Formal Specification of Bounded Buffer using Stream Functions

Gongzhu Hu
Department of Computer Science
Central Michigan University
Mount Pleasant, Michigan 48859, USA
hu1g@cmich.edu

Abstract

Formal specifications of software components are critical to software development. Several types of formal or semi-formal methods are commonly used for software specification, such as specification languages, graphic diagrams, algebraic descriptions, and stream functions. Each of these methods addresses the specification problem from a different viewpoint and has its own strengths and weaknesses. In this paper, we use the stream function approach to formally specify a particular software component, bounded buffer. Based on the specification, a state transition machine is built as an implementation of the bounded queue.

Keywords formal specification, stream function, bounded queue, serialization, state transition machine.

1 Introduction

A software application is normally composed of modules, or components, that are self-contained with clearly defined functionalities. The components interact with the environment through input/output parameters. These functionalities are described in components’ specifications that should be precise and formal. Various types of formal methods are available for specification of software components, such as specification languages (e.g. Object-Z [12]), graphical languages (e.g. Unified Modeling Language (UML) [7]), and algebraic descriptions [1]. Most of the specification and graphic languages either are semi-formal, or using pre- and post-conditions.

The stream function [13] approach treats a software component as a black-box with its functionalities defined as a function (mapping) from input streams to output streams. It is a formal approach and has been successfully applied to various software components.

In this paper, we give a formal specification of the software component queue, and bounded queue in particular, as a function that maps an input stream to an output stream. The input stream is a sequence of serialized enter-queue and de-queue commands, and the output stream is a sequence of data items. The serialization of input streams is in fact an abstraction of the mutually exclusive access to the critical region. From the specification, we create a finite state transition machine to implement the bounded queue.

Although the idea of describing a software component as a mapping from input stream to output stream is not new, actual description for specific types of components has not been sufficiently addressed in the literature, mainly because the vast diverse properties of the components. Specification of each type of software components is by itself a research topic. In particular, specifications of queue as a stream function has been given before such as in [6], but no work has been done considering a bounded queue with parallel accesses, as in the producer-consumer problem. This paper addresses this problem and presents a way in which the specification can be implemented using a state transition machine.

2 Component Specification and Stream Function

Using stream function in the specification of an interactive component is to define the behavior of the component as a mapping from the input streams to the component and the output stream produced by the component.

Given an alphabet \( \mathcal{A} \), the set \( \mathcal{A}^* \) comprises all streams \( A = \langle a_1, a_2, \ldots, a_k \rangle \) of length \( |A| = k \) with elements \( a_i \in \mathcal{A} \) (\( i \in [1, k], k \geq 0 \)). Several operations can be applied to communication streams, one of the very basic is concatenation that is mostly used in this paper. The concatenation operator, \( \& \), is defined as

\[ \& : \mathcal{A}^* \times \mathcal{A}^* \rightarrow \mathcal{A}^* \]

of two streams \( A = \langle a_1, a_2, \ldots, a_k \rangle \) and \( B = \langle b_1, b_2, \ldots, b_l \rangle \) and yields the stream \( A \& B = \langle a_1, a_2, \ldots, a_k, b_1, b_2, \ldots, b_l \rangle \).
A stream function \( f : \mathcal{I}^* \rightarrow \mathcal{O}^* \) maps an input stream to an output stream. We may also use Input and Output as the input and output alphabet, respectively. Hence, a stream function may also be denoted as \( f : \text{Input}^* \rightarrow \text{Output}^* \), or simply \( f : I^* \rightarrow O^* \).

We adopt the notion similar to [5], using capital letters like \( X, Y \) to represent a stream (e.g. \( X = \langle x_1, \cdots, x_n \rangle \)). The symbol \& is used as the concatenation operator on two streams. That is, for \( X \) and \( Y \) in \( \mathcal{I}^* \), the concatenation of \( X \) and \( Y \) is \( X \& Y = \langle x_1, \cdots, x_n, y_1, \cdots, y_m \rangle \).

### 3 Unbounded Queue

In our formal specification of queue, let \( \mathcal{D} \) be the set of data values storable in a queue, \( \text{enq}(d) \) be an enter-queue command, \( \text{deq} \) be a de-queue command, and \( \text{exc} \) \( \notin \mathcal{D} \) be a message representing an exception. An input to the queue is

\[
I = \text{enq}(\mathcal{D}) \cup \{ \text{deq} \}
\]

and an output of the queue is

\[
O = \mathcal{D} \cup \{ \text{exc} \}
\]

The queue is a mapping

\[
\text{queue} : I^* \rightarrow O^*
\]

We use \( \text{Enq} \) and \( \text{Deq} \) to represent zero or more \( \text{enq} \) and \( \text{deq} \) commands, respectively:

\[
\text{Enq} \in \text{enq}(\mathcal{D})^*, \text{Deq} \in \text{deq}^*
\]

We also use \( \text{enq}^m \) to denote \( m \) \( \text{enq} \) commands and \( \text{deq}^m \) to denote \( m \) \( \text{deq} \) commands.

We first review the definition of an unbounded queue. As mentioned before that the only case an exception arises for an unbounded queue is a de-queue command on an empty queue. There are several ways to handle such an exception. The simplest is to treat it as an error.

The regular behavior of an unbounded queue is defined by the following equations.

\[
\text{queue}(\text{Enq}) = \langle \rangle \quad (1)
\]

\[
\text{queue}(\langle \text{deq} \rangle \& X) = \langle \text{exc} \rangle \& \text{queue}(X) \quad (2)
\]

\[
\text{queue}(\langle \text{enq}(d) \rangle \& \text{Enq} \& \langle \text{deq} \rangle \& X) = \langle d \rangle \& \text{queue}(\text{Enq} \& X) \quad (3)
\]

Equation (3) defines the normal operation of the queue. The exception raised upon a \( \text{deq} \) command on an empty queue is specified in Equation (2). How to handle such an exception is up to the application that receives the output from a queue. It may be considered an error, or simply ignored as no-operation, or the \( \text{deq} \) command is fed back to the input stream.

For some applications, it is desirable to have the queue to manage the exception. There are several “irregular” behaviors of a queue regarding how the exceptions are handled, similar to the specification for irregular stacks in [4], including fault sensitive, fault tolerance, and fault correcting.

### 4 Bounded Queue

For bounded queue, most of the above specification will still hold but we must also consider the case when an \( \text{enq} \) command in the input stream is encountered when the queue is full. That is, the mapping \( \text{queue} \) not only depends on the input stream but also depends the number of data values that are in the queue but have not been de-queued. This is the storage limitation the producer-consumer problem is supposed to deal with.

#### 4.1 Specification

Let \( \mathbb{N} \) be the set of non-negative integers, \( N \in \mathbb{N} \) be the size limit of the queue, and \( k \in \mathbb{N}, k \leq N \) be the size of the queue, i.e. the number of data items in the queue. Bounded queue \( bqueue \) is a mapping

\[
bqueue : I^* \times \mathbb{N} \rightarrow O^* \times \mathbb{N}
\]

that maps an input stream to an output stream depending on the size of the queue, and at the same time the queue size is also changed.

For bounded queue, we need to make a distinction between infinite input stream and finite input stream. In the case that the input stream is finite (e.g. the application using the bounded queue eventually terminates), we should make an assumption, without loss of generality, that the “excessive” \( \text{enq}(d) \) commands (those commands after the queue is full) are eventually de-queued. That is \( (m-n) \leq N \), where \( m \) and \( n \) are the total number of \( \text{enq}(d) \) commands and the total number of \( \text{deq} \) commands, respectively, when the application terminates, regardless how large the input stream is. Obviously, \( m+n = M \) where \( M \) is the length of the input stream upon the termination of the application.

The bounded queue mapping functions are defined below.

#### Enter-queue

An \( \text{enq}(d) \) command on a non-full queue produces no output, but enters \( d \) into the queue so that the size of the queue is increased by one, as specified by Equation (4). The
queue can take a sequence of \textit{enq}(d) commands as long as it does not cause an overflow, shown in Equation (5).  

\begin{align}
\text{bqueue}((\text{enq}(d)) & \& X, k) = \\
\text{bqueue}(X, k + 1), & 0 \leq k < N \quad (4) \\
\text{bqueue}(\text{enq}, t) = ((), k + 1), & 1 \leq k, k + 1 \leq N \quad (5)
\end{align}

When the queue is full, additional \textit{enq}(d) commands are suspended until a \textit{deq} command removes a data item from the queue to the output. This situation is defined in Equation (6).

\begin{align}
\text{bqueue}(\text{Enq} & \& (\text{deq}) & \& X, N) = \\
\text{bqueue}(\langle \text{deq} \rangle & \& \text{Enq} & \& X, N) \quad (6)
\end{align}

\section*{De-queue}

The specification for bounded queue on \textit{deq} command is similar to unbounded queue, except that we also need to specify the queue size in addition to the input and output streams in the mapping. Because a \textit{deq} command may produce an output, we adapt the notion 

\begin{align}
\langle d \rangle & \& \text{bqueue}(\ldots)
\end{align}

to mean that \(d\) is concatenated to the output stream part of \textit{bqueue}(\ldots), while the queue length is not part of the concatenation.

Equation (7) states that the first \textit{deq} command consumes the first item \(d\) in the queue and produces \(d\) to the output stream, and leaves the queue length unchanged.

\begin{align}
\text{bqueue}(\langle \text{enq}(d) \rangle & \& \text{Enq} & \& (\text{deq}) & \& X, k) = \\
\langle d \rangle & \& \text{bqueue}(\text{Enq} & \& X, k), & 0 \leq k \leq N \quad (7)
\end{align}

As stated before that the total number of \textit{deq} commands in a finite input stream cannot be more than the total number of \textit{enq}(d) commands, each \textit{deq} will have a matching \textit{enq}(d) somewhere in the input stream. A \textit{deq} command, therefore, will not change the length of the queue.

\textit{Deq} commands on an empty queue are suspended, just as in the case for unbounded queue, shown in Equation (8). The output stream part of the \textit{bqueue} mapping is a concatenation of \(\langle d \rangle\) and a mapping function of fault correcting queue.

\begin{align}
\text{bqueue}(\langle \text{deq} \rangle & \& \text{Deq} & \& \langle \text{enq}(d) \rangle & \& X, 0) = \\
\langle d \rangle & \& \text{bqueue}(\text{Deq} & \& X, 0) \quad (8)
\end{align}

\subsection*{4.2 Parallel Access by Serialization}

In the bounded queue problem, the producer and the consumer interact with the queue asynchronously and independently. The specification of bounded queue given in the previous sections provide all the functionalities of the queue dealing with the \textit{enq} and \textit{deq} operations, except the issue of parallel access of the queue.

The approach we adopt here is a serialization method similar to what is used for concurrency control of database transactions. Here we can treat the producer and consumer as two parallel transactions and the \textit{enq} and \textit{deq} command are write operations to the shared queue. To be more specific, let the producer “transaction” be a sequence of \(m\) \textit{enq} commands:

\begin{align}
P = (\text{enq}(d_1), \text{enq}(d_2), \ldots, \text{enq}(d_m))
\end{align}

and the consumer “transaction” be a sequence of \(n\) \textit{deq} commands:

\begin{align}
C = (\text{deq}_1, \text{deq}_2, \ldots, \text{deq}_n)
\end{align}

The data item \(d_i\) entered by the \(i\)-th \textit{enq} command is to be de-queued only by the \(i\)-th \textit{deq} command. That is, we only need to make sure \textit{enq}(d_i) and \textit{deq}_i are in the right order. Ideally, the right order is \textit{enq}(d_i) goes before \textit{deq}_i so that both commands can be carried out successfully at the time they are encountered. If they are out of order, the \textit{deq}_i command is suspended, which is exactly what is specified in the bounded queue specification. Hence, the serialization of the commands becomes very simple: an arbitrary mix of the commands is acceptable as long as the ordering of \textit{enq}(d_i) and the ordering of \textit{deq}_i are preserved:

\begin{align}
t(\text{enq}(d_i)) < t(\text{enq}(d_j)), & \forall i, j, i < j \quad (9) \\
t(\text{deq}_i) < t(\text{deq}_j), & \forall i, j, i < j \quad (10)
\end{align}

where \(t(x)\) is the time stamp of \(x\).

A serialization of producer and consumer transactions can be depicted in Figure 1. The two original parallel transactions are shown in (a) and a possible serialization of the commands is shown in (b).

\section*{5 Implementation}

Like any software component specifications, the specification of the bounded queue given above is from an abstract point of view. It is independent of the way the queue is implemented. Here we offer one possible implementation using a state transition machine. A state transition machine for our bounded queue can be derived from the specification (Equation (4) – (8)).

\subsection*{5.1 State Transition Machine}

A state transition machine with input and output is a 6-tuple

\begin{align}
M = (Q, I, O, \delta, \phi, q_0)
\end{align}

where
We also use the abbreviation STM for state transition machine.

5.2 States in the STM for Bounded Queue

Recall that the bounded queue is defined as a mapping

\[ bqueue : I^* \times \mathbb{N} \rightarrow O^* \times \mathbb{N} \]

The output depends not only the input stream but also the size of the queue, and so changes in the process. Since the \( enq(d) \) and \( deq \) commands may be postponed when the queue is full or empty, we also need to include information about the postponed commands in a state. Because all \( deq \) commands are the same (without parameter), we can simply use an integer to represent the number of postponed \( deq \) commands. The \( enq(d) \) commands, however, are different based on the parameter \( d \) and hence a state should include the “buffered” data \( d \). This leads to the definition of a state as

\[ s = [D, k, B, c] \in Q \]

where \( D \in Data^* \) is the contents of the queue, \( k \in \mathbb{N} \) is the length of the queue, \( B \in Data^* \) is the data to be entered but postponed, and \( c \) is the count of postponed \( deq \) commands.

Note that the following are true regarding the state of bounded queue:

\[ c > 0 \quad \Rightarrow \quad D = \langle \rangle \land k = 0 \land B = \langle \rangle \] (11)
\[ B \neq \langle \rangle \quad \Rightarrow \quad k = N \land c = 0 \] (12)

Equation (11) says that if there are delayed \( deq \) commands, the queue must be empty and there are no delayed \( enq(d) \) commands. Equation (12) says that if there are delayed \( enq(d) \) commands, the queue is full and there are no delayed \( deq \) commands.

The initial state is an empty queue with no delayed \( enq(d) \) or \( deq \) commands:

\[ q_0 = [\langle \rangle, 0, \langle \rangle, 0] \]

5.3 State Transition and Output Functions

We describe the state transition function \( \delta \) and output function \( \phi \) based on the next command in the input being an \( enq(d) \) or a \( deq \). These functions are constructed according to the formal specifications given before.

**Transition on** \( enq(d) \)

An \( enq(d) \) command on a non-full queue of size \( 0 \leq k < N \) enters \( d \) into the queue and does not produce output if no \( deq \) commands were delayed:

\[ \delta([D,k,\langle \rangle,0],enq(d)) = [D \& d,k+1,\langle \rangle,0] \] (13)
\[ \phi([D,k,\langle \rangle,0],enq(d)) = \langle \rangle \] (14)

Note that these two functions are the direct result from the specification Equations (4) and (5).

However, if there were delayed \( deq \) commands (and hence the queue is empty), an \( enq(d) \) command will cause one of the \( deq \) commands be be processed and produce \( d \) as output, according to the specification Equation (8). The transition and output functions are given below:

\[ \delta([\langle \rangle,0,\langle \rangle,0],enq(d)) = [\langle \rangle,0,\langle \rangle,0] \] (15)
\[ \phi([\langle \rangle,0,\langle \rangle,0],enq(d)) = \langle d \rangle \] (16)

Equation (6) in the specification says when the queue is full \( (k = N) \), an \( enq(d) \) command is delayed until a \( deq \) is encountered at a later time. Hence the transition function \( \delta \) buffers the data item \( d \) and the output function \( \phi \) produces no output:

\[ \delta([D,N,B,0],enq(d)) = [D,N,B \& d,0] \] (17)
\[ \phi([D,N,B,0],enq(d)) = \langle \rangle \] (18)
Table 1. State transition table \( (d,b \in \text{Data}, D,B \in \text{Data}^*, N = \text{queue size limit}) \)

<table>
<thead>
<tr>
<th>( q )</th>
<th>( Input )</th>
<th>( q' = \delta(q, Input) )</th>
<th>( Output = \phi(q, Input) )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>([D,k &lt; N, \emptyset, 0])</td>
<td>( \text{enq}(d) )</td>
<td>([D &amp; d, k &lt; N, \emptyset, 0])</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>([D,N, \emptyset, 0])</td>
<td>( \text{enq}(d) )</td>
<td>([D,N,d,0])</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>3</td>
<td>([D,N,B,0])</td>
<td>( \text{enq}(d) )</td>
<td>([D,N,B &amp; d,0])</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>4</td>
<td>([\emptyset,0,\emptyset,c])</td>
<td>( \text{enq}(d) )</td>
<td>([\emptyset,0,\emptyset,c-1])</td>
<td>( \langle d \rangle )</td>
</tr>
<tr>
<td>5</td>
<td>([\emptyset,0,\emptyset,1])</td>
<td>( \text{enq}(d) )</td>
<td>([\emptyset,0,\emptyset,0])</td>
<td>( \langle d \rangle )</td>
</tr>
<tr>
<td>6</td>
<td>([\langle d \rangle &amp; D,k &lt; N,\emptyset,0])</td>
<td>( \text{deq} )</td>
<td>([D,k-1,\emptyset,0])</td>
<td>( \langle d \rangle )</td>
</tr>
<tr>
<td>7</td>
<td>([\emptyset,0,\emptyset,0])</td>
<td>( \text{deq} )</td>
<td>([\emptyset,0,\emptyset,1])</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>8</td>
<td>([\emptyset,0,\emptyset,c])</td>
<td>( \text{deq} )</td>
<td>([\emptyset,0,\emptyset,c+1])</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>9</td>
<td>([\langle d \rangle &amp; D,N,b,0])</td>
<td>( \text{deq} )</td>
<td>([D &amp; (b),N,0,B,0])</td>
<td>( \langle d \rangle )</td>
</tr>
<tr>
<td>10</td>
<td>([\langle d \rangle &amp; D,N,b,0])</td>
<td>( \text{deq} )</td>
<td>([D &amp; (b),N,\emptyset,0])</td>
<td>( \langle d \rangle )</td>
</tr>
</tbody>
</table>

### Transition on \( \text{deq} \)

Equation (7) specifies the regular behavior upon a \( \text{deq} \) command on non-empty queue of length \( k \geq 1 \). Accordingly, the transition function \( \delta \) moves to a state in which the first item \( d \) is removed and the queue length is decreased by 1, and the function \( \phi \) produces \( d \) as the output:

\[
\begin{align*}
\delta([d \& D,k,\emptyset,0], \text{deq}) &= [D,k-1,\emptyset,0] \quad (19) \\
\phi([d \& D,k,\emptyset,0], \text{deq}) &= [d] \quad (20)
\end{align*}
\]

When the queue is empty \( (k = 0) \), the \( \text{deq} \) command is postponed as specified in Equation (8). In this case, the \( STM \) moves to a state where the number of postponed \( \text{enq} \) commands is increased by 1 and no output is produced:

\[
\begin{align*}
\delta([\emptyset,0,\emptyset,c], \text{deq}) &= [\emptyset,0,\emptyset,c+1] \quad (21) \\
\phi([\emptyset,0,\emptyset,c], \text{deq}) &= \emptyset \quad (22)
\end{align*}
\]

### Consolidated State Transition Machine

In the above state transition and output functions (13 – 22), we do not distinguish two states \( q_i = [D_i,k_i,B_i,c_i] \) and \( q_j = [D_j,k_j,B_j,c_j] \) if they differ only in certain internal values such as the length of the queue \( 0 \leq k < N \), or the number of delayed commands \( |B| \) or \( c \), otherwise the \( STM \) would have potentially infinite number of states. For example, \( q_i \) and \( q_j \) are the same state if one of the following holds:

- \( 0 < k_i < N, 0 < k_j < N \) (hence \( B_i = B_j = \emptyset \), and \( c_i = c_j = 0 \)) even \( k_i \neq k_j \). This is normal state.
- \( c_i > 0, c_j > 0 \) (hence \( D_i = D_j = \emptyset \), and \( k_i = k_j = 0 \)) even \( c_i \neq c_j \). This is underflow state.
- \( k_i = k_j = N \) and \( |B_i| > 0, |B_j| > 0 \) (hence \( c_i = c_j = 0 \)) even \( |B_i| \neq |B_j| \). This is overflow state.

And, we do not distinguish \( D_i = D \& (d) \) and \( D_j = (d) \& D \) simply because both are in \( \text{Data}^* \). Similar argument applies to \( B_i = B \& (b) \) and \( B_j = (b) \& B \).

The state transition functions of the consolidated \( STM \) are shown in Table 1, in which \( q' = \delta(q, Input) \) is the next state, and \( Output = \phi(q, Input) \) is the output. A comment is given in the table for each transition. It is clear that the \( STM \) is complete because it includes all possible situations of the bounded queue.

### Remarks

Let \( P \) be the data stream in the input stream from the producer, \( C \) be the input stream of the consumer, and \( O \) be the output stream. It is clear that the following invariants hold at any time \( i \):

\[
\begin{align*}
|P| &= |P_i| + |D_i| + |B_i| + |O_i| \quad (23) \\
|C| &= |C_i| + c + |O_i| \quad (24)
\end{align*}
\]

Invariant (23) says that at any time \( i \), the data items from the producer remain in the input stream or reside in the queue or stay in the buffer (postponed) or have gone to the output stream. The invariant (24) indicates that the \( \text{deq} \) commands from the consumer either remain in the input stream or are delayed or have produced output by de-queueing the queue. And, the ordering of the data items is preserved. Namely, \( P = O_i \& D_i \& B_i \& P_i \).

### 6 Related Work

A lot of work has been done on formal methods for software components and their designs [8, 10, 11]. As mentioned in the Introduction section, there are different ways to formally specify components and using stream function is one of them that defines an interactive component as a “black box” to map the input stream to output stream. Interactive model for system component specification have been studied and proposed in the literature, such as asynchronous unidirectional channels [3] and communication.
history model [9]. In some of these studies, interactive components are described using stream functions.

Study on specifications of queue as a mapping from input stream to output stream has been reported, such as [5, 6]. In these studies, the behaviors of the queue in both normal mode and error mode were considered, but only for unbounded queue without parallel input streams.

As an application using bounded queue, the producer-consumer problem is a classical problem found in almost any textbook about parallel processing [14]. Besides the traditional aspect of the producer-consumer problem, it was studied as an example at an abstract level in [2], in which the producer-consumer problem was an example to illustrate the abstract model describing the components by history assertions.

7 Conclusion

In this paper, we defined the stream functions for the specification of the bounded queue for parallel $\text{enq}(d)$ and $\text{deq}$ streams. The two parallel streams are serialized in such a way that the $\text{enq}(d)$ commands in the producer’s input stream keep their ordering unchanged in the serialization. An $\text{enq}(d)$ command is postponed when the queue is full, and a $\text{deq}$ command is delayed when the queue is empty. The postponed commands are immediately executed when the queue becomes non-full or non-empty.

A state transition machine was created as an implementation of the bounded queue based on the formal specification. Each state in the state transition machine contains the data stored in the queue, the queue length, buffered $\text{enq}(d)$ and $\text{deq}$ commands. The transition function defines the behavior of the queue in both normal mode and exception mode, and the output function defines the output stream.

Systematic approach to software component development is a much studied area and needs more research, particularly in formal methods for specification and implementation. The approach presented in this paper is such an effort. We are currently working on applying similar ideas to other software components and to incorporate time requirements to these components.

References


