A Concurrent Buffer

Uri Abraham *

March 28, 2014

Abstract

Using the notions of state and history we analyze and reason about a two-process concurrent buffer algorithm.

1 Concurrent buffer queue specification

We repeat in this section some definitions that were made in a previous lecture about queues, but now the queue is assumed to have a bounded capacity. A queue is a FIFO data structure which supports two operations, enqueue and dequeue, so that the value of any dequeue operation \( D \) is the value of the earliest enqueue \( E \) that precedes \( D \) and whose value has not yet been dequeued. We want in this section to answer the following specification problem. Suppose that we have a sequence of events \( e_0, e_1, \ldots \) which may be infinite, and is such that every event \( e_i \) is either a dequeue event (we write \( \text{deq}(e_i) \)) or is an enqueue event (and we write then \( \text{enq}(e_i) \)). Suppose moreover that we have a function \( \text{Val} \) defined over these events with value into some Data domain. The question is this: under what conditions can we say that the sequence is a sequence of valid operations on a bounded fifo queue?

A queue-state is represented by a finite sequence \( Q \) of Data values. If \( n = |Q| \) is the length of the sequence, then \( Q[0], \ldots, Q[n - 1] \) are the Data items of \( Q \). We think of \( Q[0] \) as being the “head” value of \( Q \) and \( Q[n - 1] \) as its tail value. We assume a bound \( K \) on the length of the queue, and so \( 0 \leq |Q| \leq K \). The queue is said to be empty when \( |Q| = 0 \) and full when

*mathematical models for concurrency 2014, Ben Gurion University of the Neguev.
$|Q| = K$. The concatenation operation on sequences is notated $\sim$. So if $P$ and $Q$ are two sequences of length $n = |P|$ and $m = |Q|$ then $P \sim Q$ is the sequence $R$ of length $n + m$ defined by the following equations.

$$R[i] = P[i] \text{ for } 0 \leq i < n \quad (1)$$
$$R[j] = Q[j - n] \text{ for } n \leq j < n + m. \quad (2)$$

When $d$ is some $Data$ value then $\langle d \rangle$ denotes the sequence of length 1 whose sole entry is $d$, and we can also write $Q \sim d$ (or $d \sim Q$) rather than $Q \sim \langle d \rangle$ (or $\langle d \rangle \sim Q$).

**Definition 1.1** Let $\tau = \langle e_0, e_1, \ldots \rangle$ be a sequence of events such that for every index $i$ of the sequence either $enq(e_i)$ or else $deq(e_i)$, and for every index $i$, $Val^*(e_i)$ is some $Data$ value or else the empty value $\emptyset$. Then we say that $\tau$ satisfies the “state-based” queue specifications for a queue of capacity $K$ if there are queue states $\text{pre}(e_i)$ and $\text{post}(e_i)$ for every index $i$ of the sequence such that the following hold.

1. $\text{pre}(e_0)$ is the empty sequence. For every $i$ $\text{pre}(e_i)$ and $\text{post}(e_i)$ are sequences of length $\leq K$ of $Data$ values.

2. For every index $i$ such that $i + 1$ is also an index of the sequence $\text{post}(e_i) = \text{pre}(e_{i+1})$.

3. If $deq(e_i)$ and $d = Val^*(e_i)$ is a $Data$ value, then $Q = \text{pre}(e_i)$ is nonempty and $Q = d \sim \text{post}(e_i)$.
   
   If $deq(e_i)$ and $Val^*(e_i) = \emptyset$, then $\text{pre}(e_i)$ is the empty sequence and $\text{post}(e_i)$ remains the empty sequence.

4. If $enq(e_i)$ and $d = Val^*(e_i)$, then $\text{pre}(e_i) \sim d = \text{post}(e_i)$.

A clear advantage of the state based queue specification is its simplicity. The regime of a queue is that items are added at the tail and removed from its head.
2 A bounded buffer algorithm

We assume two processes a Producer and a Consumer that coordinate their actions with a bounded buffer $B$ that can contain up-to $k \geq 1$ items. The buffer is full when it contains $k$ items, and it is empty when it contains no items. The Producer loads an item only when the buffer is not full, and the Consumer unloads an item only when it is non-empty. The role of the algorithm is to ensure that this is indeed the case. We assume that the buffer stores its items in an array $B[0, \ldots, k-1]$ of items. We say that $B[i]$ is the $i$-th entry of $B$. The algorithm uses two registers $s$ and $r$ that hold natural numbers. Register $s$ is written by the Producer process. This variable records the number of items produced so far by the Producer, and $s \mod k = i$ is the index of the entry $B[i]$ into which Producer is going to deposit its $s$-th item. Register $r$ records the number of items that were removed by the Consumer, and $r \mod k = j$ is the index of the entry $B[j]$ from which Consumer is going to unload its $r$-th item. Initially $r = s = 0$.

The Bounded Buffer algorithm is in figure 1.

The state variables include, in addition to $r$ and $s$, two program counter variables $PC_p$ and $PC_c$ which are the program counters of the Producer and Consumer processes respectively. The value of the program counter indicates which instruction is going to be executed next.

To sum-up, the set of state variables is

$$V = \{s,r,PC_p,PC_c,B\}.$$ 

The types of the variables is as follows. Variables $s$ and $r$ take values that are natural numbers. Variable $PC_p$ takes values in $\{1_p, \ldots, 4_p\}$, and $PC_c$ in $\{1_c, \ldots, 4_c\}$. Variable $B$ is an array of length $k$ (with entries $B[0], \ldots, B[k-1]$ of Data type). A state is thus a function that assigns to each of the variables in $V$ a value in its type.

If $\varphi$ is any statement about the state variables and $S$ is any state, then the satisfaction relation $S \models \varphi$ says that $\varphi$ holds in $S$. For example, $S \models s-r < k$

---

1 The algorithm appears in Abraham, Models for Concurrency Gordon and Breach 1999.
2 A similar algorithm was discussed by Lamport in “Proving the correctness of Multiprocess Programms, IEEE Transactions on Software Engineering, 1977.
3 Type restrictions promote understanding of the algorithm and hence they are very useful. It is part of the proof obligations to ensure that the variables are indeed in their types. For this reason, when defining steps, one must be very careful to note that the type restrictions are respected.
Producer

1. p.  if \( s - r = k \) goto 1;

2. p.  produce a Data value \( d \) and \( B[s \ (mod \ k)] := d; \)

3. p.  \( s := s + 1; \)

4. p.  goto 1p

Consumer

1. c.  if \( s = r \) goto 1c;

2. c.  consume \( B[r \ (mod \ k)]; \)

3. c.  \( r := r + 1; \)

4. c.  goto 1c;

Figure 1: The bounded concurrent buffer algorithm for two processes.

means that \( S(s) - S(r) < k \). States can thus be considered to be structures with a well defined satisfaction relation.

An initial state for the buffer algorithm is a state \( S \) such that \( S(s) = S(r) = 0 \), \( S(PC_p) = 1_p \) and \( S(PC_c) = 1_c \). And \( S(B) \) is an arbitrary array of length \( k \) of Data values.

A pair of states (over \( V \)) is called a “step” when it describes an atomic change of the system (a detailed definition is given below).

There are two kind of steps in our buffer system: steps of the Producer process and steps of the Consumer. Any instruction of the protocol corresponds to a set of steps: those steps that represent an execution of the instruction.

A history (for our protocol) is a sequence of states over \( V \),

\[
H = \langle S_i \mid i \in \omega \rangle,
\]

where \( \omega \) is the set of natural numbers (we assume for simplicity that \( H \) is infinite), such that \( S_0 \) is an initial state and for every \( i \in \omega \) there is an instruction \( R \) in the protocol (of one of the two processes) such that step \( \langle S_i, S_{i+1} \rangle \) is an execution of \( R \). A history contains steps by the two processes, and thus the execution is said to “interleave” the actions of the processes in a history.

The collection of all resulting histories describes the possible executions of the given protocol, and in order to prove that all protocol executions satisfy a certain property \( \varphi \) we have to prove that each of the resulting histories satisfies this property.
The steps of the buffer algorithm can be given names by reference to the change of the program counter. So, for example a \((1_p, 1_p)\) step is an execution of instruction \(1_p\) by the *Producer* process in which condition \((s - r = k)\) holds. And a \((1_p, 2_p)\) step is an execution of this instruction in which the condition does not hold. A \((2_p, 3_p)\) step is said to be a loading step, and a \((2_c, 3_c)\) step is an unloading step.

**Exercise 2.1** Write in details the steps of the buffer algorithm.

As an example we bring the following.

**Definition 2.2** A \((2_p, 3_p)\) step is a pair of states \((S, T)\) such that the following holds.

1. \(S(\text{PC}_p) = 2_p\), \(T(\text{PC}_p) = 3_p\).
2. For every buffer index \(0 \leq i < k\), if \(i \neq S(s) \mod k\) then \(T(B)[i] = S(B)[i]\). For \(i = S(s) \mod k\), \(T(B)[i]\) is an arbitrary message. Every state variable other than \(B\) and \(\text{PC}_p\) has the same value in \(T\) as in \(S\).

If \(e = (S, T)\) is a \((2_p, 3_p)\) step, then we say that \(e\) is an enqueue step (we can write \(\text{enq}(e)\)) and we define \(\text{Val}(e) = T(B)[T(S(s) \mod k)]\) as the value of this step.\(^3\)

Similarly, if \(e = (S, T)\) is a \((2_c, 3_c)\) step, then we say that \(e\) is a dequeue step (we can write \(\text{deq}(e)\)) and we define \(\text{Val}(e) = T(B)[T(r) \mod k]\) as the value of this step.

### 2.1 The assertional argument

Recall that an assertion is a statement about states, and an invariant is an assertion \(\varphi\) which holds in every initial state and which has the following property. If \((S, T)\) is any step such that \(\varphi\) holds in \(S\), then \(\varphi\) holds in \(T\) as well.

We have proved the following simple but important theorem. If \(\varphi\) is an invariant and \(R\) a history as above, then \(\varphi\) holds in every state of \(R\).

**Exercise 2.3** Prove that the following conjunction of \(A\), \(B\), and \(C\) is an invariant.

\(^3\)This is not the \(\text{Val}^*\) function defined in Section 1.
A. Every state variable is in its type. In particular $B$ is an array of Data values.

B. $0 \leq s - r \leq k$.

C. If $PC_p = 2_p, 3_p$ then $s - r < k$. If $PC_c = 2_c, 3_c$ then $0 < s - r$.

Directions. First prove that it holds in any initial state. Then consider in turn each of the following ten kinds of steps. For each kind and for every step $st = (S,T)$ of that kind, assume that $S \models A \land B \land C$, and then prove that $T \models X$ for each of $X = A, B, C$.

1. $st$ is some $(1_p, 1_p)$ step.
2. $st$ is some $(1_p, 2_p)$ step.
3. $st$ is some $(2_p, 3_p)$ step.
4. $st$ is some $(3_p, 4_p)$ step.
5. $st$ is some $(4_p, 1_p)$ step.
6. $st$ is some $(1_c, 1_c)$ step.
7. $st$ is some $(1_c, 2_c)$ step.
8. $st$ is some $(2_c, 3_c)$ step.
9. $st$ is some $(3_c, 4_c)$ step.
10. $st$ is some $(4_c, 1_c)$ step.

3 A queue is implemented

Exercise 2.3 does not yet establish that our algorithm implements a queue. In order to prove this main result we need to introduce a new variable $Q$ whose values are Data sequences. This variable is not to be found in the algorithm text of Figure 1—it is an artificially introduced data structure which behaves as a queue should behave (that is as specified in Section 1). The correctness proof then has to show that the values of the dequeue steps in a history are equal to the values of the corresponding dequeues of the hypothetical
queue $Q$. So now we will have two functions that give values to enqueue and
dequeue steps. The $Val$ function which was defined above in Definition 2.2,
and the $Val^*$ which is defined in Section 1 and the correctness proof has to
establish that these two functions are equal. This is going to be clearer with
the additional details that follow.

We add a new state variable $Q$ whose values are sequences of Data items.
So now the set of variables is redefined as:

$$V = \{s, r, PC_p, PC_c, B, Q\}.$$ 

An initial state is a state $S$ such that $S(s) = S(r) = 0$, and $S(PC_p) = 1_p$, $S(PC_c) = 1_c$, and $S(B)$ is an array of length $k$ of Data values, as before,
and the additional requirement now is that $S(Q)$ is the empty sequence $\emptyset$. That is, the queue is initially empty.

Steps are defined as before but with additional requirements for the en-
dequeue steps (that is, the $(2_p, 3_p)$ and $(2_c, 3_c)$ steps).

1. **Enqueue steps.** The additional requirement for a $(2_p, 3_p)$ step $(S, T)$
is that $T(Q) = S(Q) \setminus \langle T(d) \rangle$. That is, the queue $Q$ is extended with
the newly produced Data value added to its tail. And if $e = (S, T)$ is
some $(2_p, 3_p)$ step then in addition to the $Val$ function that was defined
in 2.2, we define $Val^*(e) = T(d)$. (So $Val^*(e) = Val(e)$, and the two
functions agree by definition on the enqueue steps.)

2. **Dequeue steps.** If $e = (S, T)$ is a $(2_c, 3_c)$ step, then in addition to the
requirements of Definition 2.2 we ask the following. If $S(Q) = \emptyset$, then
$T(Q) = \emptyset$ and we define $Val^*(e) = \emptyset$. That is, if the queue is empty
then the queue remains empty in a dequeue step and the value* of the
step is the empty value $\emptyset$. But if $S(Q) \neq \emptyset$ and $x$ is the head value
of the queue (that is $x = S(Q)(0)$) then we define $Val^*(e) = x$ and
$\langle x \rangle \setminus S(Q) = T(Q)$ (or in simple words, $T(Q)$ is obtained from $S(Q)$ by
removing the first item $x$ of $S(Q)$).

Now consider an execution history of our buffer algorithm, $R = \langle S_i \mid i \in \omega \rangle$,
where the states $S_i$ are states with the added variable $Q$. If we prove that
the length of $Q$ never exceed $k$ (that is $|S_i(Q)| \leq k$ for every index $i$) and
that for every dequeue step $e_i = (S_i, S_{i+1})$ $Val^*(e_i) = Val(e_i)$ then we have
proven that the algorithm implements a queue. With this aim in mind, we
first introduce definitions and notations.
The set \([0, \ldots, k) = \{i \mid 0 \leq i < k\}\) is supplied with the \(\mod k\) successor function that takes \(i\) to \(i + 1 \mod k\). For indices \(i, j \in [0, \ldots, k)\), we denote with \([i, \cdot, j)\) the sequence that goes in a circular way from \(i\) to \(j - 1 \mod k\). So, for example, if \(k = 7\) then the mod 7 successor of \(6\) is 0, \([3, \cdot, 6)_7\) is the sequence \(\langle 3, 4, 5, \rangle\), and \([6, \cdot, 3)_7\) is the sequence \(\langle 6, 0, 1, 2\rangle\). Note also that \([5, \cdot, 0)\) is the sequence \(\langle 5, 6\rangle\). The formal definition is the following.

1. If \(i = j\) then \([i, \cdot, j)_k = \emptyset\).
2. If \(i < j\) then \([i, \cdot, j)_k\) is the sequence of length \(j - i\) that enumerates the interval \([i, \ldots, j)\). That is, \([i, \cdot, j)_k = \langle i, \ldots, j - 1\rangle\).
3. If \(i > j\) then \([i, \cdot, j)_k = \langle i, \ldots, k - 1\rangle \mathbin{\mod} \langle 0, \ldots, j - 1 \mod k\rangle\) (a sequence of length \(j - i \mod k\), that is \(k - (i - j)\)).

\textbf{Lemma 3.1} Let \(i, j \in [0, \ldots, k)\) be indices. If \([i, \cdot, j)_k\) is of positive length \(\ell > 0\), then \([i + 1 \mod k, \cdot, j)_k\) is the sequence of length \(\ell - 1\) obtained by removing the first member, \(i\), of \([i, \cdot, j)_k\).

If \([i, \cdot, j)_k\) is of length \(\ell < k\), then \([i, \cdot, j + 1 \mod k)_k\) is the sequence of length \(\ell + 1\) obtained by adding \(j\) to \([i, \cdot, j)_k\). That is \([i, \cdot, j + 1 \mod k)_k = [i \cdot j)\langle j\rangle\).

For example, for \(k = 7\), \([6, \cdot, 0)_7 = \langle 6\rangle\) and \([6 + 1 \mod 7, \cdot, 0)_7 = \emptyset\). And \([3, \cdot, 6 + 1 \mod 7)_7 = \langle 3, 4, 5, 6\rangle = [3, \cdot, 6)_7 \langle 6\rangle\).

If \(B\) is an array of length \(k\) of Data values and \(0 \leq i, j < k\) are indices, then \(B \circ [i, \cdot, j)_k\) denotes the sequence of Data values obtained as the composition of the functions \(B\) and \([i, \cdot, j)_k\). Following an intuitive image, we can say that \(B \circ [i, \cdot, j)_k\) is the \emph{arc} of \(B\) that extends from index \(i\) and up-to (but not including) index \(j\). In details, the definition is the following.

1. If \(i = j\) then \(B \circ [i, \cdot, j)_k\) is the empty sequence.
2. If \(i < j\), then \(B \circ [i, \cdot, j)_k\) is the sequence (of length \(j - i\))
   \[B[i], \ldots, B[j - 1],\]
   (or, in another notation the sequence \(\langle B[\ell] \mid i \leq \ell < j\rangle\)).
3. If \(i > j\), then \(B \circ [i, \cdot, j)_k\) is the sequence (of length \(k - (i - j)\))
   \[\langle B[i], \ldots, B[k - 1], B[0], \ldots, B[j - 1]\rangle.\]
For example, suppose that $B$ is the sequence (array) of length 7: $B = \langle a, b, c, d, e, f, g \rangle$. So $B[0] = a, \ldots, B[6] = g$. Then $B \circ [3, \ldots, 6]_7 = \langle d, e, f \rangle$ and $B \circ [6, \ldots, 3] = \langle g, a, b, c \rangle$.

The following is an immediate consequence of Lemma 3.1.

**Lemma 3.2** If $R = B \circ [i, \ldots, j]_k$ is of positive length $\ell > 0$, and $R' = B \circ [i + 1 \mod k, \ldots, j]_k$, then $R'$ is the result of the dequeue operation on $R$, and $B[i]$ is the value that this dequeue operation returns.

Likewise, if $R = B \circ [i, \ldots, j]_k$ is of length $\ell < k$, if $B'$ is an array of length $k$ with the same entries as $B$ except possibly that $d = B'[j]$ is some given Data value and we define $R' = B' \circ [i, \ldots, j + 1 \mod k]$, then $R'$ is obtained by enqueueing the value $d$ to $R$.

For example, if $B$ is the array of length 7 as above, and $R = B \circ [3, \ldots, 6]_7 = \langle d, e, f \rangle$, then after executing $B[6] := h, B \circ [3, \ldots, 0]_7$ is equal to $\langle e, f, g, h \rangle$.

In the following theorem we consider equation $(B \circ [m \mod k, \ldots, n \mod k])_k = Q$; using the arc imagery we can say that it holds in state $S$ iff the sequence $S(Q)$ is equal to the arc of $S(B)$ that extends from index $S(i)$ and up-to (but not including) index $S(j)$.

**Theorem 3.3** Let $m$ and $n$ be natural numbers and $\varphi(m, n)$ be the statement

$$(B \circ [m \mod k, \ldots, n \mod k])_k = Q. \quad (3)$$

The following conjunction $\alpha = E_1 \land E_2 \land E_3 \land E_4$ is an invariant of our buffer algorithm.

$E_1$ If $PC_p \neq 3_p$ and $PC_c \neq 3_c$ then $\varphi(r, s)$.

$E_2$ If $PC_p = 3_p$ and $PC_c = 3_c$ then $\varphi(r + 1, s + 1)$.

$E_3$ If $PC_p \neq 3_p$ and $PC_c = 3_c$ then $\varphi(r + 1, s)$.

$E_4$ If $PC_p = 3_p$ and $PC_c \neq 3_c$ then $\varphi(r, s + 1)$.

It is convenient to use the following notation. If $\tau$ is any expression that involves state variables, if $S$ is any state, then $\tau^S$ denotes the value of $\tau$ as it is evaluated in state $S$. For example, expression $B \circ [r + 1 \mod k, \ldots, s \mod k]_k$ involves variables $B, r$ and $s$, and so

$$(B \circ [r + 1 \mod k, \ldots, s \mod k])^S = S(B) \circ [S(r) + 1 \mod k, \ldots, S(s) \mod k].$$
We sketch some steps of the proof and let the reader complete the rest. We first have to prove that our invariant \( \alpha \) holds in every initial state. If \( S \) is an initial state then \( S |\!\!Q = \emptyset = \emptyset \wedge r = 0 \wedge s = 0 \). But \( [0, \cdots , 0)_k = \emptyset \) and hence \( \varphi(0,0) \) is obvious. (Since the antecedents\(^4\) of \( E_2, E_3 \) and \( E_4 \) are false in the initial state \( S \), these statements are true.)

Now we have to deal with each type of step. As an example we take \((S,T)\) to be some \((2p,3p)\) step and assume that \( \alpha \) holds in \( S \). We have to prove that \( \alpha \) holds in \( T \) as well. Recall the definition of \((2p,3p)\) steps for states with the additional variable \( Q \): a \((2p,3p)\) step is a pair of states \((S,T)\) such that the following holds.

1. \( S(PC_p) = 2p, T(PC_p) = 3p, \)
2. for every buffer index \( 0 \leq i < k \), if \( i \neq S(s) \mod k \) then \( T(B)[i] = S(B)[i] \). For \( i = S(s) \mod k \), \( T(B)[i] = T(d) \) is an arbitrary Data value. Every state variable other than \( B, d \) and \( PC_p \) has the same value in \( T \) as in \( S \).
3. \( T(Q) = S(Q) \land (T(d)) \).

Since \( T \models PC_c = 3c \) and \( T \models PC_c \neq 3c \).

1. \( T \models PC_c = 3c \). In this case the antecedent of \( E_2 \) holds and we have to prove that \( T \models \varphi(r+1,s+1) \) (note that statements \( E_1, E_3, E_4 \) hold trivially in \( T \) since their antecedents are false). In state \( S \), \( PC_p = 2p \) and \( PC_c = 3c \) (as the value of \( PC_c \) does not change in this step). As \( \alpha \) holds in \( S \), \( S \models E_3 \) and so \( S \models \varphi(r+1,s) \). That is

\[
S \models (B \circ [r+1 \mod k, \cdots , s \mod k]) = Q.
\]

And we have to prove that

\[
T \models (B \circ [r+1 \mod k, \cdots , s + 1 \mod k]) = Q. \tag{4}
\]

How \( Q \) is changed in this step? \( T(Q) = S(Q) \land (T(d)) \) by definition of this step.

How \( B \) is changed in this step? If \( 0 \leq i < k \) and \( i \neq T(s) \mod k \) then \( T(B)[i] = S(B)[i] \), and, for \( i = T(s) \mod k \), \( T(B)[i] = T(d) \). Note that

\(^4\)In an implication \( X \rightarrow Y \), \( X \) is the antecedent and \( Y \) the subsequent of the implication.
by Exercise 2.3 (itec C) \( S \models 0 < s - r < k \), and hence \( T \models 0 < s - r < k \) (as these variables do not change). So \( T \models 0 \leq s - (r + 1) < k \). Thus the length of \((r + 1 \mod k, \ldots, s \mod k)_k^S\) is \( s - (r + 1) < k \), and (the second statement of) Lemma 3.2 says then that \((B \circ [r + 1 \mod k, \ldots, s + 1 \mod k])_T\) is obtained from \((B \circ [r + 1 \mod k, \ldots, s \mod k])_S\) by enqueing \(T(d)\). Thus proving (4).

2. The second case is when \( T \models PC_c \neq 3c \). Exercise: complete this case.

For the next example we consider a \((3p, 4p)\) step \((S, T)\). That is, and execution of instruction \( s := s + 1 \). Suppose that \( \alpha \) holds in \( S \) and we have to prove that it holds in \( T \) as well. Again there are two cases, depending on whether \( T \models PC_c = 3c \) or not. Suppose for example that in state \( T \) \( PC_c \neq 3c \). Hence our obligation is to prove that \( E_1 \) holds in \( T \) since the antecedent of all other conjuncts of \( \alpha \) are false in \( T \). in state \( T \) \( PC_c \neq 3c \), it follows that \( S \models PC_c \neq 3c \wedge PC_p = 3p \). So by \( E_4 \), \( \varphi(r, s + 1) \) holds in \( S \). Statement \( \varphi(r, s + 1) \) involves variables \( B, Q, r \) which do not change in this step, and variable \( s \) which does, as \( T(s) = S(s) + 1 \). Hence \( T \) satisfies \( \varphi(r, s) \) as required.

We ask the interested student to complete the proof that \( \alpha \) is an invariant, and we continue with the correctness proof assuming that we already know that \( \alpha \) holds in every state of the given history.

Recall that the main issue in proving that the algorithm implements a queue is in proving that for every dequeue step \( e \) in the history \( Val(e) = Val^*(e) \). So let \( e = (S, T) \) be one of the dequeue steps in the history (i.e. a \((2c, 3c)\) step). By Definition 2.2,

\[
Val(e) = T(B)[T(r) \mod k]
\]

is the value of this step.

In state \( S \), \( PC_c = 2c \), and hence (by Exercise 2.3) \( 0 < s - r \) holds in \( S \). If \( S(PC_p) = 3p \) then \( \varphi(r, s + 1) \) holds in \( S \) by \( E_4 \), and if \( S(PC_p) \neq 3p \) then \( \varphi(r, s) \) holds by \( E_1 \). Hence \( S(Q) \) is either \((B \circ [r \mod k, \ldots, s + 1 \mod k])_S \) or \((B \circ [r \mod k, \ldots, s \mod k])_S \). In both cases, since \( r < s \) in \( S \), these arcs are nonempty, and hence \( S(Q) \) is non-empty, and the first member of \( S(Q) \) (the one at its tail) is defined to be \( Val^*(e) \). So \( Val^*(e) = Val(e) \) follows.