Analysing Concurrent Queues Using CSP: Examining Java's *ConcurrentLinkedQueue*

Kevin Chalmers * and Jan Bækgaard Pedersen

Posted Date: 13 May 2025

doi: 10.20944/preprints202505.0924.v1

Keywords: CSP; Formal Verification; Wait Free Data Structures; Concurrent Queue

 doi:10.20944/preprints202505.0924.v1

Article

Analysing Concurrent Queues Using CSP: Examining Java's *ConcurrentLinkedQueue*

Kevin Chalmers [1,*,†] **and Jan Bækgaard Pedersen** [2,†]

[1] University of Roehampton
[2] University of Nevada Las Vegas
* Correspondence: Kevin.Chalmers@roehampton.ac.uk
† These authors contributed equally to this work.

Abstract: In this paper we examine the OpenJDK library implementation of the *ConcurrentLinkedQueue*. In particular, we use model checking to verify OpenJDK's *ConcurrentLinkedQueue* behaves as the algorithm upon which it claims to be implemented, Michael and Scott's fast, and practical non-blocking concurrent queue algorithm. In addition, we develop a simple concurrent queue specification in CSP and verify that Michael and Scott's algorithm meets this specification. We conclude that both the algorithm and the implementation are correct and that they both behave as our simpler concurrent queue specification, which we can utilise in place of either implementation in future investigations requiring a concurrent queue.

Keywords: CSP; formal verification; wait-free data structures; concurrent queue

1. Introduction and Motivation

Library use in modern programming is ubiquitous. Java, for example, has an extensible SDK, with Java 21 having 4387 classes. The source code forming the basis of these libraries are rarely externally scrutinised for formal correctness. We take for granted that a library shipped with an SDK will work as defined in all circumstances.

In this paper we examine a specific class in the OpenJDK: the `ConcurrentLinkedQueue` [1], that is based on work by Michael and Scott [1]. Our aim is to have a simple specification of a concurrent queue that we can use in other system models. The simpler specification allows us to reduce the state space we are exploring in models, thus speeding up overall model checking time.

This work can be viewed as an extension of Lowe's [2]. In their work they also consider the Michael and Scott algorithm although only consider a garbage collecter friendly version. We want to explore if the Java implementation operates as defined by Michael and Scott, and furthermore develop a simplified reusable specification of such behaviour.

Our work utilises Hoare's [3–6] Communicating Sequential Processes (CSP) to verify specification equivalence. We start by translating both the Java implementation and the algorithm from Michael and Scott's work [1] to CSP_M (a machine-readable form of CSP). For the Java code, we perform the following steps:

- First we extract the Java implementation of `ConcurrentLinkedQueue` from the OpenJDK.
- We change the object-oriented code to structured code to simplify translation to CSP.
- We decompose complex operations to further simplify translation to CSP.
- Finally, we translate the code to CSP.

We perform similar steps for Michael and Scott's algorithm. Once we have the CSP versions of both the original algorithm and the Java implementation, we proceed to show — using the FDR [7]

[1] We have taken the implementation from the OpenJDK source available at https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/concurrent/ConcurrentLinkedQueue.java

 doi:10.20944/preprints202505.0924.v1

refinement checking tool — that the Java implementation indeed behaves as the algorithm set forth in [1], and vice versa.

Finally, we develop a more general specification of a concurrent queue in CSP and then show that Michael and Scott's and OpenJDK's `ConcurrentLinkedQueue` Java implementation behave according to our new specification. With a simplified specification of a concurrent queue, we can use it in place of the larger implementation models to represent a concurrent queue when modelling systems.

1.1. Layout of the Paper

In Section 2 we provide a brief introduction of the language of CSP. We also discuss concurrent and lock-free data structures in general and the Michael and Scott algorithm in more detail. Finally, we consider related work and discuss serializability and linearizability.

Section 3 explains the conversion method we developed to transform the algorithm and the OpenJDK code into CSP. We also explain the CSP of the memory system to support the internal queue data structure. Section 3 also presents the simplified concurrent queue specification. In Section 4 we examine OpenJDK's implementation of *ConcurrentLinkedQueue* and consider the CSP translation of it using the method from Section 3. In Section 5 we investigate why a simple sequential queue cannot replace a more complex specification for a concurrent queue as well as proving linearizability or our concurrent queue specification. Finally, Section 6 contains a summary of the results and Section 7 wraps up with a conclusion and some future work.

2. Background and Related Work

In this section we present the background of our work and further related work. We start with a brief introduction to Communicating Sequential Processes (CSP) [3–6]. We then discuss concurrent and non-blocking data structures, including Michael and Scott's algorithm for a wait-free concurrent queue [1] and OpenJDK's *ConcurrentLinkedQueue*. Finally, we discuss serializability and linearizability [8], and present related work.

2.1. Communicating Sequential Processes

Communicating Sequential Processes (CSP) [3,4,6,9] is a process algebra used for specifying concurrent systems through the use of *processes* and *events*. Processes are abstract components characterised by the events they perform.

Events in CSP are *atomic*, *synchronous*, and *instantaneous*: events are indivisible, causing all participating processes to wait until the event occurs, and when an event occurs, it does so immediately for all involved processes. A CSP model specifies the events with which processes are willing to synchronise at various stages of their execution.

The simplest CSP processes are STOP — which performs no events and does not terminate — and SKIP — which also performs no events but will eventually terminate.

We introduce events into a process using the *prefix* operator ($\rightarrow$). For example, the process $P = x \rightarrow \mathsf{STOP}$ engages in event x, then halts. The general form of a process definition is: *Process* = *event* $\rightarrow$ *Process'*. Processes can be recursive (e.g., $P = x \rightarrow P$).

2.1.1. Choice

CSP provides several *choice* operators to model branching behaviour. The three most commonly used types of choice are *external (or deterministic) choice*, *internal (or non-deterministic) choice*, and *prefix choice*.

Given two processes P and Q, the definition $P \Box Q$ (external choice) represents a process that will behave as P or Q depending on the first event offered by the environment. For example, the process:

$$P = (a \rightarrow Q) \Box (b \rightarrow R)$$

can accept event a and then behave as Q, or accept event b and then behave as R. The system can non-deterministically choose between the two options when both a and b are available.

 doi:10.20944/preprints202505.0924.v1

An internal choice, represented by $\sqcap$, allows a process to behave as either P or Q without considering the external environment. That is, a process $P \sqcap Q$ can behave as either P or Q in isolation.

Both external and internal choices can be applied to a set of events. If $E = e_1, \ldots, e_n$ is a set of events, then:

$$\Box_{a \in E} a \to P \equiv e_1 \to P \Box e_2 \to P \Box \cdots \Box e_n \to P$$
$$\sqcap_{a \in E} a \to P \equiv e_1 \to P \sqcap e_2 \to P \sqcap \cdots \sqcap e_n \to P$$

With prefix choice, we can define an event and a parameter. If we define a set of events as $c.v \mid v \in Values$, we can interpret c as a channel willing to communicate a value v. In this case, input and output operations (? and !, respectively) are used to specify communication and the binding of variables. This shorthand can be represented by the following identities:

$$c!v \to P \equiv c.v \to P$$
$$c?x \to P \equiv \Box_{x \in Values} c.x \to P$$

where *Values* is a finite set.

CSP supports a functional *if* statement, where each branch must conclude with a process definition. The semantics of $c?v \to (\text{if } (v == x) \text{ then } P \text{ else } Q)$ in CSP is equivalent to $(c.x \to P) \Box (\Box_{v \in X - x} c.v \to Q)$[2].

2.1.2. Pre-Guards

The availability of choice branches can be controlled by placing a boolean expression followed by a & as a pre-guard before each branch. For example:

$$e_1 \;\&\; c?x \to \ldots \Box\, e_2 \;\&\; d?x \to \ldots$$

If the boolean expression e_1 is true and e_2 is false, only the $c?x$ guard will be considered. Similarly, if only e_2 is true, only the $d?x$ guard will be considered.

2.1.3. Process Composition

Processes can be combined through *parallel* and *sequential* composition. We denote parallel composition as $P \,||\, Q$, where P and Q must synchronise on a shared set of events. There are two forms of parallel composition:

- The generalised parallel $P \underset{A}{||} Q$ defines two processes that synchronise on a specific *synchronisation set* A. In CSP_M (a machine-readable version of CSP) this is written as $P|[A]|Q$.
- The alphabetised parallel $P \;{}_A||_B\; Q$ defines two processes, each restricted to their respective *alphabets*: P with alphabet A and Q with alphabet B. In CSP_Mthis is written as $P|[A \mid B]|Q$.

For generalised parallel, both P and Q must offer an event in A simultaneously for it to occur. Events outside A do not synchronise P and Q. For example:

$$(a \to b \to P) \underset{\{b\}}{||} (b \to c \to Q)$$

Here, a must occur first, them both processes syncrhonise on b and then c is performed. If the synchronisation set contained c, the right-hand side process would block as the left-hand does not offer c.

For alphabetised parallel, P and Q synchronise on their alphabets' intersection. For example:

$$(a \to b \to P) \;{}_{\{a,b\}}||_{\{b,c\}}\; (b \to c \to Q)$$

[2] Assuming that v does not appear in P.

 doi:10.20944/preprints202505.0924.v1

Both processes synchronise on $\{b\}$. a must happen first, followed by b synchronising and then c. If the left-hand side alphabet did not contain a, the system would deadlock as a could not be performed.

Two processes can also *interleave*: $P \,|||\, Q$, where P and Q execute in parallel but *do not* synchronise on any shared events. We denote sequential composition as $P\,;Q$, meaning P must terminate before Q begins.

2.1.4. Traces and Hiding

The *trace set* (or *traces*) of a process is all externally observed sequences of events a process can perform. For example, $P = a \rightarrow P$ has the empty trace $\langle\rangle$ as the shortest observable trace. If P performs a, the trace is $\langle a\rangle$. Each subsequent a extends the trace. The traces of P are $traces(P) = \{\langle\rangle, \langle a\rangle, \langle a, a\rangle, \ldots\}$. Similarly, for a process $Q = a \rightarrow b \rightarrow Q$, the traces are $\{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, a\rangle, \ldots\}$.

We use the hiding operator $\setminus$ to conceal events, replacing events with the (ignored) τ event. For example, $(a \rightarrow b \rightarrow a \rightarrow \mathsf{SKIP})\setminus\{a\}$ has traces $\{\langle\rangle, \langle b\rangle\}$. Similarly, $traces(P\setminus\{a\}) = \{\langle\rangle\}$, and $traces(Q\setminus\{a\}) = \{\langle\rangle, \langle b\rangle, \langle b, b\rangle, \ldots\}$.

2.1.5. Models

There are three semantic models of behaviour to analyse CSP models: the *traces* model, the *failures* model, and the *failures/divergence* model.

The Traces Model

The **traces model** defines externally observable behaviour. If the traces of a system *implementation* Q is a subset of the trace set of a system *specification* P (i.e., $traces(Q) \subseteq traces(P)$), we say $P \sqsubseteq_T Q$ (Q *trace refines* P). In a refinement test, *Specification* $\sqsubseteq_T$ *Implementation*, the *Specification* represents allowable behaviour.

Hiding events allows us to focus on the external behaviour of a process. An implementation may include events not present in the specification to model the system's implementation. We hide internal events of the implementation when comparing to the specification.

The Stable Failures Model

The **stable failures model** [6,9] examines events refused by a process after a trace. A stable state is one where a process cannot make internal progress (i.e., via hidden events) and must engage externally. A refusal is an event that a process cannot participate in when in a stable state.

The stable failures model overcomes limitations of trace comparison. For example, $(P = a \rightarrow P) \sqsubseteq_T ((P = a \rightarrow P) \sqcap \mathsf{STOP})$, although the right-hand side may non-deterministically refuse to accept any event. A *failure* is a pair (s, X): s is a trace, and X is the set of refused events after the trace s.

If $P \sqsubseteq_F Q$ then whenever Q refuses to an event, P does likewise. More formally, $P \sqsubseteq_F Q \Leftrightarrow failures(Q) \subseteq failures(P)$.

The Failures-Divergences Model

Divergences are potential livelock scenarios, where a process continuously performs internal events without making externally observable. For example $P = a \rightarrow \mathsf{STOP} \sqsubseteq_F Q = (a \rightarrow \mathsf{STOP}) \sqcap \mathsf{DIV}$, where DIV is an immediately diverging process. However, Q can refuse a and continuously perform τ. The refinement $\sqsubseteq_{FD}$ allows comparisons between such processes.

Formally, process P has a pair: $(failures_{\perp}(P), divergences(P))$. $P \sqsubseteq_{FD} Q \Leftrightarrow failures(Q) \subseteq failures(P) \wedge divergences(Q) \subseteq divergences(P)$. $failures_{\perp}(P)$ is defined as $failures(P) \cup (s, X) \mid s \in divergences(P)$; the set of traces leading to divergence are added to the set of stable failures to form the extended failures set.

Of note, if both the specification and the implementation are divergence free, we only need to establish equivalence in the stable failures model. For our verification purposes, both specification and implementation are divergence-free, so we only verify $SPEC \sqsubseteq_F IMPLEMENTATION$.

 doi:10.20944/preprints202505.0924.v1

2.1.6. FDR

FDR [10] can verify properties of CSP processes written in CSP_M, testing models against specifications. FDR supports the three models discussed in the previous section. There are other tests for deadlock freedom, divergence freedom, and determinism.

In the stable failures model, FDR considers a process P to be deterministic if no evidence of non-determinism exists. A "witness" is defined as a trace tr and an event a, such that P can both accept and refuse a after the trace tr [7]. In the failures-divergences model, the process must be divergence free.

2.1.7. Modules

CSP_M has a *module* system that allows encapsulation of defined names. The typical structure of a module is as follows:

```
module ModuleName
   <private declaration list>
exports
   <public declaration list>
endmodule
```

Modules can be nested and parametrised (and instantiated). Declarations from the public declaration can be accessed as *ModuleName::VariableName*.

2.2. Concurrent and Non-Blocking Data Structures

Safe concurrent access to a shared data structure normally requires mutual exclusion. Often, we use locks to ensure only a single thread can access the data structure at a time. There may be some possible concurrent operations, but modification of the data structure normally requires sequential access control.

Sequential access control creates bottlenecks as we lose concurrency. Data structures that do not require locks are advantageous. There are three different levels of non-blocking data structures (from weakest to strongest):

1. *Obstruction-Free*: A thread will progress if all other threads are suspended. This differs from using locks as when a thread holding a lock gets suspended, a thread waiting for the lock can continue.
2. *Lock-Free*: A data structure is lock-free if at least one thread can continue execution at any time. This differs from obstruction-freedom in that at least one thread remains non-starving, regardless of suspension.
3. *Wait-Free*: A data structure is wait-free if it is lock-free and ensures that every thread will make progress after a finite number of steps.

The concurrent queue that we are considering in this paper is *lock-free*. Lock-freedom is often supported via the implementation of *compare-and-swap* operations. Compare-and-swap ($CAS(x, ev, nv)$) will compare x to the value ev, and if $x == ev$, set $x = nv$. CAS returns a boolean indicating operation success.

The algorithm underpinning OpenJDK's implementation of *ConcurrentLinkedQueue* is the non-blocking (lock-free) concurrent queue algorithm by Michael and Scott [1]. The algorithm uses a linked list of nodes to store values in the queue, and CAS operations to support link changes.

2.3. Serializable vs. Linearizable

Serializability, is a well-known concept from database systems. Serializability ensures that a set of parallel events result in the same outcome if some serial order had been imposed on the events. In terms of CSP, such an ordering is a trace. Consider the following where P performs a then b then c before terminating. Q performs d then b then e before terminating. PQ is the parallel composition of P and Q synchronising on $\{b\}$:

 doi:10.20944/preprints202505.0924.v1

$$P = a \rightarrow b \rightarrow c \rightarrow \mathsf{SKIP}$$
$$Q = d \rightarrow b \rightarrow e \rightarrow \mathsf{SKIP}$$
$$PQ = P \,[\![b]\!]\, Q$$

The traces[3] of PQ are $\{\langle a,d,b,c,e\rangle, \langle d,a,b,c,e\rangle, \langle a,d,b,e,c\rangle, \langle d,a,b,e,c\rangle\}$. a and d can happen in any order before b. Similarly, c and e can happen in any order after b.

Herlihy and Wing [8] define serializability as: "A history is serializable if it is equivalent to one in which transactions appear to execute sequentially, i.e., without interleaving." In order words, we are guaranteed transactional isolation.

Linearizability originates from concurrent object systems. It is a stronger consistency condition than serializability, requiring that each individual operation on a shared object appears to occur atomically at some point between its invocation and response — known as the *linearization point*. Linearizability preserves the real-time ordering of non-overlapping operations — if one operation completes before another begins, this order must be reflected in the system's behaviour.

Assume two processes are interacting with a shared a register with an initial value of 0. We model their actions using *call* and *ret* events:

$$P = call.write!1 \rightarrow ret.write \rightarrow \mathsf{SKIP}$$
$$Q = call.read \rightarrow ret.read?x \rightarrow \mathsf{SKIP}$$
$$PQ = P \,|||\, Q$$

P and Q are invoking *write* and *read* concurrently. P and Q do not synchronise — each process independently calls and returns from the shared object.

Let P perform an invocation $call.write.1 \rightarrow ret.write$. If P completes its interaction before Q begins (i.e., $call.read$ follows $ret.write$ in the trace), then a linearizable execution must respect this ordering, and thus x would equal 1.

Let us consider a few trace examples:

- Linearizable traces:
  - $\langle call.write.1, ret.write, call.read, ret.read.1\rangle$ — P completes before Q begins, and Q sees the value of the register as 1.
  - $\langle call.read, ret.read.0, call.write.1, ret.write\rangle$ — opposite ordering, and so Q sees the value of the register as 0.
  - $\langle call.write.1, call.read, ret.read.1, ret.write\rangle$ — although P has not completed, it has begun and the linearization point has occurred, thus Q sees the value of the register as 1.
- Non-linearizable traces:
  - $\langle call.write.1, ret.write, call.read, ret.read.0\rangle$ — P completes before Q begins, but Q sees the value of the register as 0. This is not linearizable as it violates the real-time ordering.

Herlihy and Wing [8] define linearizability as: "A history is linearizable if it can be extended (by appending zero or more response events) to a history that is equivalent to some legal sequential history, and that preserves the real-time order of non-overlapping operations." In other words, linearizability ensures both consistency and temporal ordering of operations on shared state.

Linearization points are important as we can consider these as events in our models. We will return to use these points later in the paper.

2.4. Related Work

Lowe [2] has explored lock-free data structures using CSP using a practical approach. Lowe explored a garbage collected safe version [11] of Michael and Scott's [1] original queue, the former being the basis of the Java SDK's implementation of the `ConcurrentLinkedQueue`. Lowe's work is similar to ours in that it uses CSP to model a lock-free algorithm. However, Lowe's work does not

[3] We have omitted the ✓ representing termination.

 doi:10.20944/preprints202505.0924.v1

consider Michael and Scott's algorithm, nor whether the garbage collected version behaved as the original Michael and Scott algorithm. We explore both of these implementations and define how to specify a general concurrent data structure in CSP. Lowe's goal was to explore linearizability in general, whereas we are interested in understanding and specifying concurrent data structure behaviour for reuse in other models. Lowe is not explicit in the usage of linearization points, whereas we present them as fundamental for specifying concurrent data structures in CSP. Lowe's work is tightly coupled to the specific instance explored, rather than considering general cases — the models are task focused rather than specification focused.

Liu et al. [12] also explore linearizability using CSP, but with the PAT tool (another CSP refinement checker). Liu et al. specifically check for linearizability but without linearization points. Their work is more general than Lowe's, exploring linearizability outside a specific implementation case.

Verification of linearizability is an ongoing area of research. O'Hearn et al. [13] use a Hindsight Lemma to verify linearizability by inferring global state. They do not require linearization points. In contrast, our work models linearization points explicitly within CSP to enable automated verification.

Derrick et al. [14] also explore linearizability verification using Input/Output Automata (IOA) to ensure correctness under crash-recovery scenarios. In contrast, our work focuses on modelling linearizability using CSP, facilitating automated verification through model checking. Integrating Derrick et al.'s insights could potentially enhance our approach, particularly for systems requiring durability guarantees.

Dongol et al [15] provide a more thorough overview of linearizability verification examining work undertaken and classifying them into different approach categories.

3. Methods

In this section we present our approach to produce the CSP models of the implementations of OpenJDK's *ConcurrentLinkedQueue* and Michael and Scott algorithm. We start by considering the transformation of algorithms into something more CSP friendly.

The second part of this section presents the memory model that we have implemented; we have modelled both regular and atomic variables as well as the compare-and-swap actions of atomics.

Finally, the third part of this section develops a sequential and a concurrent queue specification that can be used for both the OpenJDK implementation and the Michael and Scott's algorithm for verification using CSP and FDR.

3.1. OO to CSP Pipeline

Figure 1 illustrates the two pipelines used in translation of the original algorithm (the top one) as well as the OpenJDK implementation (the bottom one).

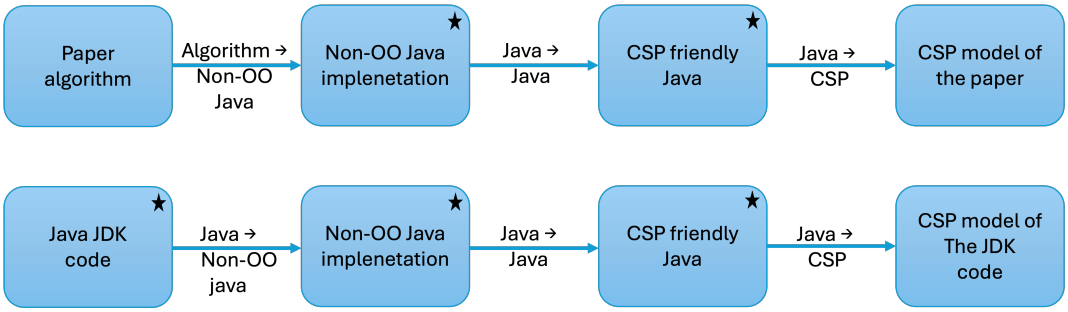

Figure 1. Translation Pipeline.

The algorithm from [1] is written in pseudo-C. This can be relatively easily translated to non-object-oriented Java. During the second step we are concerned with the translation of Java code into something more easily translatable to CSP; that is, we replace loops with recursion and if-statements are completed with else-parts. Finally, we translate the resulting Java code into CSP.

 doi:10.20944/preprints202505.0924.v1

For the OpenJDK implementation we OO-Java code, that must be converted to non-OO Java, then to CSP-friendly Java and finally to CSP.

Boxes in Figure 1 marked with a ★ represent Java code that has been tested and found to perform correctly.

3.1.1. Common Code

We define two data types — *Node* and *TestQueue*. *Node* is defined in Michael and Scott's work as:

- *value*: An *AtomicInteger* for the value stored on the node. The value can be **null**.
- *next*: An *AtomicReference*<*Node*> to the next node in the queue. The node can be **null**.

TestQueue also has two values:

- *head*: An *AtomicReference*<*Node*> pointed to the head node of the queue.
- *tail*: An *AtomicReference*<*Node*> pointed to the tail node of the queue.

These values will never be **null** even for an empty queue.

3.1.2. Converting Michael and Scott to Structured Java

Let us first look at Michael and Scott's [1] algorithm for enqueuing a new value onto a non-blocking queue:

```
node = new_node()
node→value = value
node→next.ptr = null
loop
   tail = Q→Tail
   next = tail.ptr→next
   if tail == Q→Tail
      if next.ptr == null
         if CAS(&tail.ptr→next, next, node)
            break
         endif
      else
         CAS(&Q→Tail, tail, next.ptr)
      endif
   endif
endloop
CAS(&Q→Tail, tail, node)
```

Michael and Scott [1] explain their algorithm in more detail in their paper. Here, we are concerned with the process of converting this code into a structured (minimal-OO) Java implementation. Let us break the code down into separate parts.

Firstly, the algorithm initialises a new node. Converting this code into Java is straightforward:

Michael & Scott	Structured Java
node = *new_node*() *node*→*value* = *value* *node*→*next.ptr* = **null**	*Node node* = **new** *Node*(); *node.value.set*(*value*); *node.next.set*(**null**);

value and *next* in the Java implementation are *AtomicInteger* and *AtomicReference* types respectively. To assign values we use the *set*() method of the atomic types and to retrieve values the *get*() method.

The loop in Michael and Scott's algorithm is a **while**-loop in Java and **if** is a simple replacement. Therefore, we only need concern ourselves with the **CAS** calls. In Java, these are replaced with *compareAndSet*() method calls on the atomic types.

 doi:10.20944/preprints202505.0924.v1

With these steps in place, we can convert Michael and Scott's algorithm to structured Java. We provide the complete enqueue method is provided in Table 1, and dequeue is likewise straightforward.

Table 1. Conversion of Michael and Scott's algorithm to Java.

Michael & Scott	Structured Java
node = *new_node*() *node→value* = *value* *node→next.ptr* = **null**	*Node node* = **new** *Node*(); *node.value.set*(*value*); *node.next.set*(**null**); *Node tail*; *Node next*;
loop	**while** (**true**)
tail = *Q→Tail* *next* = *tail.ptr→next*	*tail* = *Q.tail.get*(); *next* = *tail.next.get*();
if *tail* == *Q→Tail*	**if** (*tail* == *Q.tail.get*()) {
if *next.ptr* == **null**	**if** (*next* == **null**) {
if CAS(&*tail.ptr→next*, *next*, *node*)	**if** (*tail.next.compareAndSet*(*next*, *node*)) {
break **endif**	**break**; } }
else	**else** {
CAS(&*Q→Tail*, *tail*, *next.ptr*) | **endif** **endif**	*Q.tail.compareAndSet*(*tail*, *next*); } }
endloop	}
CAS(&*Q→Tail*, *tail*, *node*) |	*Q.tail.compareAndSet*(*tail*, *node*); |

3.1.3. Converting OpenJDK Code to Structured Java

We have used the OpenJDK source code for *ConcurrentLinkedQueue*[4] as the Java implementation. The code is written in an object-oriented manner, incompatible with CSP. We therefore also convert it into structured code format. To do so, we create **static** methods that take in a *TestQueue* variable and operate on it rather than member variables within a Java object instance. The two relevant methods to reimplement are:

- *offer*() becomes *enqueue*().
- *poll* becomes *dequeue*().

We ignore all the other methods as they are not relevant for our work.

There are two challenges in the OpenJDK source code:

- *Ternary Operators*: In the *offer*() method, OpenJDK uses ternary operators to update values. For example, *p* = *(t != (t = tail)) ? t : head;*. CSP does not support such operations, so we expand them into a sequence of operations to aid later conversion.
- Use of *VarHandle*: Rather than use Java's atomic types directly, OpenJDK uses *VarHandle*s to nodes and values as an optimisation. A *VarHandle* has atomic operators (e.g., *compareAndSet*). We convert this approach to use the standard Java *AtomicReference* and *AtomicInteger* types.

With these, we can produce equivalent structured version of the OpenJDK *ConcurrentLinkedQueue* implementation.

3.1.4. Converting Structured Java to CSP Friendly Java

Our next step is converting structured Java code to be more CSP-friendly. CSP_M has a functional syntax, lacks global variables, and cannot chain calls together. To convert the structured Java to a more CSP-friendly version we:

- Remove method chaining.

[4] https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/concurrent/ConcurrentLinkedQueue.java

 doi:10.20944/preprints202505.0924.v1

- Store return values before use.
- Expand conditionals fully.
- Move final operations outside loops.
- Flag loop continues.

Method chaining is a standard method in object-orientation. For example, our structured Java code will contain lines such as:

```
node.next.set(null);
```

In CSP_M, we have no method chaining equivalent, and therefore we must introduce temporary variables. For example, we expand the code above into:

```
tmp_node = node.next;
tmp_node.set(null);
```

Similarly, we cannot use return values from method calls directly and must store them. *compareAndSet*() returns a success condition and in our structured Java code, we store this first.

```
if (tail.next.compareAndSet(next, node))
```

which we convert to:

```
tmp_node = tail.next;
boolean succ = tmp_node.compareAndSet(next, node);
if (succ)
  ...
```

For conditionals, CSP_M requires both branches to be defined. We therefore add all **else** branches. Often, when the conditional branch does nothing, the behaviour is to start the next iteration of the operation loop. We flag this using **continue** which making it easier to see when converting loops to recursion for CSP_M.

Finally, we take any final operation after the main loop and move it into the necessary branches of the conditionals within the loop. Again, this is to make it easier to undertake the final conversion.

As an example of these final steps, let us examine the loop for enqueueing a value provided by Michael and Scott. In our structured Java, this is defined as:

```
while (true) {
  tail = Q.tail.get();
  next = tail.next.get();
  if (tail == Q.tail.get()) {
    if (next == null) {
      if (tail.next.compareAndSet(next, node))
        break;
    }
    else {
      Q.tail.compareAndSet(tail, next);
    }
  }
}
Q.tail.compareAndSet(tail, node);
```

The CSP-friendly version of the loop expands out the necessary steps and introduces complete conditions, and other steps. This version of the code is as follows:

 doi:10.20944/preprints202505.0924.v1


```
while (true){
    tail = Q.tail.get();
    tmp_node = tail.next;
    next = tmp_node.get();
    Node tmp = Q.tail.get();
    if (tail == tmp) {
        if (next == null) {
            tmp_node = tail.next;
            boolean succ = tmp_node.compareAndSet(next, node);
            if (succ) {
                Q.tail.compareAndSet(tail, node);
                break;
            }
            else {
                continue;
            }
        }
        else {
            Q.tail.compareAndSet(tail, next);
            continue;
        }
    }
    else {
        continue;
    }
}
```

3.2. Implementing Shared Memory Programs in CSP_M

With CSP-friendly Java code, we can complete the transformation from original code to CSP_M. There are three areas of consideration for our work: how to model global state in CSP, how to model atomic operations in CSP, and how to convert Java code structures to CSP_M.

3.2.1. Modelling Global State in CSP

CSP only has a global event space without a global data space. Therefore, we must model global variables via global events, using processes to manage state values. A state variable is a process that maintains the current variable value with events to *load* the current value or *store* a new value. We define a process *VARIABLE* as follows:

$$
\begin{aligned}
&VARIABLE(myLoad, myStore, val) = \\
&\quad (myLoad!val \rightarrow VARIABLE(myLoad, myStore, val)) \\
&\quad \Box \\
&\quad (myStore?newVal \rightarrow VARIABLE(myLoad, myStore, newVal))
\end{aligned}
$$

myLoad communicates the value to the environment, *myStore* accepts new values, and *val* is the current value.

3.2.2. Modelling Atomic Variables in CSP

Our variables provide atomic interaction, insofar that *load* and *store* operations are immediate and indivisible. However, the queue implementations also use *CAS* (*compare-and-swap*).

CSP channels can perform input and output operations in a single step, so are well suited for modelling *CAS* operations. We can extend *VARIABLE* to include *CAS*:

 doi:10.20944/preprints202505.0924.v1

ATOMIC_VARIABLE(*get*, *set*, *cas*, *val*) =
  get!*val* → *ATOMIC_VARIABLE*(*get*, *set*, *cas*, *val*)
  □
  set?*val* → *ATOMIC_VARIABLE*(*get*, *set*, *cas*, *val*)
  □
  cas?*expected*?*newVal*!(*expected* == *val*) →
    if (*expected* == *val*) then
      ATOMIC_VARIABLE(*get*, *set*, *cas*, *newVal*)
    else
      ATOMIC_VARIABLE(*get*, *set*, *cas*, *val*)

get communicates the value to the environment, *set* accepts new values, and *cas* accepts a value to compare with, a new value to set if the comparison is successful, and the success. *val* is the current value of the variable.

3.2.3. Converting CSP Friendly Java to CSP_M

We define a CSP system via a set of processes. We define each process by a set of events that can be performed. Events can carry types, and can thereby allow input and output parameters. The CSP_M language allows us to define processes that can communicate with each other via these events.

We define object-oriented systems in terms of classes and methods. In CSP_M, we can define the operation of a method as a process, and the events of the process as the methods invocations. The parameters of the methods are the input and output types of the invocation events.

Objects (and data structures in general) are passive, whereas processes are active. In CSP_M, we can model passive objects as processes that communicate values with the environment, and the threads using passive objects as processes invoking events on the passive objects. This allows us to model the behaviour of the system in a more structured way.

For example, we can define a queue as a pointer to its head and tail. In Michael and Scott's algorithm, this is specified as:

structure *pointer_t* {*ptr*: **pointer to** *node_t*}
structure *node_t* {*value*: *datatype*, *next*: *pointer_t*}
structure *queue_t* {*Head*: *pointer_t*, *Tail*: *pointer_t*}

Although we could define types in CSP_M to represent these structures, we would still require processes to manage the values. As such, we define a process *QUEUE* that represents the head and tail as two separate *ATOMIC_VARIABLE*s.

HEAD_PTR =
  ATOMIC_VARIABLE(*queue.HEAD.load*, *queue.HEAD.store*, *queue_cas.HEAD*, *NODE.1*)

TAIL_PTR =
  ATOMIC_VARIABLE(*queue.TAIL.load*, *queue.TAIL.store*, *queue_cas.TAIL*, *NODE.1*)

QUEUE_OBJ = *HEAD_PTR* ||| *TAIL_PTR*

We define the *queue* and *queue_cas* events as:

channel *queue* : *QUEUE.AccessOperations.Nodes_Not_Null*
channel *queue_cas* : *QUEUE.Nodes_Not_Null.Nodes_Not_Null.Bool*

where we define *QUEUE* as an enumerated datatype with values *HEAD* and *TAIL*. *AccessOperations* is the enumerated datatype *load* and *store*, and the value types are defined accordingly.

Nodes_Not_Null represents all the node IDs in the system not including **null** (0). As there is no passive data creation, *we must* create processes for each node the system will use, each with an identifier (much like a memory address — like *NODE.1* above) to allow access.

 doi:10.20944/preprints202505.0924.v1

All such data values require such definition and will execute during system operation. For example, the full set of *NODE*s we define as:

```
NODE_OBJS =
  ||| id : NODES_NOT_NULL •
    (
      VARIABLE(next.load.NODE.id, next.store.NODE.id, ANODE.NULL)
      |||
      VARIABLE(value.load.NODE.id, value.store.NODE.id, AINT.NULL)
    )
```

Each property of the node is defined as a separate process. The *ANODE* and *AINT* are the atomic variables for the node and integer respectively.

To simplify interaction, we define aliases to simplify the algorithm. For example, we define the following aliases for accessing the queue:

```
getHead = queue.HEAD.load
setHead = queue.HEAD.store
casHead = queue_cas.HEAD
getTail = queue.TAIL.load
setTail = queue.TAIL.store
casTail = queue_cas.TAIL
```

We want to observe method call events to test correct operation. We define the *enqueue* and *dequeue* events as:

```
channel dequeue, end_enqueue : Processes
channel enqueue : Processes.Integers_Not_Null
channel return : Processes.Integers
```

This creates *begin* and *end* operation pairs (i.e., *enqueue* → *end_enqueue* and *dequeue* → *return*) that we can compare to our specification.

Our final change is to replace loops with recursion. This is a fairly straightforward process, using **continue** to indicate when to recurse. For example, we replace the enqueue loop in Michael and Scott's algorithm with:

```
ENQUEUE'(node, tl, nxt) =
  getTail?tl →
  getNext.tl?tmp_node →
  getNode.tmp_node?nxt →
  getTail?tmp →
  if (tl == tmp) then
  (
    if (nxt == nullNode) then
    (
      getNext.tl?tmp_node →
      casNode.tmp_node!nxt!node?succ →
      if (succ) then
      (
        casTail!tl!node?succ →
        SKIP
      )
      else
```

```
while true
  tail = Q→tail
  next = tail.ptr→next

  if tail == Q→tail
  {
    if next.ptr == null
    {

      if CAS(&tail.ptr→next, next, node)
      {
        CAS(&Q→tail, tail, node)
        break
      }
      else
```

 doi:10.20944/preprints202505.0924.v1


```
            ENQUEUE'(node, tl, nxt)                    continue
        )                                          }
        else                                       else
        (                                          {
            casTail!tl!nxt?succ →                      CAS(Q→tail, tail, next.ptr)
            ENQUEUE'(node, tl, nxt)                    continue
        )                                          }
    )                                          }
    else                                       else
        ENQUEUE'(node, tl, nxt)                    continue
```

using SKIP to end the recursive call through successful termination at the return point of the algorithm. The *continue* points have been replaced with recursion calls. The complete code is available on GitHub[5]

3.3. Queue Specifications

We require specifications to determine if both current queue implementations operate as expected. We start with a simple specification of a *sequential* queue and then proceed to a specification of a *concurrent* queue that uses linearization points.

3.3.1. A Sequential Queue Specification

We model a sequential queue in CSP by a *choice* between accepting *enqueue* events (if the queue still has capacity[6]) and *dequeue*. An empty queue returns **null**.

```
QUEUE_SPEC(q) =
    length(q) ≤ MAX_QUEUE_LENGTH &
        enqueue?proc?v → end_enqueue.proc → QUEUE_SPEC(q^ < v >)
    □
    dequeue?proc →
        if (length(q) == 0) then
            return.proc!mem::INT.mem::NULL → QUEUE_SPEC(q)
        else
            return.proc!(head(q)) → QUEUE_SPEC(tail(q))
```

A user process simulates a thread interacting with the queue. There may be multiple threads, increasing the number of users to simulate additional threads interacting with the queue. We define the thread process as *USER*:

```
USER(id, count) =
    count > 0 & enqueue.id?value → end_enqueue.id → USER(id, count - 1)
    □
    dequeue.id → return.id?value → USER(id, count)
```

Each *USER* will enqueue a set number of values (the initial value of *count*) or choose to dequeue a value.

We wrap the sequential queue definition in a module *SeqQueue*, providing a *SPEC* process to act as the specification based on the number of user threads:

```
module SeqQueue
    USER(id, count) = ... – from above
    QUEUE_SPEC(q) = ... – from above
```

[5] https://github.com/mattunlv/Concurrent-Queue

[6] We need to set an upper limit of the length of the queue here. This is necessary because CSP does not handle unlimited data structures.

 doi:10.20944/preprints202505.0924.v1

exports
  $SPEC(users)$ =
    $|||\ id : users \bullet USER(id, MAX\_QUEUE\_LENGTH\ /\ \mathsf{card}(users))$
    $[\![\ \alpha INTERACTION\ ]\!]$
    $QUEUE\_SPEC(<>)$
endmodule

where $\alpha INTERACTION$ is the synchronising the processes. We omit the alphabets from the paper, but they can be found online.

With this sequential queue, we can consider the specification of a concurrent queue.

3.3.2. A Concurrent Queue Specification

A queue can *enqueue* elements, or *dequeue* elements. However, we must allow the following scenario. In a system with two processes trying to enqueue a value where the *enqueue* event takes a process identifier and a value (e.g., *enqueue*.0.1 — process 0 wants to enqueue 1), we must consider happens with two (concurrent) events:

$$enqueue.0.1\ |||\ enqueue.1.2$$

Process 0 wants to enqueue 1 and process 1 wants to enqueue 2. If the *enqueue* event is the point of commitment, the trace *enqueue*.0.1, *enqueue*.1.2 **must** result in [1, 2] and the trace *enqueue*.1.2, *enqueue*.0.1 must result in [2, 1]. We know that is not necessarily true. It should be possible for the trace *enqueue*.0.1, *enqueue*.1.2 to result in the queue [2, 1]. In order to allow such behaviour, we expand *enqueue* to consist of three events:

- *enqueue.proc.value* — the start of an enqueue operation.
- *lin_enqueue.proc.value* — the time of commitment — *the linearization point* — where the value is committed into the queue.
- *end_enqueue.proc* — the end of an enqueuing operation.

For the dequeuing we have events *dequeue.proc*, *lin_dequeue.proc.value*, and *return.proc.value*. Note, there is always a sequential ordering within the three events of an enqueue and a dequeue. We can now observe the following trace:

$$\langle enqueue.0.1, enqueue.1.2, lin\_enqueue.1.2, end\_enqueue.1, lin\_enqueue.0.1, end\_enqueue.0\rangle$$

that results in a queue like [2, 1].

Figure 2 illustrates this point. S represents the start of the enqueue, C the commit point, and E the end of the enqueue. We have four different scenarios that a second process (P_2) could be in relative to the first (P_1). Similar issues arise with dequeue as well.

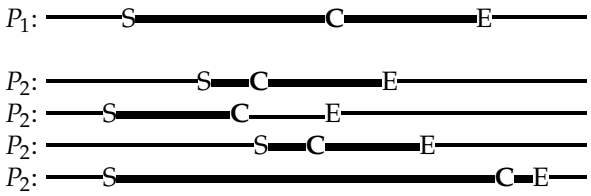

Figure 2. Example of different commit/linearization times.

In the first and third example of P_2 in Figure 2, P_2 starts *after* P_1 but commits its value *before* P_1. These commit points are **linearization** points.

A generic queue can now be specified as an external choice between these six events. The only two issues we have to handle are:

 doi:10.20944/preprints202505.0924.v1

- What happens if the queue is full? In reality this never happens as a linked list can grow until we run out of memory, but we must manage the system state size for verification, so we set an upper limit (*MAX_QUEUE_LENGTH*) and only allow *lin_enqueue* events to happen if there is room in the queue.
- What happens if the queue is empty on *dequeue*? In this case, we return **null**.

We define the CSP specification of a concurrent queue as:

$$
\begin{aligned}
&QUEUE\_SPEC(q) = \\
&\quad enqueue?\_?\_ \rightarrow QUEUE\_SPEC(q) \\
&\quad \Box \\
&\quad \mathsf{length}(q) \leq MAX\_QUEUE\_LENGTH \;\&\; lin\_enqueue?\_?v \rightarrow QUEUE\_SPEC(q^\frown < v >) \\
&\quad \Box \\
&\quad end\_enqueue?\_ \rightarrow QUEUE\_SPEC(q) \\
&\quad \Box \\
&\quad dequeue?\_ \rightarrow QUEUE\_SPEC(q) \\
&\quad \Box \\
&\quad (\\
&\qquad \mathsf{if}\ (\mathsf{length}(q) == 0)\ \mathsf{then} \\
&\qquad\quad lin\_dequeue?\_!mem::INT.mem::NULL \rightarrow QUEUE\_SPEC(q) \\
&\qquad \mathsf{else} \\
&\qquad\quad lin\_dequeue?\_!\mathsf{head}(q) \rightarrow QUEUE\_SPEC(\mathsf{tail}(q)) \\
&\quad) \\
&\quad \Box \\
&\quad return?\_?\_ \rightarrow QUEUE\_SPEC(q)
\end{aligned}
$$

Note the ?_?_ in *enqueue* and other events. The first _ represents the *process identifier* and the second the *value*. _ means "accept any value and discard it". The queue never cares about the *process identifier* and only cares about the *values* at the two linearization points. We cannot remove the *process identifier* and *value* as they are required for specification checking.

The *USER* process now uses the linearization events between *enqueue*/*end_enqueue* and *dequeue*/*return* events.

$$
\begin{aligned}
&USER(id, count) = \\
&\quad count > 0 \;\&\; enqueue.id?val \rightarrow lin\_enqueue.id.val \rightarrow end\_enqueue.id \rightarrow \\
&\qquad USER(id, count - 1) \\
&\quad \Box \\
&\quad dequeue.id \rightarrow lin\_dequeue.id?value \rightarrow return.id.value \rightarrow USER(id, count)
\end{aligned}
$$

We also create a *ConcQueue* module as *SeqQueue* to simply usage. The code is available online. Note, the two channels for linearization exist in the module as they are not required for communication with the external environment.

4. Examining OpenJDK's Implementation of a Current Queue

ConcurrentLinkedQueue [16] is a parametrised class (in the *java.util.concurrent* package) extending the parametrised class *AbstractQueue*, which extends the parametrised class *AbstractCollection*. It provides a thread-safe, lock-free, and non-blocking linked-list queue. The documentation cites [1] as the base of the implementation. We are primarily interested in inserting and removing elements from the queue, thus we only consider the following methods:

- *add/offer* — Inserts an element into queue.
- *poll* — Retrieves and removes an element from the queue, or returns null if this queue is empty.

 doi:10.20944/preprints202505.0924.v1

With access to the implementation and a specification in terms of [1], we wish to employ the following approach:

- Translate the OpenJDK Java implementation to CSP (this is the *implementation*).
- Use the CSP model of Michael and Scott's algorithm (this is the *specification*).
- Verify that the OpenJDK implementation *behaves* in accordance with the specification.

4.1. The OpenJDK/Java Version in CSP

To be brief, as we outlined in Section 3.1 the process and resulting code for checking the Michael and Scott algorithm, we do not present the CSP for the translated OpenJDK version of the queue. However, the complete code can be found in the online repository.

4.2. Results

We have translated the OpenJDK implementation of the concurrent queue and the Michael and Scott algorithm into CSP. We can now check that the two implementations behave like each other.

Firstly, we ensure *MSYSTEM* and *JSYSTEM* are deadlock- and divergence-free. Deadlock freedom is obvious, and divergence freedom means we only need to perform specification checking in the failures model. Both systems are indeed deadlock- and divergence-free up to three users (we limit to three users as the state space is too large for specification checking with four users).

We can now perform the check that the OpenJDK version behaves as the version proposed in [1] (and vice versa). The checks are:

$$MSYSTEM(X) \sqsubseteq_F JSYSTEM(X)$$
$$JSYSTEM(X) \sqsubseteq_F MSYSTEM(X)$$

where $X = \{P1\}, \{P1, P2\}, \{P1, P2, P3\}$. The check passes in both directions for both the *traces* and the *stable failures* models.

We have now shown that the OpenJDK implementation indeed behaves as Michael and Scott's algorithm. Though Michael and Scott argue for the correctness of their algorithm in the paper, we have no formal proof that indeed the algorithm — and by extension — the OpenJDK implementation behave as a concurrent queue.

We know that *MSYSTEM* and *JSYSTEM* behave like each other, but to ensure correctness we must compare them to *ConcQueue*. *ConcQueue* is both deadlock- and livelock-free, so we can check that the Michael and Scott algorithm behaves like *ConcQueue* in the failures model.

First, we check if *MSYSTEM* and *JSYSTEM* behave like a *sequential* queue. We can check if *MSYSTEM* behaves as a sequential queue with just a single process:

$$SeqQueue::SPEC(\{P1\}) \sqsubseteq_F MSYSTEM(\{P1\})$$
$$MSYSTEM(\{P1\}) \sqsubseteq_F SeqQueue::SPEC(\{P1\})$$

However, for more than one user process this check fails in the stable failures model. Instead, we get a single trace-refinement, namely:

$$MSYSTEM(\{P1, P2\}) \sqsubseteq_T SeqQueue::SPEC(\{P1, P2\})$$

This check is treating *MSYSTEM* as a specification of *SeqQueue*, and *SeqQueue* (as an implementation) only has traces that *MSYSTEM* also provides. In other words, the *SeqQueue* behaviour is still possible within *MQUEUE* (all traces are present). However, *MSYSTEM* can now perform multiple (two) *enqueue* or *dequeue* operations at a time. Thus, *MSYSTEM* has traces (and thus acceptances) that *SeqQueue* does not.

Using *ConcQueue* leads to passing in both directions in the stable failures model:

$$ConcQueue::SPEC(X) \sqsubseteq_F MSYSTEM(X)$$
$$MSYSTEM(X) \sqsubseteq_F ConcQueue::SPEC(X)$$

 doi:10.20944/preprints202505.0924.v1

where $X = \{P1, P2\}, \{P1, P2, P3\}$, and $\{P1, P2, P3, P4\}$. Thus, not only does *MSYSTEM* only behave as defined by *ConcQueue*, but *ConcQueue* can only behave as *MSYSTEM*.

Since *MSYSTEM* and *JSYSTEM* are failure-divergence refinements of each other, naturally, *JSYSTEM* and *ConcQueue* also failure-divergence refines each other.

4.3. Code Refactoring

We have proved that *ConcQueue* has the same behaviour as both *JSYSTEM* and *MSYSTEM* and vice verse. Since *ConcQueue* is a smaller and simpler implementation of a concurrent queue, any future model checking and formal verification that involved either the Michael/Scott version (*MSYSTEM*) or the OpenJDK version (*JSYSTEM*) can now simply swap either with the much simpler and smaller *ConcQueue*, which reduces the required state space to check.

Because our ConcQueue is modular and parameterised, it can be imported into new CSP models involving concurrent components without re-verifying underlying queue behaviours —— a common bottleneck in formal modelling.

5. Not Sequential but Linearizable

We now know that the OpenJDK concurrent and Michael and Scott's algorithm behave as the concurrent queue specification. However, there are still a few interesting questions to ask. The first question is whether the OpenJDK and the paper versions **also** behave like a regular sequential queue?

We presented a sequential queue in Section 3.3.1, and checked if *MSYSTEM* and *JSYSTEM* behave like a sequential queue. Unsurprisingly, with a single user process, both *MSYSTEM* and *JSYSTEM* do, but for any number of user processes greater than one, further behaviours are possible (note that the complete sequential queue behaviour was still possible).

Naturally, this is caused by a single process being unable to overlap operations. For more processes, a concurrent queue can overlap operations and that there is no guarantee that beginning first means reaching the linearization point first.

The second question we can answer is the concurrent queue is linearizable. The definition of linearizable is that the trace behaviour of the concurrent queue can be transformed into that of a sequential queue. That is, we can move the begin and end events of the concurrent queue to match the linearization points and the behaviour observed would be one that the sequential queue also provides. This converts the trace into a linearization.

Therefore, linearizability means that each operation must appear to commit in an instant at some point in time within the operation execution. We can describe a linearizable queue for which any history of concurrent operations has a linearization that matches a correct sequential history of operations on the queue.

Linearization points are when operations commit. If we ignore the begin and end events and observe just the linearization points, we can observe the points where the queue is modified. With this knowledge, we can create a new specification *SimpleQueue*. We use the *simple_enqueue* event to model adding a value to the queue and *simple_dequeue* to model removing a value from the queue:

module *SimpleQueue*
  SIMPLE_QUEUE(q) =
    length(q) <= *MAX_QUEUE_LENGTH* & *simple_enqueue*?*proc*?v →
      SIMPLE_QUEUE(q ^ < v >)
    □
    length(q) == 0 & *simple_dequeue*?*proc*!*mem::INT.mem::NULL* →
      SIMPLE_QUEUE(q)
    □
    length(q) > 0 & *simple_dequeue*?*proc*!head(q) → *SIMPLE_QUEUE*(tail(q))

  USER(*id*, *count*) =

 doi:10.20944/preprints202505.0924.v1

$count > 0$ & $simple\_enqueue.id?val \rightarrow USER(id, count - 1)$
$\square$
$simple\_dequeue.id?value \rightarrow USER(id, count)$

exports
SPEC(*users*) =
$|||\ id : users \bullet USER(id, MAX\_QUEUE\_LENGTH \ /\ \mathsf{card}(users))$
$|[\ \{|\ simple\_enqueue, simple\_dequeue\ |\}\ ]|$
$SIMPLE\_QUEUE(<>)$
endmodule

Thus, *SimpleQueue* models changes to the queue as atomic operations without begin and end events. We consider a trace from *SimpleQueue* to be a linearization that we can verify against. We do this by assuming that the begin and end events occur as part of the simple events. That is, we have reordered our begin and end events into a sequential trace by only observing the commit or linearization points. For example, an event *simple_enqueue* can be considered the trace $\langle begin\_enqueue, simple\_enqueu, end\_enqueue \rangle$.

We can now use *SimpleQueue* to check if *ConcQueue* only provides linearizations in its behaviour. As we have demonstrated that *ConcQueue* is specifies our two queue implementations, demonstrating *ConcQueue* only produces traces from *SimpleQueue* will prove that the two queue implementations are linearizable. We must modify *ConcQueue* so that linearization events are visible rather than hidden, and begin and end events are hidden rather than visible.

We add the following process to the *ConcQueue* where the *enqueue, end_enqueue, dequeue, return* events have been hidden and the *lin_enqueue* event has been renamed to *simple_enqueue* and *lin_dequeue* to *simple_dequeue* as follows:

$SPEC'(users) =$
(
$|||\ id : users \bullet USER(id, MAX\_QUEUE\_LENGTH \ /\ \mathsf{card}(users))$
$|[\ \{|\ enqueue, lin\_enqueue, end\_enqueue, dequeue, lin\_dequeue, return\ |\}\ ]|$
$QUEUE\_SPEC(<>)$
)$\setminus \{|\ enqueue, end\_enqueue, dequeue, return\ |\}$
$[[\ lin\_enqueue \leftarrow simple\_enqueue, lin\_dequeue \leftarrow simple\_dequeue]]$

We can now perform the following two successful checks:

$SimpleQueue::SPEC(X) \sqsubseteq_T ConcQueue::SPEC'(X)$
$ConcQueue::SPEC'(X) \sqsubseteq_F SimpleQueue::SPEC(X)$

where $X = \{P1\}, \{P1, P2\}, \{P1, P2, P3\}$.

Note, the second assertion passes in the stable failures model, whereas the first only passes in the traces model. This is expected. The stable failures model considers stable states — ones where there are no internal operations possible. We have hidden the begin events in *ConcQueue*, and a user committing to start an operation is a steady state. *SimpleQueue* has no such internal commitment to a stable state. However, we are not concerned with the failures model as for linearization we only need to demonstrate that the traces of *ConcQueue* are contained in the specification of *SimpleQueue*, which we have achieved.

6. Overall Results

In this section, we summarise the results obtained from Section 4.2 and Section 5. For a number of processes, N, we have shown the following results:

 doi:10.20944/preprints202505.0924.v1

(1)	*SeqQueue*	$\sqsubseteq_{FD}$ *MSYSTEM*	(for N = 1 and vice versa)
(2)	*SeqQueue*	$\sqsubseteq_{FD}$ *JSYSTEM*	(for N = 1 and vice versa)
(3)	*MSYSTEM*	$\sqsubseteq_{T}$ *SeqQueue*	(for $2 \leq N \leq 4$)
(4)	*JSYSTEM*	$\sqsubseteq_{T}$ *SeqQueue*	(for $2 \leq N \leq 4$)
(5)	*MSYSTEM*	$\sqsubseteq_{FD}$ *JSYSTEM*	(for $1 \leq N \leq 3$ and vice versa)
(6)	*ConcQueue*	$\sqsubseteq_{FD}$ *MSYSTEM*	(for $1 \leq N \leq 4$ and vice versa)
(7)	*ConcQueue*	$\sqsubseteq_{FD}$ *MSYSTEM* $\sqsubseteq_{FD}$ *JSYSTEM*	(for $1 \leq N \leq 3$ and vice versa)
(8)	*SimpleQueue*	$\sqsubseteq_{T}$ *ConcQueue*	(for $1 \leq N \leq 4$ — linearizable)

Both the sequential (*SeqQueue*) and the concurrent (*ConcQueue*) queue specifications are deadlock- and divergence-free in the failures-divergence model with one, two, and three *users*. This means that checking assertions in the stable failures model is sufficient to conclude that the assertion will hold in the failures/divergence model (See Section 2.1.5.3). Also, note that a single user can both *enqueue* and *dequeue* through an external choice, thus operating as if the system had a dedicated enqueue and a dedicated dequeue process.

Furthermore, the queue implementations of [1] (*MSYSTEM*) and the OpenJDK implementation (*JSYSTEM*) are also divergence- and deadlock-free for one, two, and three users.

The crux of the paper was two-fold:

- Show that the algorithm of Michael and Scott [1] behaves like a concurrent queue.
- Show that the OpenJDK implementation of the *ConcurrentLinkedQueue* behaves like Michael and Scott's algorithm and transitively is a correct implementation of a concurrent queue.

FDR proved that *MSYSTEM* failure refines *JSYSTEM* and vice-versa for one, two, and three users (See Section 4.2). Also in that same section, we establish a relationship between *MSYSTEM*/*JSYSTEM* and the *ConcQueue* specification. We achieved this by checking refinement in the stable failures model between *MSYSTEM* and *ConcQueue*. Again, FDR proved that refinement holds in both directions.

Because *MSYSTEM* and *JSYSTEM* failure refines each other in the stable-failures model, we can conclude that *JSYSTEM* also failure-refines *SeqQueue* and vice-versa.

The penultimate family of checks we performed were comparisons to the generic sequential and concurrent queue specifications. We started by asking a few questions about the *SeqQueue* and the *ConcQueue* with respect to refinement.

- Does the concurrent queue specification refine the sequential queue specification? For a single user, the answer is "yes, in the stable-failures model." For more users, the answer is "no, not even in the traces model."
- Does the sequential queue specification refine the concurrent queue specification? For a single user, the answer is again "yes, in the stable-failures model." For more users, the answer is "yes, but only in the traces model." This is because the sequential queue has a different set of failures compared to the concurrent queue; This is because the number of operations in progress in a sequential queue at any time is at most one, whereas in a concurrent queue many operations can be in progress at the same time.

And finally, the last family of check we performed were comparisons of the *ConcQueue* (with *enqueue*, *end_enqueue*, *dequeue*, and *return* events hidden, and the linearization events visible and renamed) to a simple queue (*SimpleQueue*). See Section 5.

Our results tell provide three clear outcomes:

- The first two results show that the concurrent queue implementations behave like a sequential queue when only one user thread is using them. The third and fourth result demonstrate that the concurrent queues are not sequential with more than one thread, but the concurrent queues still contain the complete behaviour of the sequential queue.
- The fifth result demonstrate that the OpenJDK implementation of a concurrent queue behaves as Michael and Scotts original algorithm (and vice-versa). The sixth result demonstrates that Michael and Scott's algorithm also behaves as our *ConcQueue* specification, leading to the seventh

 doi:10.20944/preprints202505.0924.v1

result through the transitive relationship of CSP refinement. This meets the key aim of our work — development of a specification of a concurrent queue that can be used in other models.
- The final result demonstrates that the concurrent queue specification and (and therefore the implementations) are linearizable, insofar that they behave as a queue that atomically allows adding and removing items. We only require trace refinement in this regard.

6.1. Limitations

Due to the state space significantly increasing as the number of users increases, our verification is limited:

- We have only checked $MSYSTEM \sqsubseteq_F JSYSTEM$ to three user processes. However, this is inline with other works using CSP for verification (e.g., Lowe [2]) of such complex models. Three user processes still provides a rich set of interactions with the queue.
- We have only checked $ConcQueue \sqsubseteq_F MSYSTEM$ for four user processes. Likewise, this provides a rich set of comparative behaviours to check against.
- We do not utilise any node recycling, allowing only a fixed number of enqueues to occur. Lowe [2] implemented a garbage collection system that we could have emulated (at the cost of increased state space complexity), but our work has focused on specification development. The queue allows multiple enqueues from different threads to occur, and unlimited dequeues. Again, this provides a rich set of behaviour transactions to explore for model checking.

These limitations are standard practice in model checking of concurrent structures because of the exponential blowup of the state space as the number of threads explored increases.

7. Conclusion and Future Work

7.1. Conclusion

In this paper we investigated the Michael and Scott algorithm for a wait-free concurrent queue found in [1] as well as the OpenJDK's implementation of the *ConcurrentLinkedQueue*. We have shown that the OpenJDK implementation behaves according to the algorithm and vice versa in the stable failures model of CSP. Furthermore, we have shown that both implementations also behave as a generic concurrent queue, and that they are linearizable. Formally, we have demonstrated:

$$ConcQueue \sqsubseteq_{FD} MSYSTEM \sqsubseteq_{FD} JSYSTEM$$

That is, the OpenJDK implementation meets the specification behaviour of Michael and Scott's algorithm, which itself meets the specification of a concurrent queue.

With such a result, we can be confident in using our *ConcQueue* specification in other models where we require a concurrent queue as part of the implementation. This will greatly reduce the state space that must be explored during any model checking.

7.1.1. Contributions

Our work strengthens existing approaches using CSP for linearizability by producing a reusable, formally verified concurrent queue model using CSP. We have produced a specification that is simple to plug into other models, enabling reduced state space verification. Additionally, we have demonstrated that our specification is linearizable by simply considering the linearization points that enable the concurrent behaviour.

Our approach is also general. By modelling specifications with linearization points, and a number of users to simulate concurrent access, we can build other concurrent data structure specifications simply. For example, a stack will be similar to a queue but with pushing and popping to the head of the sequence:

$$STACK\_SPEC(s) =$$
$$\vdots$$

 doi:10.20944/preprints202505.0924.v1

length(s) <= MAX & $lin\_push?proc?v \rightarrow STACK\_SPEC(< v > \hat{} s)$
□
if length(s) == 0 then
  $lin\_pop?proc!nullInt \rightarrow STACK\_SPEC(s)$
else
  $lin_pop?proc!$head(s) $\rightarrow STACK_SPEC($tail(s)$)$
⋮

We have demonstrated both theoretical value (the formal specification model) and practical value (application to OpenJDK), including establishing the behavioural equivalence of OpenJDK to the original work of Michael and Scott [1] through full failures/divergences refinement checking.

Furthermore, we have provided an outline for converting existing object-oriented Java code into CSP_M. We hope that the outline and techniques will enable others to accurately model concurrent Java applications using FDR.

7.2. Future Work

Our work has several directions of future research:

1. **Model Scaling via Composition Techniques**. While we have verified up to four concurrent users, the state space grows exponentially. Future work could explore compositional verification or data independence techniques to verify a greater number of users, following work by Roscoe and others.
2. **Node Recycling and Garbage Collection Semantics**. Out models assume a fixed number of enqueues and do not recycle nodes. Incorporating garbage collection (e.g., as Lowe [2]) and node deletion schemes would enable a more accurate modelling of real-world behaviour. The limitation is in the size of models that would be created.
3. **Application to Other Lock-Free Structures**. The methodology and reusable specification style developed here could be applied to stacks, dequeues, priority queues, and concurrent maps. These structures present unique challenges in terms of linearization point placement and memory consistency.
4. **Exploration of Other Memory Semantics**. We have assumed that atomic operations occur in the order they invoked (they are *sequentially consistent*). In modern memory systems, caching means we can consider different memory semantics (e.g., relaxed vs. sequentially consistent). Building a richer set of atomic memory components would allow exploration of these different semantics.
5. **Tool Integration and Usability**. While our CSP_Mmodels are available online, integrating our workflow into tools could broaden adoption of verification of Java (or other language) code. Semi-automated translation from structured Java to CSP_Mcould support teaching or wider industry adoption.
6. **Durable Linearizability**. Future work could also investigate durable linearizability — i.e., ensuring consistency after a failure — building on the work of Derrick et al. [14]. This is particularly relevant for persistent memory systems.

Data Availability Statement: All the code referenced in this paper can be found on GitHub at https://github.com/mattunlv/Concurrent-Queue

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Michael, M.M.; Scott, M.L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, New York, NY, USA, 1996; PODC '96, p. 267–275. https://doi.org/10.1145/248052.248106.

 doi:10.20944/preprints202505.0924.v1

2. Lowe, G. Analysing Lock-Free Linearizable Datatypes Using CSP. In *Concurrency, Security and Puzzles: Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*; Gibson-Robinson, T.; Hopcroft, P.; Lazić, R., Eds.; Springer, 2017; Vol. 10160, *LNCS*.
3. Hoare, C.A.R. Communicating Sequential Processes. *CACM* **1978**, *21*, 666–677.
4. Hoare, C.A.R. *Communicating Sequential Processes*; Prentice-Hall, 1985.
5. Roscoe, A. CSP is expressive enough for π. In *Reflections on the Work of CAR Hoare*; Springer, 2010; pp. 371–404.
6. Roscoe, A.W. *The theory and practice of concurrency*; Prentice Hall, 1998. The text book teaching material can be found at http://www.comlab.ox.ac.uk/publications/books/concurrency/.
7. FDR 4.2 Documentation. https://cocotec.io/fdr/manual/. Accessed: April 29, 2025.
8. Herlihy, M.P.; Wing, J.M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **1990**, *12*, 463–492. https://doi.org/10.1145/78969.78972.
9. Roscoe, A.W. *Understanding Concurrent Systems*; Springer, 2010.
10. Gibson-Robinson, T.; Armstrong, P.; Boulgakov, A.; Roscoe, A.W. FDR3 — A Modern Refinement Checker for CSP. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems; Ábrahám, E.; Havelund, K., Eds., 2014, Vol. 8413, *Lecture Notes in Computer Science*, pp. 187–201.
11. Herlihy, M.; Shavit, N.; Luchangco, V.; Spear, M. *The art of multiprocessor programming*; Newnes, 2020.
12. Liu, Y.; Chen, W.; Liu, Y.A.; Sun, J. Model checking linearizability via refinement. In Proceedings of the FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings 2. Springer, 2009, pp. 321–337.
13. O'Hearn, P.W.; Rinetzky, N.; Vechev, M.T.; Yahav, E.; Yorsh, G. Verifying linearizability with hindsight. In Proceedings of the Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, 2010, pp. 85–94.
14. Derrick, J.; Doherty, S.; Dongol, B.; Schellhorn, G.; Wehrheim, H. Verifying correctness of persistent concurrent data structures: a sound and complete method. *Formal Aspects of Computing* **2021**, *33*, 547–573.
15. Dongol, B.; Derrick, J. Verifying linearisability: A comparative survey. *ACM Computing Surveys (CSUR)* **2015**, *48*, 1–43.
16. OpenJDK. LinkedBlockingQueue.java. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html, 2018. Accessed: April 29, 2025.