `pushBottom`	$[0, 0]$
H	Line 4 of `pushBottom`	$[0, 0]$
I	Epilogue of `pushBottom`	$[0, 0]$

Table 2: Colors corresponding to particular owner program counter values.

Thief Color	*Assertions*	*Interval*
$\mathbf{A}_0$	P_0	$[15, 16]$
$\mathbf{A}_1$	$\neg P_0$	$[15, 16]$
$\mathbf{B}_{0,0}$	P_0, P_1, $Q_0(0)$	$[1, 15]$
$\mathbf{B}_{0,1}$	$\neg P_0$, P_1, $Q_0(0)$	$[1, 15]$
$\mathbf{B}_i$, $i > 0$	P_2, $Q_0(i)$	$[3, 16]$
$\mathbf{B}_{i,0}$, $i > 0$	P_2, P_3, $Q_0(i)$	$[1, 2]$
$\mathbf{B}_{i,1}$, $i > 0$	P_2, $\neg P_3$, $Q_0(i)$	$[1, 2]$
$\mathbf{C}_0$	P_1, P_4, $Q_0(0)$	$[1, 13]$
$\mathbf{C}_i$, $i > 0$	P_2, $Q_0(i)$	$[1, 13]$
$\mathbf{D}_0$	P_1, $P_5(0)$, P_6, P_7, $Q_0(0)$	$[1, 2]$
$\mathbf{D}_i$, $i > 0$	P_2, $Q_0(i)$	$[1, 2]$
$\mathbf{E}_0$	P_8	$[1, 16]$
$\mathbf{E}_1$	$\neg P_8$	$[1, 16]$

Table 3: The remaining thief colors. The assertions are defined in Table 5.

Owner Color	*Assertions*	*Interval*
$S_{0,0}$	$P_9(0)$, P_{10}, $P_{11}(0)$, Q_1	$[1, 16]$
S_0	$P_9(0)$, $P_{11}(0)$, P_{12}, Q_1	$[1, 16]$
S_i, $i > 0$	$P_9(0)$, $P_{11}(i)$	$[1, 16]$
$A_{0,0}$	$P_9(0)$, P_{10}, $P_{11}(0)$, Q_1	$[1, 16]$
A_0	$P_9(0)$, $P_{11}(0)$, P_{12}, Q_1	$[1, 16]$
A_i, $i > 0$	$P_9(0)$, $P_{11}(i)$	$[1, 16]$
B_0	$P_9(1)$, $P_{11}(-1)$, $P_{13}(0)$, Q_1	$[1, 12]$
B_1	$P_9(1)$, $P_{11}(0)$, $P_{13}(0)$	$[1, 11]$
B_i, $i > 1$	$P_9(1)$, $P_{11}(i-1)$, $P_{13}(0)$	$[1, 10]$
C_0	$P_9(0)$, $P_{11}(-1)$, $P_{13}(0)$, $\neg P_{14}$, Q_1	$[1, 9]$
C_1	P_1, P_7, $P_9(0)$, $P_{13}(0)$, P_{14}	$[1, 8]$
D_0	$P_5(1)$, $P_9(0)$, P_{10}, $\neg P_{14}$, P_{15}, Q_1	$[1, 7]$
D_1	P_1, $P_5(1)$, P_7, $P_9(0)$, P_{10}, P_{14}, P_{15}	$[1, 6]$
E_0	$P_5(1)$, $P_9(0)$, P_{10}, P_{15}, Q_1	$[1, 5]$
$F_{0,0}$	$\neg P_8$, $P_9(0)$, P_{10}, $P_{11}(0)$, Q_1	$[1, 16]$
$F_{0,1}$	P_8, $P_9(0)$, P_{10}, $P_{11}(0)$, Q_1	$[1, 16]$
F_0	P_8, $P_9(0)$, $P_{11}(0)$, P_{12}, Q_1	$[1, 16]$
F_i, $i > 0$	P_8, $P_9(0)$, $P_{11}(i)$	$[1, 16]$
G_0	P_7, $P_9(0)$, $P_{11}(0)$, Q_1	$[1, 16]$
G_i, $i > 0$	P_7, $P_9(0)$, $P_{11}(i)$	$[1, 16]$
H_0	$P_9(1)$, $P_{11}(0)$, $P_{13}(1)$, Q_1	$[1, 4]$
H_i, $i > 0$	$P_9(1)$, $P_{11}(i)$, $P_{13}(1)$	$[1, 4]$
I_0	$P_9(0)$, $P_{11}(0)$, P_{12}, Q_1	$[1, 16]$
I_i, $i > 0$	$P_9(0)$, $P_{11}(i)$	$[1, 16]$

Table 4: The remaining owner colors. The assertions are defined in Table 5.

Predicate	*Definition*
P_0	$\texttt{bot} > \texttt{age.top}$
P_1	$\texttt{age} = \texttt{oldAge}$
P_2	$\texttt{age} > \texttt{oldAge}$
P_3	$\texttt{bot} > \texttt{oldAge.top}$
P_4	$\texttt{deq}[\texttt{age.top}] \neq \texttt{NIL}$
$P_5(i)$	$\texttt{newAge.tag} = \texttt{oldAge.tag} + i$
P_6	$\texttt{newAge.top} = \texttt{oldAge.top} + 1$
P_7	$\texttt{item} \neq \texttt{NIL}$
P_8	$\texttt{returnValue} \neq \texttt{NIL}$
$P_9(i)$	$\texttt{deq}[j] \neq \texttt{NIL}, 0 \leq \texttt{age.top} \leq j < \texttt{bot} + i$
P_{10}	$\texttt{bot} = 0$
$P_{11}(i)$	$\texttt{bot} = \texttt{age.top} + i$
P_{12}	$\texttt{age.top} > 0$
$P_{13}(i)$	$\texttt{localBot} = \texttt{bot} + i$
P_{14}	$\texttt{localBot} = \texttt{age.top}$
P_{15}	$\texttt{newAge.top} = 0$
$Q_0(i)$	this thief has count i
Q_1	no thief has color $\mathbf{A}_0$, $\mathbf{B}_{0,0}$, $\mathbf{C}_0$, or $\mathbf{D}_0$

Table 5: List of the predicates appearing in Tables 3 and 4. The predicate P_2 should be interpreted as $(\texttt{age.tag} > \texttt{oldAge.tag} \vee (\texttt{age.tag} = \texttt{oldAge.tag} \wedge \texttt{age.top} > \texttt{oldAge.top}))$.

respect to σ.

The assertions associated with the owner colors are the state predicates P_1, P_5, and P_i, $7 \leq i \leq 15$, and the execution predicate Q_1. (Remark: It can be shown that, unlike Q_0, Q_1 is logically equivalent to a state predicate. Having introduced the machinery of execution predicates to handle Q_0, we find it convenient to treat Q_1 as an execution predicate.) For any 0-execution σ, we say that the owner has color λ with respect to σ if and only if the owner program counter is consistent with λ, and any assertions associated with color λ hold for the owner with respect to σ.

5.2 A sequence of deques

The goal of this section is to define the set of ℓ-actions associated with deque Δ_ℓ, $1 \leq \ell \leq 16$.

Table 6 defines a number of symbols corresponding to particular code blocks. We now define a set of ***special*** actions and associate a unique identifying symbol with each such special action. The code blocks of the special thief (resp., owner) actions are defined in the second column of Table 7 (resp., Table 8).

The guard of each special action will be defined momentarily. For the moment we simply point out that the guard of each special action ψ is at least as strong as the guard of the 0-action corresponding to the first instruction in the code block of ψ. In other words, a special action ψ is enabled for a given process p only if the program counter of p points to the first instruction of the code block of ψ. It follows that a special action is enabled in a given state only if a corresponding sequence of 0-actions is applicable in that state. Using this observation inductively, we conclude that every execution involving only special actions corresponds to a unique 0-execution. For each ℓ, $1 \leq \ell \leq 16$, we will define the set of ℓ-actions as some subset of the set of special actions. Thus each ℓ-execution corresponds to a unique 0-execution, and we can extend the color definitions of Section 5.1 to ℓ-executions as follows: The color of a process p with respect to an ℓ-execution is defined as the color of p in the corresponding 0-execution.

Having extended the notion of process color to ℓ-executions, we are now able to define the guard of each special action. A special action ψ is enabled for a process p with respect to a given ℓ-execution σ if and only if the color of p with respect to σ is equal to the color specified in the third column Table 7 (resp., Table 8)

Symbol	*Code Block*
$[$	`popTop` prologue
$\mathbf{a}$	Line 1 of `popTop`
$\mathbf{b}$	Lines 2 to 4 of `popTop`
$\mathbf{c}$	Lines 5 to 7 of `popTop`
$\mathbf{d}$	Lines 8 to 11 of `popTop`
$]$	`popTop` epilogue
$\langle$	`popBottom` prologue
a	Lines 1 to 5 of `popBottom`
b	Lines 6 to 9 of `popBottom`
c	Lines 10 to 12 of `popBottom`
d	Lines 13 to 16 of `popBottom`
e	Lines 17 to 18 of `popBottom`
$\rangle$	`popBottom` epilogue
$\{$	`pushBottom` prologue
g	Lines 1 to 3 of `pushBottom`
h	Line 4 of `pushBottom`
$\}$	`pushBottom` epilogue

Table 6: List of symbols denoting particular code blocks.

Action	*Code Block*	*Guard*	`bot`	`age`	`deq`	*Interval*
$[$	$[$	$\mathbf{S}$				$[0, 16]$
$\mathbf{a}$	$\mathbf{a}$	$\mathbf{A}$		R		$[0, 16]$
$\mathbf{a}'_0$	$\mathbf{abcd}$	$\mathbf{A}_0$	R	R/W	R	$[15, 16]$
$\mathbf{a}'_1$	$\mathbf{abcd}$	$\mathbf{A}_1$	R	R		$[15, 16]$
$\mathbf{b}$	$\mathbf{b}$	$\mathbf{B}$	R			$[0, 1]$
$\mathbf{b}_{0,0}$	$\mathbf{b}$	$\mathbf{B}_{0,0}$	R			$[1, 13]$
$\mathbf{b}_{0,1}$	$\mathbf{bcd}$	$\mathbf{B}_{0,1}$	R			$[1, 15]$
$\mathbf{b}_{i,0}, i > 0$	$\mathbf{b}$	$\mathbf{B}_{i,0}$	R			$[1, 3]$
$\mathbf{b}_{i,1}, i > 0$	$\mathbf{bcd}$	$\mathbf{B}_{i,1}$	R			$[1, 3]$
$\mathbf{b}'_{0,0}$	$\mathbf{bcd}$	$\mathbf{B}_{0,0}$	R	R/W	R	$[13, 15]$
$\mathbf{b}'_1$	$\mathbf{bcd}$	$\mathbf{B}_1$	R	R	R	$[3, 16]$
$\mathbf{b}'_i, i > 1$	$\mathbf{bcd}$	$\mathbf{B}_i$	R	R	R	$[3, 14]$
$\mathbf{c}$	$\mathbf{c}$	$\mathbf{C}$			R	$[0, 1]$
$\mathbf{c}_i, i \geq 0$	$\mathbf{c}$	$\mathbf{C}_i$			R	$[1, 2]$
$\mathbf{c}'_0$	$\mathbf{cd}$	$\mathbf{C}_0$		R/W	R	$[2, 13]$
$\mathbf{c}'_i, i > 0$	$\mathbf{cd}$	$\mathbf{C}_i$		R	R	$[2, 13]$
$\mathbf{d}$	$\mathbf{d}$	$\mathbf{D}$		R/W		$[0, 1]$
$\mathbf{d}_0$	$\mathbf{d}$	$\mathbf{D}_0$		R/W		$[1, 2]$
$\mathbf{d}_i, i > 0$	$\mathbf{d}$	$\mathbf{D}_i$		R		$[1, 2]$
$]$	$]$	$\mathbf{E}$				$[0, 16]$

Table 7: The special thief actions.

Action	*Code Block*	*Guard*	`bot`	`age`	`deq`	*Interval*
$\langle$	$\langle$	S				$[0, 16]$
a	a	A	R/W			$[0, 1]$
$a_{0,0}$	$abcde$	$A_{0,0}$	R			$[1, 16]$
a_0	a	A_0	R/W			$[1, 12]$
a_1	a	A_1	R/W			$[1, 12]$
$a_i, i > 1$	a	A_i	R/W			$[1, 10]$
a'_0	$abcde$	A_0	R/W	R/W	R	$[12, 16]$
a'_1	$abcde$	A_1	R/W	R/W	R	$[11, 16]$
$a'_i, i > 1$	$abcde$	A_i	R/W	R	R	$[10, 16]$
b	b	B		R	R	$[0, 1]$
b_0	b	B_0		R	R	$[1, 9]$
b_1	b	B_1		R	R	$[1, 8]$
$b_i, i > 1$	$bcde$	B_i		R	R	$[1, 10]$
b'_0	$bcde$	B_0	W	R	R	$[9, 12]$
b'_1	$bcde$	B_1	W	R/W	R	$[8, 11]$
c	c	C	W			$[0, 1]$
c_0	c	C_0	W			$[1, 7]$
c_1	c	C_1	W			$[1, 6]$
c'_0	cde	C_0	W			$[7, 9]$
c'_1	cde	C_1	W	R/W		$[6, 8]$
d	d	D		R/W		$[0, 1]$
d_0	d	D_0				$[1, 5]$
d_1	de	D_1		R/W		$[1, 6]$
d'_0	de	D_0		W		$[5, 7]$
e	e	E		W		$[0, 5]$
$\rangle$	$\rangle$	F				$[0, 16]$
$\{$	$\{$	S				$[0, 16]$
g	g	G	R		W	$[0, 1]$
$g_i, i \geq 0$	g	G_i	R		W	$[1, 4]$
$g'_i, i \geq 0$	gh	G_i	R/W		W	$[4, 16]$
h	h	H	W			$[0, 1]$
$h_i, i \geq 0$	h	H_i	W			$[1, 4]$
$\}$	$\}$	I				$[0, 16]$

Table 8: The special owner actions.

ℓ	ℓ-Outgoing	ℓ-Incoming
1	$\{a, b, c, d, g, h, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$	see caption
2	$\{\mathbf{c}_i, \mathbf{d}_i : i \geq 0\}$	$\{\mathbf{c}'_i : i \geq 0\}$
3	$\{\mathbf{b}_{i,0}, \mathbf{b}_{i,1} : i > 0\}$	$\{\mathbf{b}'_i : i > 0\}$
4	$\{g_i, h_i : i \geq 0\}$	$\{g'_i : i \geq 0\}$
5	$\{d_0, e\}$	$\{d'_0\}$
6	$\{c_1, d_1\}$	$\{c'_1\}$
7	$\{c_0, d'_0\}$	$\{c'_0\}$
8	$\{b_1, c'_1\}$	$\{b'_1\}$
9	$\{b_0, c'_0\}$	$\{b'_0\}$
10	$\{a_i, b_i : i > 1\}$	$\{a'_i : i > 1\}$
11	$\{b'_1\}$	$\{a'_1\}$
12	$\{a_0, a_1, b'_0\}$	$\{a'_0\}$
13	$\{\mathbf{b}_{0,0}\} \cup \{\mathbf{c}'_i : i \geq 0\}$	$\{\mathbf{b}'_{0,0}\}$
14	$\{\mathbf{b}'_i : i > 1\}$	
15	$\{\mathbf{b}'_{0,0}, \mathbf{b}_{0,1}\}$	$\{\mathbf{a}'_0, \mathbf{a}'_1\}$

Table 9: The ℓ-outgoing and ℓ-incoming ℓ-actions, $1 \leq \ell < 16$. The set of 1-incoming 1-actions is too large to fit in the table; it consists of all special actions for which the associated symbol is subscripted and unprimed.

ℓ	*Increment*	*Unemptying*	*Decrement*	*Emptying*	*Reset*	*Aging*	*Steal*
2	h_i, $i \geq 0$	h_0	a_i, $i \geq 0$	a_1	c_0, c_1	$\mathbf{c}'_0$, $\mathbf{d}_0$, d_1, e	$\mathbf{c}'_0$, $\mathbf{d}_0$
3	h_i, $i \geq 0$	h_0	a_i, $i \geq 0$	a_1	c_0, c_1	$\mathbf{c}'_0$, d_1, e	$\mathbf{c}'_0$
4	g'_i, h_i, $i \geq 0$	g'_0, h_0	a_i, $i \geq 0$	a_1	c_0, c_1	$\mathbf{c}'_0$, d_1, e	$\mathbf{c}'_0$
5	g'_i, $i \geq 0$	g'_0	a_i, $i \geq 0$	a_1	c_0, c_1	$\mathbf{c}'_0$, d'_0, d_1, e	$\mathbf{c}'_0$
6	g'_i, $i \geq 0$	g'_0	a_i, $i \geq 0$	a_1	c_0, c_1, c'_1	$\mathbf{c}'_0$, c'_1, d'_0, d_1	$\mathbf{c}'_0$
7	g'_i, $i \geq 0$	g'_0	a_i, $i \geq 0$	a_1	c_0, c'_0, c'_1	$\mathbf{c}'_0$, c'_0, c'_1, d'_0	$\mathbf{c}'_0$
8	g'_i, $i \geq 0$	g'_0	a_i, $i \geq 0$	a_1	b'_1, c'_0, c'_1	$\mathbf{c}'_0$, b'_1, c'_0, c'_1	$\mathbf{c}'_0$
9	g'_i, $i \geq 0$	g'_0	a_i, $i \geq 0$	a_1	b'_0, b'_1, c'_0	$\mathbf{c}'_0$, b'_0, b'_1, c'_0	$\mathbf{c}'_0$
10	g'_i, $i \geq 0$	g'_0	a_0, a_1, a_i, a'_i, $i > 1$	a_1	b'_0, b'_1	$\mathbf{c}'_0$, b'_0, b'_1	$\mathbf{c}'_0$
11	g'_i, $i \geq 0$	g'_0	a_0, a_1, a'_i, $i > 0$	a_1, a'_1	a'_1, b'_0, b'_1	$\mathbf{c}'_0$, a'_1, b'_0, b'_1	$\mathbf{c}'_0$
12	g'_i, $i \geq 0$	g'_0	a_0, a_1, a'_i, $i \geq 0$	a_1, a'_1	a'_0, a'_1, b'_0	$\mathbf{c}'_0$, a'_0, a'_1, b'_0	$\mathbf{c}'_0$
13	g'_i, $i \geq 0$	g'_0	a'_i, $i \geq 0$	a'_1	a'_0, a'_1	$\mathbf{b}'_{0,0}$, $\mathbf{c}'_0$, a'_0, a'_1	$\mathbf{b}'_{0,0}$, $\mathbf{c}'_0$
14	g'_i, $i \geq 0$	g'_0	a'_i, $i \geq 0$	a'_1	a'_0, a'_1	$\mathbf{b}'_{0,0}$, a'_0, a'_1	$\mathbf{b}'_{0,0}$
15	g'_i, $i \geq 0$	g'_0	a'_i, $i \geq 0$	a'_1	a'_0, a'_1	$\mathbf{a}'_0$, $\mathbf{b}'_{0,0}$, a'_0, a'_1	$\mathbf{a}'_0$, $\mathbf{b}'_{0,0}$
16	g'_i, $i \geq 0$	g'_0	a'_i, $i \geq 0$	a'_1	a'_0, a'_1	$\mathbf{a}'_0$, a'_0, a'_1	$\mathbf{a}'_0$

Table 10: Certain distinguished sets of ℓ-actions, $2 \leq \ell \leq 16$.

The `bot` (resp., `age`) column of Tables 7 and 8 indicates whether there exists an execution in which the associated action reads/writes the shared variable `bot` (resp., `age`). The `deq` column of Tables 7 and 8 indicates whether there exists an execution in which the associated action reads/writes some location of the shared array `deq`.

Note that Tables 7 and 8 define an interval for each special action. A special action ψ is an ***ℓ-action*** if and only if ℓ belongs to the interval of ψ. An ℓ-action is ***ℓ-outgoing (resp., ℓ-incoming)*** if and only if it is not an $(\ell + 1)$-action (resp., $(\ell - 1)$-action). For the sake of convenience, Table 9 lists the ℓ-outgoing and ℓ-incoming ℓ-actions, $1 \leq \ell < 16$.

For $2 \leq \ell \leq 16$, Table 10 defines the set of ***increment*** (resp., ***decrement***, ***emptying***, ***reset***, ***aging***, ***steal***) ℓ-actions.

Code Block	*Current Color*	*New Color*
[	**S**	**A**
$\mathbf{a}$	$\mathbf{A}_i$, $0 \le i \le 1$	$\mathbf{B}_{0,i}$
$\mathbf{abcd}$	$\mathbf{A}_i$, $0 \le i \le 1$	$\mathbf{E}_i$
$\mathbf{b}$	$\mathbf{B}_{i,0}$, $i \ge 0$	$\mathbf{C}_i$
$\mathbf{bcd}$	$\mathbf{B}_{0,0}$	$\mathbf{E}_0$
$\mathbf{bcd}$	$\mathbf{B}_{i,j}$, $i + j > 0$	$\mathbf{E}_1$
$\mathbf{c}$	$\mathbf{C}_i$, $i \ge 0$	$\mathbf{D}_i$
$\mathbf{cd}$	$\mathbf{C}_0$	$\mathbf{E}_0$
$\mathbf{cd}$	$\mathbf{C}_i$, $i > 0$	$\mathbf{E}_1$
$\mathbf{d}$	$\mathbf{D}_0$	$\mathbf{E}_0$
$\mathbf{d}$	$\mathbf{D}_i$, $i > 0$	$\mathbf{E}_1$
]	**E**	**S**

Table 11: This table shows the effect on the color of a thief p of a burst associated with p for which the associated code block is as specified in the first column. Remark: A thief with color $\mathbf{A}$ (resp., $\mathbf{E}$) either has color $\mathbf{A}_0$ or $\mathbf{A}_1$ (resp., $\mathbf{E}_0$ or $\mathbf{E}_1$).

5.3 Establishing ℓ-congruence of ℓ-executions

Tables 1, 2, 3, and 4 define an interval for each color. A color λ is an ***ℓ-color*** if and only if ℓ belongs to the interval associated with λ. It is straightforward to prove that for any ℓ-execution σ, each process has at most one ℓ-color with respect to σ. An ℓ-execution σ is ***ℓ-nice*** if and only if each process has an ℓ-color with respect to σ. The assignment of colors to processes induced by an ℓ-nice ℓ-execution σ is called the ***ℓ-coloring*** of σ. The following lemma is straightforward to prove.

Lemma 3 *Every ℓ-execution is ℓ-nice,* $1 \le \ell \le 16$.

In order to carry out the full details of many of our proofs, such as the proofs of Lemmas 3 and 4, it is useful to understand the relationship between the ℓ-coloring of an ℓ-execution (u, α) and the ℓ-coloring of an "extended" ℓ-execution $(u, \alpha x)$, where x is an ℓ-event. This relationship is summarized in Tables 11, 12, 13, 14, and 15.

Two ℓ-executions σ and τ are ***compatible*** if and only if the ℓ-coloring of σ is equal to the ℓ-coloring of τ, $behavior(\sigma) = behavior(\tau)$, and each of the following has the same value in $final(\Delta_\ell, \sigma)$ as in $final(\Delta_\ell, \tau)$: `bot`, `age`, and any private variable or `deq` array entry that is asserted to be non-NIL by some process (via the assertions associated with the ℓ-coloring).

We omit the proof of the following lemma, which is a straightforward (albeit lengthy) proof by induction.

Lemma 4 *Any pair of ℓ-compatible ℓ-executions are ℓ-congruent,* $1 \le \ell \le 16$.

We will also need to establish 0-congruence of certain pairs of 0-executions. The following trivial lemma is sufficient for our purposes.

Lemma 5 *Any pair of* 0*-executions σ and τ such that $behavior(\sigma) = behavior(\tau)$ and $final(\Delta_0, \sigma) = final(\Delta_0, \tau)$ are* 0*-congruent.*

Lemma 5 is used in the proof of the following lemma. We omit the proof of Lemma 6, which is straightforward.

Lemma 6 *Every closed* 0*-trace is* 0*-subsumed by a closed* 1*-trace.*

Our main technical lemma follows.

Lemma 7 *Every closed ℓ-trace is ℓ-subsumed by a set of closed $(\ell + 1)$-traces,* $1 \le \ell < 16$.

Code Block	*Current Color*	*New Color*
$\langle$	$S_{0,0}$	$A_{0,0}$
$\langle$	$S_i, i \geq 0$	A_i
$abcde$	$A_{0,0}$	$F_{0,0}$
a	$A_i, i \geq 0$	B_i
$abcde$	$A_i, 0 \leq i \leq 1$	$F_{0,i}$
$abcde$	$A_i, i > 1$	F_{i-1}
b	B_0	C_0
b	B_1	C_1
$bcde$	$B_i, 0 \leq i \leq 1$	$F_{0,i}$
$bcde$	$B_i, i > 1$	F_{i-1}
c	$C_i, 0 \leq i \leq 1$	D_i
cde	$C_i, 0 \leq i \leq 1$	$F_{0,i}$
d	D_0	E_0
de	$D_i, 0 \leq i \leq 1$	$F_{0,i}$
e	E_0	$F_{0,0}$
$\rangle$	$F_{0,0}$ or $F_{0,1}$	$S_{0,0}$
$\rangle$	$F_i, i > 0$	S_i
$\{$	$S_{0,0}$	G_0
$\{$	$S_i, i > 0$	G_i
g	$G_i, i \geq 0$	H_i
gh	$G_i, i \geq 0$	I_{i+1}
h	$H_i, i \geq 0$	I_{i+1}
$\}$	$I_i, i \geq 0$	S_i

Table 12: This table shows the effect on the color of the owner of a burst for which the associated code block is as specified in the first column.

Thief Color	*Unemptying*	*Emptying*	*Reset*	*Aging*
$\mathbf{S}$	$\mathbf{S}$	$\mathbf{S}$	$\mathbf{S}$	$\mathbf{S}$
$\mathbf{A}$	$\mathbf{A}$	$\mathbf{A}$	$\mathbf{A}$	$\mathbf{A}$
$\mathbf{A}_0$		$\mathbf{A}_1$		$\mathbf{A}_0$ or $\mathbf{A}_1$
$\mathbf{A}_1$	$\mathbf{A}_0$		$\mathbf{A}_1$	$\mathbf{A}_1$
$\mathbf{B}_{0,0}$		$\mathbf{B}_{0,1}$		$\mathbf{B}_{1,0}$ or $\mathbf{B}_{1,1}$
$\mathbf{B}_{0,1}$	$\mathbf{B}_{0,0}$		$\mathbf{B}_{0,1}$	$\mathbf{B}_{1,1}$
$\mathbf{C}_0$		$\mathbf{C}_0$	$\mathbf{C}_0$	$\mathbf{C}_1$
$\mathbf{C}_i$, $i > 0$	$\mathbf{C}_i$	$\mathbf{C}_i$	$\mathbf{C}_i$	$\mathbf{C}_{i+1}$
$\mathbf{D}_0$		$\mathbf{D}_0$	$\mathbf{D}_0$	$\mathbf{D}_1$
$\mathbf{D}_i$, $i > 0$	$\mathbf{D}_i$	$\mathbf{D}_i$	$\mathbf{D}_i$	$\mathbf{D}_{i+1}$
$\mathbf{E}_i$, $0 \leq i \leq 1$	$\mathbf{E}_i$	$\mathbf{E}_i$	$\mathbf{E}_i$	$\mathbf{E}_i$

Table 13: For $2 \leq \ell \leq 16$ and λ an ℓ-color appearing in the first column, this table is useful for determining the effect on the ℓ-color λ of a thief p of an ℓ-event x associated with another process. Let ψ denoted the action associated with x. If ψ is not an unemptying, emptying, reset, or aging action, then it has no effect on the ℓ-color of p. If ψ is an unemptying action, the effect is shown in the second column. If ψ is an emptying action and not a reset action, then it is not an aging action and the effect is shown in the third column. If ψ is an emptying action and a reset action, then it is also an aging action and the effect is given by the composition of the third, fourth, and fifth columns (applying emptying first, reset second, and aging third). If ψ is a reset action and not an aging action, then it is not an emptying action and the effect is given by the fourth column. If ψ is a reset action and an aging action but not an emptying action, then the effect is given by composing the fourth and fifth columns (applying reset first and aging second). If ψ is an aging action but not a reset action, then it is not an emptying action and the effect is given by the fifth column. Blank entries assert that certain situations cannot arise; for example, if some thief has ℓ-color $\mathbf{A}_0$, then an event associated with an unemptying action cannot occur.

Thief Color	*Increment*	*Decrement*	*Reset*	*Aging*
$\mathbf{B}_i$, $i > 0$	$\mathbf{B}_i$	$\mathbf{B}_i$	$\mathbf{B}_i$	$\mathbf{B}_{i+1}$
$\mathbf{B}_{i,0}$, $i > 0$	$\mathbf{B}_{i,0}$	$\mathbf{B}_{i,0}$ or $\mathbf{B}_{i,1}$	$\mathbf{B}_{i,1}$	$\mathbf{B}_{i+1,0}$
$\mathbf{B}_{i,1}$, $i > 0$	$\mathbf{B}_{i,0}$ or $\mathbf{B}_{i,1}$	$\mathbf{B}_{i,1}$	$\mathbf{B}_{i,1}$	$\mathbf{B}_{i+1,1}$

Table 14: For $2 \leq \ell \leq 16$ and λ an ℓ-color appearing in the first column, this table is useful for determining the effect on the ℓ-color λ of a thief p of an ℓ-event x associated with another process. Let ψ denoted the action associated with x. If ψ is not an increment, decrement, reset, or aging action, then it has no effect on the ℓ-color of p. If ψ is an increment action, the effect is shown in the second column. If ψ is a decrement action and not a reset action, then it is not an aging action and the effect is shown in the third column. If ψ is a decrement action and a reset action, then it is also an aging action and the effect is given by the composition of the third, fourth, and fifth columns (applying decrement first, reset second, and aging third). If ψ is a reset action and not an aging action, then it is not a decrement action and the effect is given by the fourth column. If ψ is a reset action and an aging action but not a decrement action, then the effect is given by composing the fourth and fifth columns (applying reset first and aging second). If ψ is an aging action but not a reset action, then it is not a decrement action and the effect is given by the fifth column.

Owner Color	*Steal*
$S_{0,0}$	
S_0	
$S_i,\, i > 0$	S_{i-1}
$A_{0,0}$	
A_0	
$A_i,\, i > 0$	A_{i-1}
B_0	
$B_i,\, i > 0$	B_{i-1}
C_0	
C_1	C_0
D_0	
D_1	D_0
E_0	
$F_{0,i},\, 0 \leq i \leq 1$	
$F_{0,1}$	
F_0	
$F_i,\, i > 0$	F_{i-1}
G_0	
$G_i,\, i > 0$	G_{i-1}
H_0	
$H_i,\, i > 0$	H_{i-1}
I_0	
$I_i,\, i > 0$	I_{i-1}

Table 15: For $2 \leq \ell \leq 16$ and λ an ℓ-color appearing in the first column, this table is useful for determining the effect on the ℓ-color λ of the owner of an ℓ-event x associated with a thief. Let ψ denoted the action associated with x. If ψ is not a steal action, then it has no effect on the ℓ-color of the owner. If ψ is a steal action, the effect is shown in the second column. Blank entries assert that certain situations cannot arise; for example, if the owner has ℓ-color $S_{0,0}$, then ψ cannot be a steal action.

Proof: Fix ℓ, $1 \leq \ell < 16$, let R denote the set of rewrite rules specified by Tables 16 and 17 for this value of ℓ, and let α denote an arbitrary closed ℓ-trace. It is straightforward to prove that a finite sequence of applications of the rewrite rules in R can be used to obtain a set of closed ℓ-traces X such that α is ℓ-subsumed by X and no action associated with an event in a trace of X is ℓ-outgoing. The claim then follows immediately, since a (closed) ℓ-trace containing no ℓ-outgoing actions is also a (closed) $(\ell + 1)$-trace. ■

In Lemmas 8 and 9 below, we use the term "synchronizable" to mean "synchronizable with respect to the synchronous specification of Section 3." The following lemma is straightforward to prove using the three rewriting rules in Table 18.

Lemma 8 *Every closed behavior in* $behaviors(\Delta_{16})$ *is synchronizable.*

Lemmas 6, 7, and 8 together imply the following result.

Lemma 9 *Every closed behavior in* $behaviors(\Delta_0)$ *is synchronizable.*

As observed at the beginning of this section, Lemma 9 implies our main result.

Theorem 10 *The deque of Section 4 is correct with respect to the synchronous specification of Section 3.*

Acknowledgments

The authors benefited greatly from discussions with members of the UT Austin formal methods group.

References

[1] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, Puerto Vallarta, Mexico, June 1998.

[2] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, 1990.

[3] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8:142–153, 1986.

ℓ	*Rewriting Rule*	*Conditions*
1	$\mathbf{b} \to \{\mathbf{b}_{i,j} : i \geq 0, 0 \leq j \leq 1\}$	
	$\mathbf{c} \to \{\mathbf{c}_i : i \geq 0\}$	
	$\mathbf{d} \to \{\mathbf{d}_i : i \geq 0\}$	
	$a \to \{a_{0,0}\} \cup \{a_i : i \geq 0\}$	
	$b \to \{b_i : i \geq 0\}$	
	$c \to \{c_0, c_1\}$	
	$d \to \{d_0, d_1\}$	
	$g \to \{g_i : i \geq 0\}$	
	$h \to \{h_i : i \geq 0\}$	
2	$\mathbf{c}_i\mathbf{d}_i \to \mathbf{c}'_i$	$i \geq 0$
	$\mathbf{c}_i\psi \to \psi\mathbf{c}_i$	$i \geq 0$, $\neg aging(\psi)$
	$\mathbf{c}_i\psi \to \psi\mathbf{c}_{i+1}$	$i \geq 0$, $aging(\psi)$
3	$\mathbf{b}_{i,1} \to \mathbf{b}'_i$	$i > 0$
	$\mathbf{b}_{i,0}\mathbf{c}'_i \to \mathbf{b}'_i$	$i > 0$
	$\psi\mathbf{c}'_i \to \mathbf{c}'_i\psi$	$i > 0$, $\neg aging(\psi)$
	$\psi\mathbf{c}'_i \to \mathbf{c}'_{i-1}\psi$	$i > 1$, $aging(\psi)$
4	$g_ih_i \to g'_i$	$i \geq 0$
	$g_i\varphi \to \varphi g_i$	$i \geq 0$, $\neg aging(\varphi)$
	$g_i\mathbf{c}'_0 \to \mathbf{c}'_0 g_{i-1}$	$i > 0$
5	$d_0e \to d'_0$	
	$d_0\varphi \to \varphi d_0$	
	$d_1\mathbf{c}'_0 \to \mathbf{c}'_0 d_0$	
6	$c_1d_1 \to c'_1$	
	$c_1\varphi \to \varphi c_1$	$\neg aging(\varphi)$
	$c_1\mathbf{c}'_0 \to \mathbf{c}'_0 c_0$	
7	$c_0d'_0 \to c'_0$	
	$c_0\varphi \to \varphi c_0$	$\neg aging(\varphi)$
8	$b_1c'_1 \to b'_1$	
	$b_1\varphi \to \varphi b_1$	$\neg aging(\varphi)$
	$b_1\mathbf{c}'_0 \to \mathbf{c}'_0 b_0$	
9	$b_0c'_0 \to b'_0$	
	$b_0\varphi \to \varphi b_0$	
10	$a_ib_i \to a'_i$	$i > 1$
	$a_i\varphi \to \varphi a_i$	$i > 1$, $\neg aging(\varphi)$
	$a_i\mathbf{c}'_0 \to \mathbf{c}'_0 a_{i-1}$	$i > 1$

Table 16: Rewriting rules used in the proof of Lemma 7. For a given value of ℓ, each rule of the form $\alpha \to \beta$ (resp., $\alpha \to X$) signifies that the trace α is ℓ-subsumed by the trace β (resp., set of traces X). Each owner action appearing in a trace denotes a burst associated with the owner. Each thief action appearing in a trace denotes a burst associated with an arbitrary thief. The action variable φ denotes a thief action. Some of the rules involve more than one thief action; in such cases, we rely on the following conventions to indicate whether two actions are intended to be associated with the same thief, or with different thieves: (i) the action variable ψ, when used to denote a thief action, corresponds to a different thief than any other symbol in the trace, (ii) multiple explicitly specified thief actions appearing in the same rule (including the conditions portion of the rule) are understood to be associated with the same thief, except that a "hat" superscript denotes a different thief. Finally, the predicate $aging(\psi)$ holds if and only if ψ is an aging ℓ-action. See Table 10 for a list of aging ℓ-actions.

ℓ	*Rewriting Rule*	*Conditions*
11	$a_1 b'_1 \to a'_1$	
	$\varphi b'_1 \to b'_1 \varphi$	$\varphi \notin \{\mathbf{a}, \mathbf{b}_{0,1}\} \cup \{\mathbf{b}'_i, \mathbf{c}'_i : i > 0\}$
	$\mathbf{b}_{0,1} b'_1 \to b'_1 \mathbf{b}'_1$	
	$\mathbf{b}'_i b'_1 \to b'_1 \mathbf{b}'_{i+1}$	$i > 0$
	$\mathbf{c}'_i b'_1 \to b'_1 \mathbf{c}'_{i+1}$	$i > 0$
	$\mathbf{a} b'_1 \mathbf{b}'_1 \to b'_1 \mathbf{a} \mathbf{b}_{0,1}$	
	$\psi \mathbf{b}'_i \to \mathbf{b}'_i \psi$	$i > 0, \neg aging(\psi)$
	$\psi \mathbf{b}'_i \to \mathbf{b}'_{i-1} \psi$	$i > 1$, $aging(\psi)$
12	$a_0 b'_0 \to a'_0$	
	$a_1 \mathbf{c}'_0 \to \mathbf{c}'_0 a_0$	
	$\varphi b'_0 \to b'_0 \varphi$	$\varphi \notin \{\mathbf{a}, \mathbf{b}_{0,1}, \mathbf{c}'_0\} \cup \{\mathbf{b}'_i, \mathbf{c}'_i : i > 0\}$
	$\mathbf{b}_{0,1} b'_0 \to b'_0 \mathbf{b}'_1$	
	$\varphi \mathbf{c}'_0 b'_0 \to \mathbf{c}'_0 b'_0 \varphi$	$\varphi \neq \hat{\mathbf{a}}$
	$\mathbf{b}'_i b'_0 \to b'_0 \mathbf{b}'_{i+1}$	$i > 0$
	$\mathbf{c}'_i b'_0 \to b'_0 \mathbf{c}'_{i+1}$	$i > 0$
	$\mathbf{a} b'_0 \mathbf{b}'_1 \to b'_0 \mathbf{a} \mathbf{b}_{0,1}$	
	$\mathbf{a} \hat{\mathbf{c}}'_0 b'_0 \mathbf{b}'_2 \to \hat{\mathbf{c}}'_0 b'_0 \mathbf{a} \mathbf{b}_{0,1}$	
	$\psi \mathbf{b}'_i \to \mathbf{b}'_i \psi$	$i > 0, \neg aging(\psi)$
	$\psi \mathbf{b}'_i \to \mathbf{b}'_{i-1} \psi$	$i > 1$, $aging(\psi)$
13	$\mathbf{b}_{0,0} \mathbf{c}'_0 \to \mathbf{b}'_{0,0}$	
	$\psi \mathbf{c}'_i \to \mathbf{c}'_i \psi$	$i \geq 0, \neg aging(\psi)$
	$\psi \mathbf{c}'_i \to \mathbf{c}'_{i-1} \psi$	$i > 1$, $aging(\psi)$
	$\mathbf{b}_{0,0} \psi \to \psi \mathbf{b}_{0,0}$	$\neg aging(\psi)$
	$\mathbf{b}_{0,0} \psi \mathbf{c}'_1 \to \psi \mathbf{b}'_1$	$aging(\psi)$
14	$\psi \mathbf{b}'_i \to \mathbf{b}'_i \psi$	$i > 0, \neg aging(\psi)$
	$\psi \mathbf{b}'_i \to \mathbf{b}'_{i-1} \psi$	$i > 1$, $aging(\psi)$
15	$\mathbf{a} \mathbf{b}'_{0,0} \to \mathbf{a}'_0$	
	$\mathbf{a} \mathbf{b}_{0,1} \to \mathbf{a}'_1$	
	$\mathbf{a} \psi \to \psi \mathbf{a}$	$\neg aging(\psi)$

Table 17: Additional rewriting rules used in the proof of Lemma 7. See the caption of Table 16 for some notational remarks.

Rewriting Rule	*Conditions*
$\mathbf{a} \psi \to \psi \mathbf{a}$	$\neg aging(\psi)$
$\psi \mathbf{b}'_1 \to \mathbf{b}'_1 \psi$	$\neg aging(\psi)$
$\mathbf{a} \psi \mathbf{b}'_1 \to \mathbf{a}'_1$	$\psi \in \{a'_0, a'_1\}$

Table 18: Rewriting rules used in the proof of Lemma 8. See the caption of Table 16 for some notational remarks.