COMP3770 Concurrency Proof

Author: Jimmy Gan,
Supervisor: Dr. Nisansala Prasanthi Yatapanage

October 2024

Abstract

Concurrent algorithms are notoriously challenging to verify, especially non-blocking ones due to interference between processes. The rely/guarantee (R/G) approach provides a mathematical framework for reasoning about concurrent processes. Linearisability is a common technique for verifying concurrent algorithms. This paper, following the footsteps of [Yat24], investigates the Michael Scott Queue, a non-blocking algorithm, and explores how R/G can be applied to it. In particular, we explore whether rely/guarantee reasoning can ensure linearisability. Our investigation reveals both the potential and limitations of the rely/guarantee approach in verifying concurrent algorithms.

1 Introduction

Due to the interference between different processes, concurrent algorithms are difficult to verify. This is especially true for non-blocking concurrent algorithms. Jones' rely/guarantee approach permits reasoning about concurrent processes by providing a mathematical framework for describing interference between processes [Jon83a, Jon83b].

The rely/guarantee approach has been used to successfully verify concurrent, non-blocking algorithms [JH16]. However, it is rather difficult to define the suitable rely/guarantee conditions for certain algorithms. [JY19] attempted to develop a concurrent garbage collector with rely/guarantee, but found that the method fell just short of fully expressing the process, requiring ghost variables and other constructs. This paper investigates one such algorithm, where it is rather challenging to use standard rely/guarantee conditions to describe its intended behaviour, but this method still provides useful insight.

In particular, the current paper investigates the issues of linearisability, where the concurrent program must match the behaviour of a sequential representation [HW90]. This is achieved by assigning a linearisation point where the concurrent program is considered to take effect. The topic of linearisation will be further discussed in this paper.

This paper follows the approach of [Yat24], which attempted to verify the Treiber Stack and the Herlihy-Wing Queue, where they found issues with lin-

earisability and the environment hampering the construction of a proof. Not many have attempted to compare rely/guarantee with linearisability besides the above-cited papers. This paper investigates whether similar issues with rely/guarantee exist with different non-blocking algorithms, and wishes to draw parallels between the other papers.

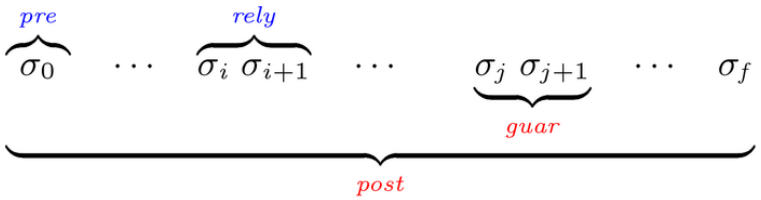
The rest of the paper is structured in the following manner: Section 2 introduces the background knowledge related to the paper; Section 3 defines the algorithm on an abstract level; Section 4 develops the abstract specification to a more concrete sequential specification; Section 5 attempts to refine the abstract specification to a concrete concurrent development; Section 6 relates the discoveries made to similar problems, and Section 7 gives the conclusions.

2 Background

2.1 Rely/guarantee

Rely/guarantee was invented by Cliff Jones [Jon83a, Jon83b] for verifying concurrent programs. [HJ17] provides a useful introduction. A rely condition is a requirement on the environment, it describes the interference the environment can make that is tolerable by the process. A guarantee condition is a promise the process made to the environment, it describes what the process is ensured to do.

The rely condition must be held by every environment step, while the guarantee condition must be held by every program step. For two processes to execute concurrently, the guarantee of one must satisfy the rely of the other.



pre/*rely* are assumptions the developer is invited to make
guar/*post* are commitments that the code must achieve

Figure 1: Picture from [JB24]

2.2 Michael Scott Queue

The algorithm investigated is the Michael Scott Queue [MS96], a queue structure that allows for enqueue and dequeue operations to run concurrently, without explicitly making processes wait for other processes' completion. It utilizes the *Compare-and-Swap (CAS)* operation, which is an atomic operation available on

most hardware. *CAS* compares the value at a specific memory location with a given expected value. If the value matches the expected one, it updates the memory location with a new value. If the comparison fails, no update occurs. *CAS* returns true on success and false if it does not.

The enqueue operation creates a new node with the given data, and then repeatedly checks for changes on the tail node and the tail pointer of the queue. Only when no changes are present during the loop does the *CAS* operation link the newly created node to the end of the queue. If not, it will attempt to swing the tail pointer to the end of the queue. The program will continuously attempt the enqueue operation until it succeeds.

The dequeue operation repeatedly checks for changes on the head pointer, the head node, the tail pointer, and the tail node. If no changes are being made to the queue at this time, then it will attempt to swing the head to its next node. The program will continuously attempt the dequeue operation until it succeeds.

See Figure 2 for the code of Michael Scott Queue.

2.3 linearisability

Linearisability is a correctness condition in concurrent computing that ensures operations on shared data appear to happen instantaneously at some single point between their invocation and their response. The specific moment when an operation appears to take effect is called the "linearisation point". This allows the concurrent behaviour to be matched to a sequential behaviour, thereby allowing programmers to specify and reason about concurrent objects using known techniques from the sequential domain [HW90].

For Michael Scott Queue, the linearisation point for its enqueue operation is when *CAS* operation at *E9* returns true; the linearisation point for its dequeue operation is the *CAS* operation at *D12* returns true. As the queue operations may delay its linearisation point, the abstract representation may not match expectations. Consider the following example:

```
enqueue(A)      (attempting to insert A)
|   enqueue(B)  (successfully inserts B)
|   dequeue()   (successfully removes B)
enqueue(A)      (successfully inserts A)
```

enqueue(A) was called first but was delayed by *enqueue(B)* and *dequeue()* on a separate thread, resulting in the final queue being *[A]*. The above may appear as an invalid behaviour, as the dequeue operation removed the tail of the queue. However, by considering each operation's linearisation points, the above can be matched to the following sequential behaviour, and would be considered valid:

```
enqueue(B)
dequeue()
enqueue(A)
```

As seen in the following sections of this paper, proof derived from rely/guarantee reasoning may demonstrate that linearisability is also satisfied.

3 Abstract Specification

On the abstract level, the state of the queue can be described as a list of values:

$\Sigma_0 :: list : Val^*$

Enqueue, at this level, can be treated as an atomic operation:

$enqueue_{abst\text{-}seq}$ (v: Val)
pre true
post $list' = list \frown [v]$

At the next stage of development, the model development was taken from [Yat24], where operations are still operating on an abstract data structure but are no longer atomic. Interference at this level means that it becomes difficult to state the postcondition, as the environment is free to modify the queue. The possible value notation, $\widehat{x}$, could be useful to describe the postcondition. It denotes a set of values x held during the program's execution [JH16]. The use of the possible value notation here allows the postcondition to state that at some point during execution, the value should have been inserted into the back of the sequence.

post-enqueue: $\exists l_1, l_2 \in \widehat{list} \cdot l_2 = l_1 \frown [v]$

However, this postcondition does not prevent the program from making further changes to the queue. Instead, this can be instated by the guarantee condition, which provides reasoning before and after any sequence of program operations. Consider the abstract sequential postcondition of enqueue:

post-enqueue: $list' = list \frown [v]$

With the introduction of concurrency, the program should behave under the requirements of linearisability, where each program step should either make no changes to the state of the system or match the state to the behaviour of some abstract postcondition if the program step is a linearization point. The guarantee condition, therefore, must state that each step either makes no changes or match the sequential postcondition.

guar-enqueue: $list' = list \lor list' = list' = list \frown [v]$

There is still the issue of multiple insertion, as the current specification does not prevent enqueue from repeatedly adding items. Similar issues were encountered by [Yat24], where they proposed using a *done* flag to keep track of whether a removal has occurred or not. The same technique can be used here to prevent

multiple insertions.

guar-enqueue: $(list' = list \Rightarrow done' = done) \wedge (list' \neq list \Rightarrow list' = list \frown [v] \vee done' \vee \neg done)$

To ensure the correct behaviour of the queue, it is essential to place requirements not only on the operations themselves but also on the environment in which they operate. The purpose of the queue is for the *dequeue* operation to consistently remove the earliest inserted item, while *enqueue* supports this by adding new items to the end of the queue. However, if the environment behaves incorrectly, such as by erroneously inserting or removing items, the queue may fail to operate as intended. Therefore, it is crucial to specify that the environment must uphold the conditions necessary for every concurrent *enqueue* and *dequeue* to perform according to specification.

However, the precondition alone cannot guarantee this. An error could happen at any point before or during the execution of an *enqueue* and the system has no way of distinguishing between a faulty operation and a series of correct operations that lead to the same outcome. This is where the rely condition applies: it ensures that every operation within the environment adheres to the expected behaviour, preventing any invalid actions from compromising the integrity of the queue.

The full *enqueue* operation is:

$enqueue_{abst\text{-}con}$ (v: Val)
pre $\neg done$
rely $done' = done \wedge$
$\quad \forall v \in list' \cdot v \notin list \Rightarrow \exists l_1, l_2 \in \widehat{list} \cdot l_2 = l_1 \frown [v] \wedge$
$\quad \forall v \in list \cdot v \notin list' \Rightarrow \exists l_1, l_2 \in \widehat{list} \cdot \mathbf{hd} l_1 = v \wedge l_2 = \mathbf{tl} l_1$
guar $(list' = list \Rightarrow done' = done) \wedge$
$\quad (list' \neq list \Rightarrow list' = list \frown [v] \wedge done' \wedge \neg done)$
post $done' \wedge \exists l_1, l_2 \in \widehat{list} \cdot l_2 = l_1 \frown [v]$

4 Sequential Development

The goal of the next level of refinement is to model the code given in Figure 2. This is performed by modelling the state of the queue to contain the heap, the head pointer, and the tail pointer. The heap is modelled as a finite constructed function mapping a pointer to a value-pointer pair, taking inspiration from [JY15]; the pointer is modified to include an integer counter. This counter models the counter in [MS96], which was used to prevent the ABA problem.

The following proofs will be using the Vienna Development Method (VDM), a long-established formal method used in software development to describe and model systems mathematically. For an overview see [Jon90]

$Ptr\_t = (ptr\colon Pointer \times count\colon Int)$
$Heap = Pointer \to (value\colon Val \times next\colon [Ptr\_t])$

(In VDM, the square brackets indicates that Ptr_t can be **nil**. The fields of a pair $pr \in (Val \times Ptr\_t)$ are accessed by index, e.g. pr_2.)

$$\begin{array}{lll} \Sigma_1 :: & heap & : [Heap] \\ & head & : [Ptr\_t] \\ & tail & : [Ptr\_t] \end{array}$$

It may be tempting to refine directly to a concrete concurrent specification. However, [JY19] has shown that making a quick detour and first attempting to refine the abstract into a concrete sequential specification may provide valuable insights and facilitate subsequent efforts. In light of this advice, this paper will begin by refining the abstract specification into a concrete sequential specification, and perform the concurrent development afterwards.

The specification for the concrete sequential enqueue is:

$enqueue_{real\text{-}seq}\ (v\colon Val)$
pre $True$
post $\exists P \in Ptr\_t \cdot heap' = \{(heap(tail_1)_2)_1 \to (V, P)\} \cup heap$
$\quad \wedge\ tail'_2 = tail_2 + 1$
$\quad \wedge\ tail'_1 = (heap(tail_1)_2)_1$
$\quad \wedge\ head' = head$

The *done* flag and rely/guarantee conditions are not required, as this is the sequential specification. The first clause of the postcondition updates the heap to include the new value. P represents a newly allocated memory address such as the ones provided by *malloc*. The Second and third clause regulates the tail pointer. The tail pointer's counter should increase, and the tail pointer should point to the tail after the *enqueue* operation. The final clause states that the head should not be changed by the *enqueue* operation.

The next step is to relate this concrete specification to the abstract specification. Proof obligations for data reification are standard in VDM [Jon90]. A retrieve function is required to correspond a given concrete state to an abstract state:

$retr_1 : \Sigma_1 \to \Sigma_0$
$retr_1((heap, head, tail)) \quad \triangleq \quad trace(heap, head)$

where

$$
\begin{array}{l}
trace : (Heap \times Ptr\_t)) \to \Sigma_0 \\
trace((heap, head)) \quad \triangleq \quad \textbf{if } heap(head_1) = \textbf{nil} \\
\qquad\qquad\qquad\qquad\quad \textbf{then } [\,] \\
\qquad\qquad\qquad\qquad\quad \textbf{else } [heap(head_1)_1] \frown trace(heap, heap(head_1)_2)
\end{array}
$$

The retrieve function is a recursive trace function that follows the pointers and recursively collects the values in the heap into an array.

VDM defines three proof obligations required for data reification: Adequacy, Domain, and Result.

Lemma 1: Adequacy states that for all abstract states, there exists at least one corresponding concrete state:

$$\forall a \in \Sigma_0 \cdot \exists r \in \Sigma_1 \cdot retr_1(r) = a$$

The proof is by induction over Σ_0. [1]

Lemma 2: Domain states that if the abstract precondition holds over an abstract state, then the concrete precondition should also hold over its corresponding concrete state:

$$\forall r \in \Sigma_1 \cdot pre_{abst\text{-}seq}(retr_1(r)) \;\Rightarrow\; pre_{real\text{-}seq}(r)$$

The proof is trivial, as the preconditions of abstract and concrete representation are *True*.

Lemma 3: Result states that the concrete postcondition should models the abstract postcondition, under the retrieve function:

$$\forall r, r' \in \Sigma_1 \cdot pre_{abst\text{-}seq}(retr_1(r)) \wedge post_{real\text{-}seq}(r, r') \;\Rightarrow\; post_{abst\text{-}seq}(retr_1(r), retr_1(r'))$$

The proof is by expanding the definitions of functions.

5 Concurrent Development

With sequential development serving as a guide, the next step is to develop the abstract representation into a concrete concurrent representation. At this level, heap operations are considered atomic, therefore there is no need to design a new heap to model the concurrent operations.

The specification for the concrete concurrent enqueue is (where done is a local variable):

[1] For the full proofs, please refer to the Appendix.

$enqueue_{real\text{-}con}$ (v: Val)

pre $\neg done$

rely $done' = done$
$\quad \wedge (\forall v \in retr_1(heap') \cdot v \notin retr_1(heap))$
$\quad \Rightarrow \ (\exists l_1, l_2 \in \overbrace{retr_1(heap)} \cdot l_2 = l_1 \frown [v])$
$\quad \wedge (\forall v \in retr_1(heap) \cdot v \notin retr_1(heap'))$
$\quad \Rightarrow \ (\exists l_1, l_2 \in \overbrace{retr_1(heap)} \cdot \mathbf{hd} l_1 = v \wedge l_2 = \mathbf{tl} l_1)$

guar $(heap' = heap \ \Rightarrow \ done' = done)$
$\quad \wedge (heap' \neq heap$
$\quad \Rightarrow \ retr_1(heap') = retr_1(heap) \frown [v] \wedge done' \wedge \neg done)$

post $done' \wedge \exists (heap_a, head_a, tail_a), (heap_b, head_b, tail_b) \in \overbrace{(heap', head', tail')} \cdot$
$\quad \exists P \in Ptr\_t \cdot heap_b = (\{heap_a(tail_{a1})_2)_1 \rightarrow (V, P)\} \cup heap_a$
$\quad \wedge tail_{b2} > tail_{a2}$
$\quad \wedge tail_{b1} = (heap_a(tail_{a1})_2)_1$
$\quad \wedge head_a = head_b$

Using the retrieve function, the rely and guarantee conditions remain mostly unchanged from their abstract counterpart. The rely condition ensures that every environmental step acting on the queue is either a valid enqueue or a valid dequeue, while the guarantee condition ensures linearisability, that every program step either makes no changes or matches the sequential postcondition.

Sequential development has shown that describing the postcondition with the content of the heap allows for simpler proof obligations. The postcondition therefore states that at some point, when the program passes its linearisation point, the heap would be updated such that it matches that of a sequential implementation, where the previous tail item's pointer now points to a newly inserted item, and the tail has been updated. The head remains unchanged.

Two methods of performing the refinement into concurrency were identified. One is to continue from the sequential development by introducing concurrency with another layer, named the *Indirect* approach. The other is to refine directly from the abstract concurrent representation, named the *Direct* approach. Both methods were explored.

5.1 Indirect Approach

For the indirect approach, the sequential development in Section 4 will now serve as the new "abstract" specification, and the goal is to refine it to introduce concurrency. As the state representation is not changed, the retrieve function is the identity function.

$retr_2 : \Sigma_1 \rightarrow \Sigma_1$

$retr_2(a) \quad \triangleq \quad a$

The following VDM proof obligation stands:

Lemma 4: For all abstract states, there exists at least one corresponding concrete state:

$$\forall a \in \Sigma_1 \cdot \exists r \in \Sigma_1 \cdot retr_2(r) = a$$

The proof is trivial, as retrieve is the identify function.

Lemma 5: If the abstract precondition holds over an abstract state, then the concrete precondition should also hold over its corresponding concrete state:

$$\forall r \in \Sigma_1 \cdot pre_{real\text{-}seq}(retr_1(r)) \ \Rightarrow \ pre_{real\text{-}con}(r)$$

The lemma is obviously false, as $pre_{real\text{-}seq}$ is $True$, while $pre_{real\text{-}con}$ is $\neg done$.

At this point, the indirect approach should be abandoned. To investigate the properties of rely/guarantee, an attempt was made on the final proof obligation regardless.

Lemma 6: The concrete postcondition should models the abstract postcondition, under the retrieve function:

$$\forall r, r' \in \Sigma_1 \cdot pre_{real\text{-}seq}(retr_1(r)) \wedge post_{real\text{-}con}(r, r') \ \Rightarrow \ post_{real\text{-}seq}(retr_1(r), retr_1(r'))$$

The proof is troublesome, as $post_{real\text{-}con}$ is weaker than $post_{real\text{-}seq}$. This is because the concurrent specification has to accommodate the interference brought by the environment. Take the tail pointer's counter, for example, the concurrent specification can only state that the value of the tail pointer has increased, due to the environment possibly performing enqueue during the program's execution. However, on the sequential version, one can confidently state the value of the counter after enqueue. The weakening of postcondition after introducing concurrency was also noticed in [JY19].

5.2 Direct Approach

For the direct approach, the goal is to refine the abstract concurrent development in Section 3 to the concrete level and introduce concurrency. The state representation changes from Σ_0 to Σ_1, which is the same as the sequential development. This suggests that the same retrieve function from Section 4 can be used.

The following VDM proof obligation stands:

Lemma 7: For all abstract states, there exists at least one corresponding concrete state:

$$\forall a \in \Sigma_0 \cdot \exists r \in \Sigma_1 \cdot retr_1(r) = a$$

As no changes is made on the state representation and retrieve function, the proof follows that of Lemma 1.

Lemma 8: If the abstract precondition holds over an abstract state, then the concrete precondition should also hold over its corresponding concrete state:

$$\forall r \in \Sigma_1 \cdot pre_{abst\text{-}con}(retr_1(r)) \;\Rightarrow\; pre_{real\text{-}con}(r)$$

The proof is obvious as $pre_{abst\text{-}con}$ and $pre_{real\text{-}con}$ are the same ($\neg done$)

Lemma 9: The concrete postcondition should models the abstract postcondition, under the retrieve function:

$$\forall r, r' \in \Sigma_1 \cdot pre_{abst\text{-}con}(retr_1(r)) \wedge post_{real\text{-}con}(r, r') \;\Rightarrow\; post_{abst\text{-}con}(retr_1(r), retr_1(r'))$$

The incomplete proof is similar in structure to Lemma 3, but issues exist with the use of possible values. To infer the abstract postcondition, some relationship between the possible value of the abstract queue and the possible value of the concrete queue is required. Specifically, knowing that $r_a, r_b \in \widehat{r}$, we need to show how $retr(r_a), retr(r_b) \in \widehat{retr(r)}$.

As the possible value is a set, intuitively we can utilise the retrieve function and apply it to every item in $\widehat{r}$, resulting in the set $\{\forall x \in \widehat{r} \cdot retr(x)\}$. However, it is difficult to show that the result set is equivalent to $\widehat{retr(r)}$. If it is possible, the proof can be easily completed by the image of set $\widehat{r}$ under function $retr$.

6 Observations

Many issues encountered in Section 3 are similar to those found in [Yat24], where rely/guarantee was used to verify the Treiber Stack and the Herlihy-Wing Queue. The abstract specification in Section 3 was developed using the same ideas and similar methods that were first presented in [Yat24], suggesting some universality in their findings – that there is a class of algorithms for which the standard rely/guarantee is insufficient for reasoning, and possible value appears to be a way to correctly specify these algorithms.

The sequential development in Section 4 has provided valuable insights and aided in concurrent developments. The simpler conditions made it easier to design and reason about the heap and its retrieve function. The proofs for sequential data reification share similar structures to their concurrent counterpart. The benefit of sequential development to concurrency reasoning was also noted in [JY19], where their sequential development allows them to observe how the rely condition and the guarantee condition must be adapted to handle concurrency

The concurrent development in Section 5 was ultimately not completed, yet it has made some interesting observations. The indirect approach demonstrated a weakening of postcondition when concurrency is introduced. This was also

observed in [JY19], where this weakening helped them in constructing postconditions. The direct approach has shown difficulties with possible value in proof, yet it is the most promising approach. [JH16] did provide laws regarding the use of possible value, and future work could further investigate the working of possible value in data reification.

7 Related Work

Linearisability has been extensively studied using various approaches. A comprehensive survey of these approaches is provided in [DD15]. Most approaches traditionally rely on simulations to establish the relationship between the concrete and abstract states. For example, [CDG05] utilises two I/O automata for a forward simulation proof to verify a simple lock-free stack.[DGLM04] utilise a similar method, employing forward and backward simulation to verify an optimised version of Michael Scott Queue.

An alternative approach is presented in [EQS+10], where a coarse-grained abstraction is constructed from finer-grained implementation through reduction and abstraction. This approach ensures trace-equivalence between the abstraction and implementation.

In [DD13], the authors use an interval-based logic to link concrete operations over any interval to corresponding abstractions. This avoids the challenge of identifying linearisation points within the code. The absence of the need to explicitly identify the linearisation point is reminiscent of this paper and [Yat24].

Rely/guarantee has been combined with other logic to successfully verify linearisability. In [Vaf08], the author introduced RGSep, which combines rely/guarantee and separation logic to verify linearisability. Their method requires the identification and capture of linearisation points and does not explore using guarantee conditions to express linearisable behaviour. In [TSR14], a proof method which combines temporal logic, rely-guarantee reasoning and possibilities is introduced. This method requires each process to preserve possibility steps as an additional guarantee condition for linearisability.

8 Conclusion

This paper explored the use of rely/guarantee reasoning in the verification of non-blocking concurrent algorithms, specifically focusing on the Michael Scott Queue. Through the sequential and concurrent development of an abstract model, we found evidence suggesting that there exists a class of algorithms for which the standard rely/guarantee approach is limited. We also notice that it is possible to state the linearisability property in terms of the guarantee condition, which provides an additional framework for reasoning about the correctness of concurrent operations. Our findings align with previous work, such as [Yat24],

We observed that introducing concurrency weakened the postconditions compared to the sequential model, a phenomenon also noted in [JY19]. The use of

possible value also posed challenges in data reification, one which further investigation may overcome. Despite these challenges, rely/guarantee still offers valuable insights.

References

[CDG05] Robert Colvin, Simon Doherty, and Lindsay Groves. Verifying concurrent data structures by simulation. *Electronic Notes in Theoretical Computer Science*, 137(2):93–110, 2005.

[DD13] Brijesh Dongol and John Derrick. Simplifying proofs of linearisability using layers of abstraction. *arXiv preprint arXiv:1307.6958*, 2013.

[DD15] Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM Computing Surveys (CSUR)*, 48(2):1–43, 2015.

[DGLM04] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *Formal Techniques for Networked and Distributed Systems–FORTE 2004: 24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004. Proceedings 24*, pages 97–114. Springer, 2004.

[EQS+10] Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems: 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 16*, pages 296–311. Springer, 2010.

[HJ17] Ian J Hayes and Cliff B Jones. A guide to rely/guarantee thinking. *School on Engineering Trustworthy Software Systems*, pages 1–38, 2017.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[JB24] Cliff B Jones and Alan Burns. Extending rely-guarantee thinking to handle real-time scheduling. *Formal Methods in System Design*, 62(1):119–140, 2024.

[JH16] Cliff B Jones and Ian J Hayes. Possible values: exploring a concept for concurrency. *Journal of Logical and Algebraic Methods in Programming*, 85(5):972–984, 2016.

[Jon83a] Cliff B Jones. Specification and design of (parallel) programs. In *9th IFIP World Computer Congress (Information Processing 83)*. Newcastle University, 1983.

[Jon83b] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[Jon90] Cliff B Jones. *Systematic software development using VDM*, volume 2. Prentice Hall Englewood Cliffs, 1990.

[JY15] Cliff B Jones and Nisansala Yatapanage. Reasoning about separation using abstraction and reification. In *Software Engineering and Formal Methods: 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, pages 3–19. Springer, 2015.

[JY19] Cliff B Jones and Nisansala Yatapanage. Investigating the limits of rely/guarantee relations based on a concurrent garbage collector example. *Formal Aspects of Computing*, 31:353–374, 2019.

[MS96] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[TSR14] Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. A compositional proof method for linearizability applied to a wait-free multiset. In *Integrated Formal Methods: 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings 11*, pages 357–372. Springer, 2014.

[Vaf08] Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical report, University of Cambridge, Computer Laboratory, 2008.

[Yat24] Nisansala P Yatapanage. Exploring the boundaries of rely/guarantee and links to linearisability. In *The Practice of Formal Methods: Essays in Honour of Cliff Jones, Part II*, pages 247–267. Springer, 2024.

Appendix

Lemma 1 states:

$$\forall a \in \Sigma_0 \cdot \exists r \in \Sigma_1 \cdot retr_1(r) = a$$

The proof is by induction over Σ_0.

from	$a \in Val^*$	
1	**from** $a = [\,]$	
1.1	$(\mathbf{nil}, \mathbf{nil}, \mathbf{nil}) \in \Sigma_1$	Σ_1
1.2	$retr_1((\mathbf{nil}, \mathbf{nil}, \mathbf{nil})) = [\,]$	$retr_1$
	infer $\exists r \in \Sigma_1 \cdot retr_1(r) = a$	1.1, 1.2
2	**from** $a \in Val^*; \exists r \in \Sigma_1 \cdot retr_1(r) = a$	IH
2.1	$r = (heap, head, tail)$	assume
2.2	$V \in Val \wedge P \in Pointer$	assume
2.3	$h' = \{P \to (V, head)\} \cup heap$	assume
2.4	$(h', P, P) \in \Sigma_1$	Σ_1
2.5	$retr_1((h', P, P)) = [v] \overset{\frown}{} retr_1(r)$	2.3, $retr_1$
	infer $\exists r' \in \Sigma_1 \cdot retr_1(r') = [v] \overset{\frown}{} a$	2.4, 2.5
infer	$\exists r \in \Sigma_1 \cdot retr_1(r) = a$	sequence-indn(1, 2)

Lemma 2 states:

$$\forall r \in \Sigma_1 \cdot pre_{abst\text{-}seq}(retr_1(r)) \;\Rightarrow\; pre_{real\text{-}seq}(r)$$

The proof is trivial, as $pre_{seq}(r)$ is *True*

Lemma 3 states:

$$\forall r, r' \in \Sigma_1 \cdot pre_{abst\text{-}seq}(retr_1(r)) \wedge post_{real\text{-}seq}(r, r') \;\Rightarrow\; post_{abst\text{-}seq}(retr_1(r), retr_1(r'))$$

The proof is by expanding the definitions of functions.

from $pre_{abst\text{-}seq}(retr_1(r)) \wedge post_{real\text{-}seq}(r, r')$

1 $r = (heap, head, tail) \wedge r' = (heap', head', tail')$ — assume

2 $\exists P \in Ptr\_t \cdot heap' = \{(heap(tail_1)_2)_1 \rightarrow (V, P)\} \cup heap$
$\wedge tail'_1 = (heap(tail_1)_2)_1 \wedge head' = head$ — $post_{real\text{-}seq}$

3 $heap'(tail'_1) = (V, P)$ — 2

4 $retr_1(r) = trace(heap, head)$
$= [heap(head_1)_1] \frown [(heap(heap(head_1)_2)_1)_1]...$
$[heap(tail_1)_1] \frown [\,]$ — $retr_1$, trace

5 $retr_1(r') = trace(heap', head')$
$= [heap'(head'_1)_1] \frown [(heap'(heap'(head'_1)_2)_1)_1]...$
$[heap'(tail'_1)_1] \frown [\,]$
$= [heap(head_1)_1] \frown [(heap(heap(head_1)_2)_1)_1]...$
$[heap(tail_1)_1] \frown [heap'(tail'_1)_1] \frown [\,]$
$= retr_1(r) \frown [V] \frown [\,]$ — 2, 3, 4, $retr_1$, trace

infer $post_{abst\text{-}seq}(retr_1(r), retr_1(r'))$ — 5

Lemma 4 states:

$$\forall a \in \Sigma_1 \cdot \exists r \in \Sigma_1 \cdot retr_2(r) = a$$

The proof is trivial, as retrieve is the identity function.

Lemma 5 states:

$$\forall r \in \Sigma_1 \cdot pre_{real\text{-}seq}(retr_1(r)) \;\Rightarrow\; pre_{real\text{-}con}(r)$$

The lemma is false, as $pre_{real\text{-}seq}$ is *True*, while $pre_{real\text{-}con}$ is $\neg done$.

Lemma 6 states:

$$\forall r, r' \in \Sigma_1 \cdot pre_{real\text{-}seq}(retr_1(r)) \wedge post_{real\text{-}con}(r, r') \;\Rightarrow\; post_{real\text{-}seq}(retr_1(r), retr_1(r'))$$

The proof appears impossible due to reasons discussed in Section 5.1.

Lemma 7 states:

$$\forall a \in \Sigma_1 \cdot \exists r \in \Sigma_1 \cdot retr_1(r) = a$$

The proof follows that of Lemma 1.

Lemma 8 states:

$$\forall r \in \Sigma_1 \cdot pre_{abst\text{-}con}(retr_1(r)) \;\Rightarrow\; pre_{real\text{-}con}(r)$$

The proof is obvious as $pre_{abst\text{-}con}$ and $pre_{real\text{-}con}$ are the same ($\neg done$)

Lemma 9 states:

$$\forall r, r' \in \Sigma_1 \cdot pre_{abst\text{-}con}(retr_1(r)) \wedge post_{real\text{-}con}(r, r') \;\Rightarrow\; post_{abst\text{-}con}(retr_1(r), retr_1(r'))$$

The proof is incomplete, due to critical assumptions made in step 5, 6. See Section 5.2 for more details.

from	$pre_{abst\text{-}con}(retr_1(r)) \wedge post_{real\text{-}con}(r, r')$	
1	$done' \wedge \exists (heap_a, head_a, tail_a), (heap_b, head_b, tail_b) \in \widehat{r'} \cdot$ $\exists P \in Ptr\_t \cdot heap_b = \{(heap_a(tail_{a1})_2)_1 \rightarrow (V, P)\} \cup heap_a$ $\wedge tail_{b2} > tail_{a2}$ $\wedge tail_{b1} = (heap_a(tail_{a1})_2)_1$ $\wedge head_a = head_b$	$post_{real\text{-}con}$
2	$heap_b(tail_{b1}) = (V, P)$	1
3	$retr_1((heap_a, head_a, tail_a)) = trace(heap_a, head_a)$ $= [heap_a(head_{a1})_1] \frown [(heap_a(heap_a(head_{a1})_2)_1)_1] ...$ $[heap_a(tail_{a1})_1] \frown [\,]$	$retr_1$, trace
4	$retr_1((heap_b, head_b, tail_b)) = trace(heap_b, head_b)$ $= [heap_b(head_{b1})_1] \frown [(heap_b(heap_b(head_{b1})_2)_1)_1] ...$ $[heap_b(tail_{b1})_1] \frown [\,]$ $= [heap_a(head_{a1})_1] \frown [(heap_a(heap_a(head_{a1})_2)_1)_1] ...$ $[heap_a(tail_{a1})_1] \frown [heap_b(tail_{b1})_1] \frown [\,]$ $= retr_1((heap_a, head_a, tail_a)) \frown [v] \frown [\,]$ 1, 2, 3	$retr_1$, trace
5	$retr_1((heap_a, head_a, tail_a)) \in \widehat{retr_1(r)}$	assume
6	$retr_1((heap_b, head_b, tail_b)) \in \widehat{retr_1(r)}$	assume
7	$\exists retr_1((heap_a, head_a, tail_a)), retr_1((heap_b, head_b, tail_b)) \in \widehat{retr_1(r)} \cdot$ $retr_1((heap_a, head_a, tail_a)) \frown [v]$	4 5, 6
infer	$post_{abst\text{-}con}(retr_1(r), retr_1(r'))$	1, 7

```
structure pointer_t {ptr: pointer to node t, count: unsigned integer}
structure node_t {value: data type, next: pointer t}
structure queue_t {Head: pointer t, Tail: pointer t}

initialize(Q: pointer to queue_t)
    node = new_node()
    node->next.ptr = NULL
    Q->Head.ptr = Q->Tail.ptr = node

enqueue(Q: pointer to queue_t, value: data type)
E1:    node = new node()
E2:    node->value = value
E3:    node->next.ptr = NULL
E4:    loop
E5:        tail = Q->Tail
E6:        next = tail.ptr->next
E7:        if tail == Q->Tail
E8:            if next.ptr == NULL
E9:                if CAS(&tail.ptr->next, next, <node, next.count+1>)
E10:                   break
E11:               endif
E12:           else
E13:               CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
E14:           endif
E15:       endif
E16:   endloop
E17:   CAS(&Q->Tail, tail, <node, tail.count+1>)

dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D1:    loop
D2:        head = Q->Head
D3:        tail = Q->Tail
D4:        next = head.ptr->next
D5:        if head == Q->Head
D6:            if head.ptr == tail.ptr
D7:                if next.ptr == NULL
D8:                    return FALSE
D9:                endif
D10:               CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
D11:           else
D12:               *pvalue = next.ptr->value
D13:               if CAS(&Q->Head, head, <next.ptr, head.count+1>)
D14:                   break
D15:               endif
D16:           endif
D17:       endif
D18:   endloop
D19:   free(head.ptr)
D20:  return TRUE
```


Figure 2: The code for the Michael Scott Queue [MS96]