Ownership-Based Owicki-Gries Reasoning

Mikhail Semenyuk
Brijesh Dongol
m.semenyuk@surrey.ac.uk
b.dongol@surrey.ac.uk
University of Surrey
Guildford, Surrey, UK

Abstract

This paper explores the use of ownership as a key auxiliary variable within Owicki-Gries framework. We explore this idea in the context of a ring buffer, developed by Amazon, which is a partial library that only provides methods for reserving (acquiring) and releasing addresses within the buffer. A client is required to implement additional functionality (including an additional shared queue) to enable full synchronisation between a writer and a reader. We verify correctness of the Amazon ring buffer (ARB) and show that the ARB satisfies both safety and progress. Our proofs are fully mechanised within the Isabelle/HOL proof assistant.

Keywords: Owicki-Gries, ownership, ring buffer, verification, safety, progress

ACM Reference Format:
Mikhail Semenyuk and Brijesh Dongol. 2023. Ownership-Based Owicki-Gries Reasoning. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3555776.3577636

1 Introduction

Verification of concurrent programs is difficult when resources have to be shared between multiple threads. A simple system with a single writer and reader pair should execute without interference, provided that an appropriate synchronisation mechanism, e.g., queuing or access control is introduced. However, naive models of such systems are often inadequate, and may require extensive fine-tuning before verification is successful.

The main mechanism we use to address this issue is a systematic auxiliary encoding of *ownership* within an Owicki-Gries framework [24]. We draw analogy here with the fact that standard Owicki-Gries style verification often utilises program counters as auxiliary variables, which are already used systematically, relieving a verifier from spending unnecessary time modelling and reasoning about control flow. At a high level, ownership enables one to reason abstractly over a group of components accessing a shared resource as opposed to individual components. For example, by only granting read and write capabilities to a resource when it is owned, one can assume mutual exclusivity (from other threads), during periods of ownership. Detailed consideration of inter-thread interaction only needs to be addressed during periods of ownership transfer.

We aim to demonstrate how use of ownership can aid verification of an even less rigid program implementation - a partial library. Unlike complete implementations, partial libraries can often require more abstraction with respect to input and output variables (before attempting to define the invariants), making the task of verification relatively harder. An example of this is our motivating example [1] (see Section 3), which is a ring buffer developed at Amazon. The library only provides methods for acquiring and releasing addresses within a shared buffer. A client writer is required to write to the acquired addresses, then transfer the acquired addresses to the reader using an external mechanism (in our example we use a queue). A client reader is required to obtain addresses from this external mechanism, read the data within these addresses, then release the addresses of the buffer. On the one hand, this provides flexibility since the exact implementation of the external mechanism is left to the client (and may change between different clients). On the other hand, there is more work left to the client. In this paper, we focus on demonstrating how adopting an ownership-based approach can aid formal verification of safety and progress properties of such programs.

Preliminary ideas for encoding a notion of resources in an Owicki-Gries system were in fact proposed by Owicki [23], but without detailed proofs. The idea of using ownership to track resources has since been extensively used in object-oriented language semantics [2–4, 10–12, 16, 20, 26] and separation logic [5, 6, 8, 9, 15, 17, 21, 22, 25, 29–32] (a review of these works is provided in Section 6). We ask: Is there a way to utilise the idea of ownership and its transfer using a more direct, global approach, as suggested by Owicki-Gries,

SAC '23, March 27-31, 2023, Tallinn, Estonia
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3577636

to make both informal and formal arguments shorter while maintaining strong invariants?

Contributions. This paper contains three contributions.

1. A technique for encoding ownership of shared resources within an Owicki-Gries framework.
2. Application of this framework to verifying safety and progress properties of a partial implementation. We show that ownership can be readily extended to cover any remaining components needed to complete a partial implementation, using a real-world algorithm developed at Amazon.
3. Mechanisation of the proofs of both safety and progress using Isabelle/HOL [27].

Overview. In Section 2 we introduce and present a technique for encoding ownership and ownership transfer within an Owicki-Gries framework. In Section 3, we present our running example, the Amazon Ring Buffer, and encode Ownership within the model in Section 4. In Section 5, we present our proofs of safety and progress, using the ownership mechanism and describe our mechanisation effort. Related work, which includes a brief survey of ownership in programming languages and verification, is presented in Section 6. Conclusions and future work is given in Section 7.

2 Overview and Initialisation of Ownership

The Owicki and Gries framework [24] extends Hoare logic to cover shared variable concurrent programs. To achieve this, they extend reasoning over preconditions (*pre*) and postconditions (*post*) with a notion of *interference freedom*, which is a property demonstrated by a program when the steps taken by either of its processes do not violate the preconditions of the other processes. Owicki-Gries' logic is known to be incomplete without *auxiliary variables*, which are variables that record the history of a program's execution. Typically, auxiliary variables are used to track program flow (program counters) as well as monitoring accesses to shared variables (which we refer to as *shared resources*) by potentially conflicting processes.

We wish to systematise the latter with the idea of *ownership* and *ownership transfer*. Ownership is an auxiliary variable, which represents and can be used to track access rights to a resource. These access rights can vary from read-only to read and write, depending on the needs of a verifier. For example, a thread having ownership over an address can modify and/or read the contents, while being the only thread allowed to perform these operations. *Ownership transfer* refers to changing the value of the ownership variable, e.g. revoking access rights from one owner (represented by the old ownership value) and granting them to another.

Our key observation in this paper is that careful management of ownership transfer enables verification of interference freedom. A thread must not be allowed to alter the contents of a shared resource, when that resource is currently in

```
{$own_a = None \wedge f = 0 \wedge d = 0 \wedge a = None \wedge r = 0$}
$T_1$
    {$own_a = None \wedge f = 0 \wedge d = 0 \wedge a = None$}
    int a ; own(a) := $T_1$
    {$own_a = T_1 \wedge f = 0 \wedge d = 0 \wedge a \neq None$}
    a := 42 ;
    {$own_a = T_1 \wedge f = 0 \wedge d = 0 \wedge a = 42$}
    f := 1 ; own(a) := $None$
    {$True$}
||
$T_2$
    {$Q$}
    d := f
    {$Q$}
    until d = 1
    (+ own(a) := $T_2$)
    {$own_a = T_2 \wedge a = 42$}
    r := a ;
    {$r = 42$}
{$r = 42$}
```

Figure 1. Message passing with ownership

the process of being read or written to by another thread. To allow for safe (interference-free) behaviour, threads have to acquire access rights to a resource, before they can perform reading or modifying operations to it. Changing ownership over a resource from one thread directly to another can lead to contradictions. Instead, an intermediate, *transient owners*, must be used to avoid direct thread-to-thread ownership transfer. Transient owners are an auxiliary type of ownership value, which facilitate safe ownership transfer. A thread should *let go* of ownership by changing the corresponding ownership value to *None* or any other transient owner, before another thread can *take* ownership, changing the ownership value to itself.

We motivate our ideas with a simple concurrent system of two threads in Fig. 1, where T_1 and T_2 are two threads, accessing a shared resource a, and:

$$Q \equiv (f = 1 \longrightarrow own_a \neq T_1) \wedge (f = 0 \longrightarrow d = 0) \\ \wedge (own_a \neq T_1 \wedge a \neq None \longrightarrow a = 42)$$

Thread T_1 updates a, then notifies T_2 that it has finished by updating the flag f to 1. Thread T_2 on the other hand waits for the flag to be set before proceeding to read a. The desired safety property is that $r = 42$ after both threads have terminated (as shown in the example).

From the point of view of ownership, while T_1 is in the process of modifying variable a, we must maintain that it is the only thread with read and write read-write access to the variable. Similarly, while T_2 performs r:=a, it should be the only thread with $R{+}W$ access. We introduce the ownership variable own_a to track these access rights to the shared resource a. This variable is tracked via *pre* and *post* assertions, coloured blue in the example.

$$own_a = y \ , \quad \text{where } y \in \{T_1, T_2, None\}$$

To enable *ownership transfer* we introduce the auxiliary function own(a):=y (coloured red in the example):

$$\texttt{own(a):=y}, \ \text{ where } y \in \{T_1, T_2, None\}$$

Thread T_1 initialises a, acquiring ownership over the resource from *None* (a transient owner). T_1 "lets go" of its ownership over the resource when performing the assignment operation of flag f, changing the value of the ownership variable to *None*. Thread T_2 takes ownership of a once it is satisfied that `d = 1`, at which point it can proceed to safely read the resource. We must observe, that before T_2 performs `own(a):=` T_2, its precondition demands $own_a \neq T_1$. In order to *restrict* this operation to be performed during the state transition of leaving the `until` loop only, we resort to decorating the ownership transfer operation with a "+". We describe other types of ownership and ownership transfer in Sections 4.1 and 4.2, respectively.

Observation of the fact, that hand-over of access rights to a during execution occurs only through transient owners, can help convince the verifier of interference freedom from a high-level analysis. The proof outline is given by *pre* and *post* assertions within the example, coloured blue.

3 Amazon's Ring Buffer

Partial libraries are often used to enable a modular approach to programming. Verification of these involves abstraction of types of input/output variables, ensuring correct execution (safety). In a concurrent setting, interference-freedom needs to be defined and verified alongside. Because of the need for this abstraction (rather than to be given concrete input/output variables) the verification process can be significantly slowed down. An example of such libraries include writer/reader methods with an external synchronisation method, such as a queue. The input/output spaces for both of the methods could be loosely defined, yet be intuitively trivial to demonstrate correct execution procedures from a high-level approach.

We motivate our ideas by considering a partial implementation of a ring buffer [1] developed at Amazon, whose pseudocode is given in Fig. 3. The full implementation must enable transfer of data (in a FIFO manner) from a writer to a reader. However, the library itself only defines the following methods:

- `acquire`, which is used to reserve a range of addresses in the buffer, and
- `release`, which is used to return reserved addresses back to the buffer.

The expectation is that the client's writer thread calls `acquire` to reserve addresses, then writes the data within the acquired addresses of the buffer. It then informs the reader of the bytes that should be read via a secondary shared FIFO queue whose precise implementation is left undefined. The client's reader thread must dequeue address ranges from the queue (one at a time and in FIFO order). For each address range that is dequeued, it must read the associated addresses of the buffer, then execute the `release` operation.

```
1  // Most general writer
2  i := 0;
3  while i <  n do
4    val := Data(i);
5    assert $size(val) \leq N$
6    m.reset();
7    x := B.acquire(\size(val)) ;
8    assert $\forall b \in bytes(x).\ own_B(b) = W \wedge own_D(\mathtt{i}) = W$
         $\wedge\ (h_W = t_W \longleftrightarrow own_T = W)$
9    if (x != OOM) then
10     B.write(x, val); transfer_ownD(i,B);
           data_index(x) := i
11     assert $\forall b \in bytes(x).\ own_B(b) = W \wedge own_D(\mathtt{i}) = B$
           $\wedge\ (h_W = t_W \longleftrightarrow own_T = W)$
12     Q.enqueue(x); transfer_ownB(i,W,Q);
           pass_ownT(W,Q)
13     assert $\forall b \in bytes(x).\ own_B(b) = Q \wedge own_D(\mathtt{i}) = B$
           $\wedge\ own_T \neq W$
14     i+=1
15   else m.backoff(); goto 6;
16
17 // Most general reader
18 j := 0;
19 while true do
20   if !Q.empty() then
21     assert $(\forall b \in bytes(x).\ own_B(b) = Q) \wedge own_D(\mathtt{j}) = B$
           $\wedge\ own_T = Q$
22     x := Q.dequeue(); take_ownB(bytes(head(Q)),R);
           pass_ownT(Q,R)
23     assert $(\forall b \in bytes(x).\ own_B(b) = R) \wedge own_D(\mathtt{j}) = B$
           $\wedge\ own_T = R$
24     val = B.read(x); transfer_ownD(data_index(x), R)
25     assert $(\forall b \in bytes(x).\ own_B(b) = R) \wedge own_D(\mathtt{j}) = R$
           $\wedge\ own_T = R$
26     B.release(x);
27     assert $(\forall b \in bytes(x).\ own_B(b) = B) \wedge own_D(\mathtt{j}) = R$
           $\wedge\ own_T \neq R$
28     j+=1;
```

Figure 2. Most general ring buffer client; statements in red and green are auxiliary statements used to model ownership

The *most general client* that uses the ring buffer is given in Fig. 2, where

$$bytes(x) = [fst(x), fst(x) + snd(x)]$$
$$data\_index : nat \times nat \longrightarrow nat$$

We assume that the writer W transfers information stored in a list *Data* to the reader R via the shared ring buffer B, using a shared queue Q. We also assume a timer object `m` that is reset at line 6, and performs a backoff at line 10, in which the object is used to guarantee progress.

The writer continually removes the elements from the *Data* list and transfers them to the reader R via the shared buffer. We assert that each element of *Data* to be transferred has size at most N = `B.size()` (so that it fits within the buffer). A writer attempts to acquire the bytes at line 7, which

```
1   // code for buffer B of size N
2   def init(x: nat)
3        N, H, T := x, 0, 0
4
5   def acquire(x: nat)
6        tW := T ; hW := H;          //$A_1$
7        if (tW == hW) && (x <= N) then
              (+ pass_ownT(Q,W))         //$A_2$
8             T, H, offset := 0, x, 0;
                  set_ownB([0,x],W)      //$A_3$
9        elif (tW > hW) && (x < = tW - hW -1) then
10            H,offset := hW + x,hW;
                  set_ownB([hW,hW+x],W)  //$A_4$
11       elif (tW <  hW) then
12          if (x < = N - hW) then     //$A_5$
13             H,offset := hW + x,hW;
                   set_ownB([0,x],W)     //$A_6$
14          elif (x <  tW) then
15             H, offset := x, 0; set_ownB([hW,N],D);
                   set_ownB([0,x],W)     //$A_7$
16          else return OOM       //$A_8$
17       else return OOM          //$A_8$
18       return (offset, H)
19
20  def release(x: nat × nat)
21       T := x.fst() + x.snd();
22       transfer_ownB(R,B); pass_ownT(R,Q) //$R_1$
```

Figure 3. Amazon Ring Buffer operations. Statements in red and green are auxiliary statements used to model ownership; comments A_1-A_8 and R_1 are control labels used in the proof of progress

either succeeds (i.e., there is enough free space in the buffer to transfer val), or the acquire returns OOM to indicate that it is currently out of memory. Since we have already checked that $size(val) \leq N$, i.e., val does indeed fit in the buffer, the writer executes m.backoff(), which causes it to wait for the addresses in the buffer to be released by the reader, before trying again.

The acquire and release operations of the buffer are given in Fig. 3. The buffer includes shared variables H and T, where H either points to the first free space in the buffer or at index N (see third case in Fig. 4), and T points to the first element that should be dequeued. The operations are designed so that H = T iff the queue is empty.

The acquire operation may succeed via one of four configurations (see Fig. 4), corresponding to lines 8, 10, 13 and 15 of the acquire operation in Fig. 3. The first, describes an acquire over an empty buffer (i.e., via line 8). This resets T to 0 and reserves the addresses at the start of the buffer. The second case (via line 10)[1] describes an acquire operation when H has wrapped around (i.e., H < T). In this case, the

[1]Assuming R has not moved T, after W performed the steps, as described by lines 6, 7 and 12 in Fig. 3.

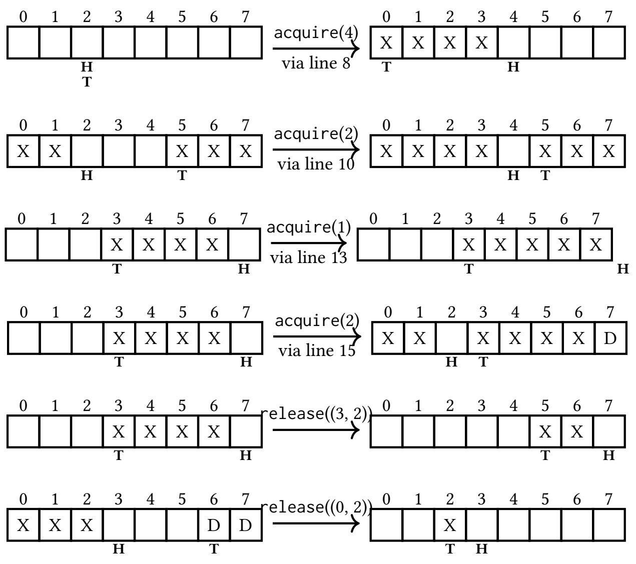

Figure 4. Ring buffer configurations before and after acquire and release operations, where "X" represents reserved addresses that may be used and "D" represents dummy reserved addresses that must not be used (assuming H = hW and T = tW)

acquire may only reserve T - H −1 addresses; otherwise H would be equal to tW, making the case indistinguishable from an empty buffer. The third case (via line 13)[1] represents an acquire operation where H has not wrapped around (i.e., T < H) and there is enough space at the end of the buffer. In this example, we also depict a phenomena where *all* of the addresses after T are reserved, and hence H has value N, which is an index that is outside the buffer. The fourth case (via line 15)[1] is one in which H has not wrapped around in the pre-state, but does wrap around in the post state since there are enough free addresses before T, but not after H. When H wraps around, it effectively "uses up" the addresses between H in the pre-state and N-1, but does not make these addresses[2] available for use by the writer.

The release operation takes as input a pair containing the offset address and the number of addresses to be released; the operation simply moves T to the new index. which is the sum of the two numbers (see second last case in Fig. 4). The final case depicts a release operation that "unwraps" the variables H and T.

There are two standard properties that are of interest for the most general client Fig. 2 that uses the buffer in Fig. 4.

Safety Data is read by the reader in the order written.
Progress All data that can be written is eventually read.

[2]These addresses will be referred to as "dummy reservations"

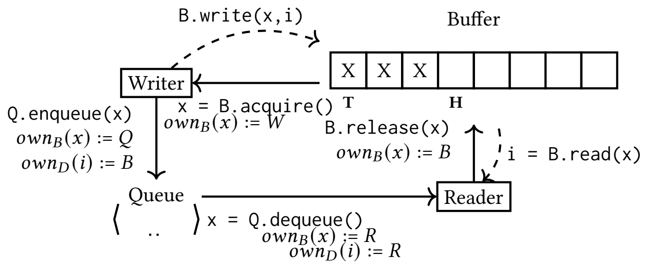

Figure 5. Ownership transfer in the ring buffer

The main challenge in the verification is the fact that safety and progress are achieved via separate interacting mechanisms: an arbitrarily sized *buffer*, whose addresses are reserved and written to in two different steps, and a *queue*, which is used to transfer reserved addresses from the writer to the reader. In Section 4, we develop a proof strategy based on ownership that is capable of proving both the safety and progress properties.

4 Augmenting Ownership Information

In this section we describe verification of the Amazon ring buffer using ownership. The essence of our proof is the observation that both the safety and progress properties described in Section 3 refer to the transfer of information from the writer to the reader. To this end, we think of the writer as initially owning a list of data to be transferred and for this data to be owned by the reader at the end of the protocol. The data must be read by the reader in order.

4.1 Ownership in ARB

An overview of ownership scheme and transfer mechanisms is given in Fig. 5. We assume a sequence *Data* of the data to be transferred from the writer to the reader, and an array *Buffer* for the ring buffer. The writer and reader are active entities that make use of the queue and the buffer as passive entities. With W we denote the writer, with R the reader, with B the ring buffer and with Q the queue:

$$\begin{aligned} own_D &: \{0, \dots, size(Data) - 1\} \longrightarrow \{W, B, R\} \\ own_B &: \{0, \dots, size(Buffer) - 1\} \longrightarrow \{W, R, B, Q, D\} \\ own_T &: \{W, R, Q\} \end{aligned}$$

where own_D is a function used to track ownership of the data in *Data* that is being transferred, and own_B is a function used to track ownership of the buffer addresses. The variable own_T is used to track ownership of the tail pointer.

Initially, $own_D(i) = W$ for each $0 \le i < size(Data)$, signifying that the data is owned by the writer. For safety, we require that $own_D(i) = R \longrightarrow own_D(i-1) = R$ for each i, to ensure that that data is transferred to the reader in order. For progress, we require that on termination, $own_D(i) = R$ for each $0 \le i < size(Data)$. Ownership of the data is transferred via the buffer. We model this by setting $own_D(i)$ to B, which occurs when the writer issues `B.write(x)`.

We also use ownership to track access rights of the buffer's addresses. Initially, $own_B(i) = B$ for each $0 \le i < size(Buffer)$ address. Once the writer acquires a span of addresses x, these addresses are effectively owned by W. The writer then transfers ownership of the addresses to the queue Q, which acts as a transient owner. The reader acquires addresses from Q, and after it has completed processing of the information contained within them, ownership of the addresses is transferred back to the buffer B (transient owner), during `B.release(x)`.

We make use of own_T to help visualising interference in the *tail* pointer assignment. The writer W takes ownership of the tail pointer from Q, whenever it performs a reset Fig. 3, line 6, and returns it to Q once it has successfully enqueued. The reader takes $own_T := R$ when performing `Q.dequeue()`, and returns it once it has finished updating the value of T Fig. 3, line 21.

4.2 Tracking Ownership Transfer

Our proof involves two types of ownership transfer for the bytes within the buffer. The first type sets ownership to a given set block of addresses $[u, v]$ to a given entity k:

$$\begin{aligned} &\texttt{set\_ownB}([u, v], k) = \\ &own_B := (\lambda j.\ \textbf{if}\ u \le j < v\ \textbf{then}\ k\ \textbf{else}\ own_B(j)) \end{aligned}$$

This function is introduced in the ring buffer operations Fig. 3 at lines 8, 10, 13 and 15. This abstraction of `set_ownB` does not handle setting of ownership in the "wrap-around" case, and a separate call must be performed, Fig. 3, line 15, to handle "dummy reservations" Fig. 4, case 4.

Example 4.1. In the augmented program, in Fig. 4 (case 2), W performs `set_ownB([2,4],W)` (since it moves `H` from index 2 to 4), resulting in a post state satisfying the property $\forall i \in \{2, 3\}.\ own_B(i) = W$. In the wrap-around case Fig. 4 (case 4), W in addition to `set_ownB([0,2],W)` performs `set_ownB([7,8],D)` (since it moves `H` from index 7 to 2), resulting in a post state satisfying:

$$own_B(7) = D \ \wedge\ (\forall i \in \{0, 1\}.\ \ own_B(i) = W)$$

The second type of ownership transfer transfers all m-owned buffer addresses to entity k:

$$\begin{aligned} &\texttt{transfer\_ownB}(m, k) \equiv \\ &own_B := (\lambda j.\ \textbf{if}\ own_B(j) = m\ \textbf{then}\ k\ \textbf{else}\ own_B(j)) \end{aligned}$$

This function is introduced in the client program Fig. 2 at line 12 and Fig. 3 at line 21.

Example 4.2. To see an application of `transfer_ownB(W,Q)`, consider Fig. 4 (case 4). After the writer has successfully performed `B.acquire(2)`, we have $\forall i \in \{0, 1, 7\}.\ own_B(i) = W$. The next step for W is to execute `Q.enqueue(0,2)`, which by the augmentation described above, also executes ownership transfer `transfer_ownB(W,Q)`. This transfers all W-owned buffer addresses to Q, resulting in a state satisfying $\forall i \in \{0, 1, 7\}.\ own_B(i) = Q$.

Tracking data ownership transfer. Data is transferred from the writer to the reader one at a time via the buffer (see Fig. 5). As such, we directly assign to the variable own_D during ownership transfer, i.e., lines 10 and 24 of Fig. 2.

$$\texttt{transfer\_ownD}(i, k) \equiv$$
$$own_D := (\lambda j.\ \textbf{if}\ j = i\ \textbf{then}\ k\ \textbf{else}\ own_D(j))$$

Tracking tail pointer ownership transfer. Ownership over the tail pointer T is transferred via `transfer_ownT(m,k)`:

$$\texttt{transfer\_ownT}(m, k) \equiv$$
$$own_T := (\textbf{if}\ own_T = m\ \textbf{then}\ k\ \textbf{else}\ own_T)$$

Example 4.3. To see an application of `transfer_ownT(_,_)`, consider Fig. 4 (case 1). Since both *head* and *tail* pointers are equal during the `B.acquire(2)` call, W would perform `transfer_ownT(Q,W)` as illustrated in Fig. 2 (line 7). This would result in $own_T = W$, and "allow" W to perform the subsequent assignment of *tail* pointer to the value 0 (line 8).

5 Verification

We now describe how ownership can be used to encode the required properties and verify correctness of the ARB. In Section 5.1, we describe an encoding of all possible scenarios for ownership of the bytes in the ring buffer. Then in Sections 5.2 and 5.3, we encode and verify safety and progress, respectively. We discuss our mechanisation effort in Section 5.4.

5.1 Ownership Scenarios

One of the main challenges to verification of systems with elements of arbitrary size is defining an appropriate state space to reason about state transitions in the context of Owicki-Gries *pre* and *post* conditions. Monitoring ownership transfer allows reasoning about a finite number of partitions of such elements, significantly reducing the complexity of limiting the state space. The Amazon ring buffer boasts an arbitrarily sized means of information transfer in the form of a buffer. Moreover, this partial library had to be verified against an arbitrarily sized "input" set of *Data*. As such, it is a good candidate for demonstrating verification using ownership transfer in action.

One of the properties we needed to verify was whether the means of information transfer, the buffer, could see illegal memory access. This ranged from overwriting of written *Data*, which had not yet been read, to retrieval of *Data*, to which the reader had not been granted access via the queuing mechanism.

Transfer of ownership within this verification of the Amazon ring buffer occurs as a result of actions by W and R. Guaranteeing interference freedom between W and R, from

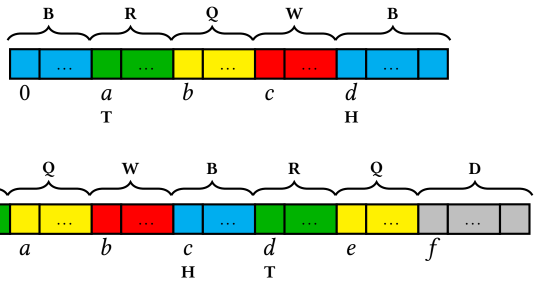

Figure 6. case_1 (top) and case_2 (bottom)

the point of view of the *Buffer* addresses, entails demonstrating that ownership transfer over the addresses follows Fig. 5. The writer, once it has performed `Q.enqueue(x)` should not be able to `B.acquire()` addresses, unless they have been released via `B.release(x)` (or have not been yet used from the start). Similarly, the reader, once it has performed `Q.dequeue()` should only be able to `B.read(x)` the addresses it has dequeued.

In a linear, "non-ring", buffer simply utilising the pointers *head* and *tail* to describe the possible scenarios with respect to *Data* (written, queued or read) would be enough to ensure the desired outcome. However, since there is no restriction of $T \leq H$, we used ownership to separate the cases into *case_1* and *case_2*:

$$case\_1:\ T \leq H$$
$$case\_2:\ H < T$$

both of which can be described by Fig. 6. The code is provided within the mechanised proof [27]. The figures represent all possible scenarios for the ownership of *Buffer* addresses. We use letters a, b, c, d, e and f to denote addresses which serve as boundaries, distinguishing addresses owned by entities across the whole buffer. We use these cases to describe the buffer, depending on which steps the threads are about to execute or have executed. Addresses coloured green are *currently being read from*, and are owned by the reader. Similarly, red describes addresses owned by the writer. Addresses coloured yellow and blue are owned by the *transient owners*, Q and B. Grey is reserved for the *dummy* reservations - addresses reserved, but not written to.

Aside from a few further restrictions (with respect to other auxiliary and non-auxiliary variables), the biggest constraint on the diagrams is that the following relationship must be preserved: $0 \leq a \leq b \leq \dots$. Looking specifically at *case_2*, an example further restriction could be that if $a > 0$, implying that R has ownership over some *Buffer* addresses between 0 and a, then $d = e$, implying that $\forall i.\ i \geq a \longrightarrow own_B(i) \neq R$. Performing ownership transfer, as shown in Fig. 5, must ensure that the state of the buffer can be described by the two cases in the *post* state.

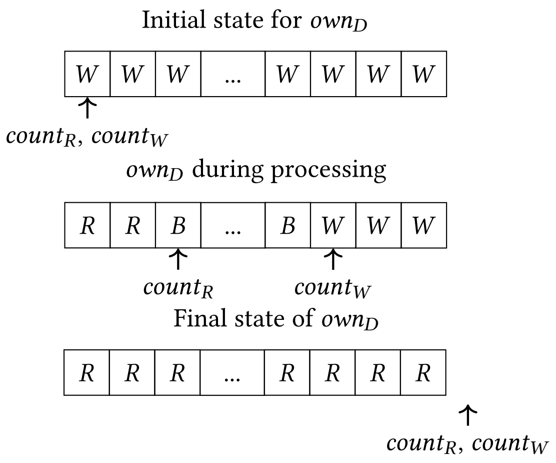

Figure 7. Transitions showing ownership of *Data*

5.2 Verifying Safety

The other main property that our system must satisfy is the correct (in order) transfer of ownership of data from the writer to the reader. As we have seen in Fig. 5, the system uses a buffer as an *endpoint* to facilitate ownership transfer.

This property our system must satisfy is therefore captured by the following simple invariant, recalling that $count_R$ tracks the index of the data to be read and $count_W$ tracks the index of the *Data* to be written.

$$
\begin{aligned}
Safety = \forall i \in \{0, \dots, size(Data) - 1\}.\\
&\textbf{if } i < count_R \textbf{ then } own_D(i) = R\\
&\textbf{elseif } count_R \le i < count_W \textbf{ then } own_D(i) = B\\
&\textbf{else } own_D(i) = W
\end{aligned}
$$

Since $count_R$ and $count_W$ are both initially 0, we have that initially $own_D(i) = W$ for all $i \in \{0, \dots, size(Data) - 1\}$. The writer sets $own_D(count_W)$ to B and increments $count_W$ by 1, once it performs `B.write(x,i)` call, whereas, the reader sets $own_D(count_R)$ to R and increments $count_R$ by 1, when it performs the `B.read(x)` call. This process is depicted in Fig. 7. This ensures that data is transferred from the writer to the reader in order. In particular, the external queue that is used for synchronisation of addresses (Fig. 2) must guarantee FIFO transfer of data. Note that a necessary supporting invariant:

$$count_R \le count_W \le size(Data) \quad (1)$$

must be introduced, in order to ensure R would not read data ahead of its successful `Q.enqueue()` by W. To synchronise this with the state of the queue Q, we introduce two further auxiliary variables, *numEnqs* and *numDeqs*, which are incremented upon successful `Q.enqueue(x)` and `Q.dequeue()` calls, respectively.

Relating ownership to non-auxiliary variables. In order to describe the system in terms of different levels of ownership, we must relate all non-auxiliary components to own_B and own_D. The abstraction, and subsequent mechanisation, preserves the second line of the main invariant through a supporting invariant, as given below:

$$size(Q) = numEnqs - numDeqs \quad (2)$$

Because transfer of own_D occurs during `B` operations, preservation of the relationship between *numEnqs*, *numDeqs*, $count_W$, $count_R$ and $size(Q)$ ensures the second line of the main invariant. Note, that the discrepancy between *numEnqs* and $count_W$ occurs only after the transition of `B.write(x,i)`. Similarly, $count_R$ differs from *numEnqs* only immediately after a successful `Q.dequeue()` call. As such, special caution must be taken when relating $size(Q)$ to $count_W$ and $count_R$.

Once W has finished reserving and writing into addresses from the *Buffer*, post the initial `acquire` step, variable H is bound to satisfy $H > 0$. This is characterised by the following supporting invariant:

$$count_W > count_R \longrightarrow H > 0 \quad (3)$$

$$count_W = count_R \land pc_R \neq idle \longrightarrow H > 0 \quad (4)$$

where pc_R is the program counter for the reader.

Observing Buffer ownership. Observing *Buffer* ownership is sufficient to describe, and maintain during execution, the states of all buffer addresses, provided ownership properties are satisfied w.r.t *case*_1 and *case*_2. If an address is owned by Q or B, we can guarantee that its respective own_B value would not change, unless either W or R performed an `acquire` or `release`.

The definitions of `set_ownB()` and `trans_ownB()` alongside other minor invariants, which guaranteed that variables (such as H and T) would abide by an upper bound of N, were enough to describe the idea of *dummy reservations* and transfer the reliance on observation of variables H and T explicitly to allow a more intuitive approach (specifically during informal reasoning stage) - asking, e.g. whether B owns enough of addresses for W to take ownership, or Q holds ownership tokens for R to take. An example of informal reasoning is given as follows: if Q has elements in it, the information contained in described addresses will not be overwritten, because W is restricted from taking ownership of addresses from Q.

We formalise the general proof of the scenarios presented in [27] using the definitions of buffer ownership transfer functions and the following supporting invariants, assuming x_R is the local reader variable that is set at line 4 of Fig. 2:

$$
\begin{aligned}
&Q \neq empty \longrightarrow\\
&\forall j \in \mathrm{dom}(Q).\ fst(Q(j)) + snd(Q(j)) \le size(Buffer) \quad (5)\\
&pc_R \neq idle_R \longrightarrow fst(x_R) + snd(x_R) \le size(Buffer) \quad (6)
\end{aligned}
$$

which dictate that all retrieved by R, and held by Q, entries preserve the property of boundness.

Besides these, a number of smaller (less interesting) invariants are necessary for bookkeeping purposes, e.g., $0 \le$

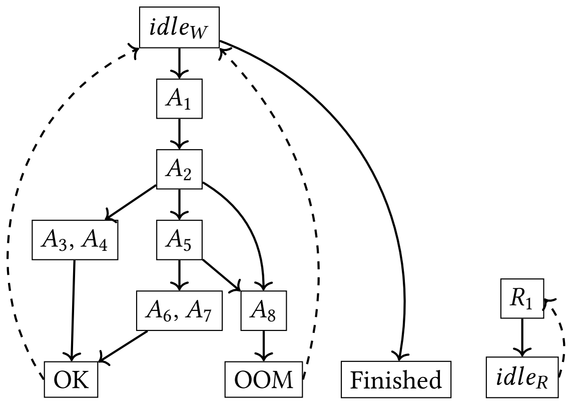

Figure 8. Orders over pc_W and pc_R; dashed arrows are transitions that restart the loops

$H, T \leq size(Buffer)$, which ensures that H and T are within their expected bounds. Details of these invariants may be found in our Isabelle mechanisation.

5.3 Verifying Progress

The main progress property that the ring buffer must guarantee is that all data is eventually transferred from the writer to the reader. Such a property is easily expressed in our model using ownership. In particular, we show that it is always possible to reach a state satisfying $Progress_{min}$ such that following holds:

$$Progress_{min} \longrightarrow \forall i \in \mathrm{dom}(Data).\ own_D(i) = R$$

One of the ways of observing the above, as suggested by Section 5.2, is to consider $count_W$ and $count_R$ as a method of representing own_D. $Data$ ownership for $Progress_{min}$ is satisfied when $count_R = count_W = size(Data)$, which implies that W has enqueued all desired data, and R has dequeued the last possible entry from Q. This idea forms the basis for using a well-founded order [13, 14] that includes $count_W$ and $count_R$ as parameters. We also have to be able to claim progress occurs, when either W or R executes a line of code without incrementing their respective data counters. To enable this, the well-founded order we use includes information on the program counter values of each thread.

In Fig. 8, labels A_i correspond to different labels within `B.acquire(x)`, see Fig. 3. Thus, the order given in Fig. 8 represents the lexicographical order:

$$idle_W \ > \ A_1 \ > \ A_2 \ > \ \{A_3, A_4, A_5\} > \cdots > \ Finished$$

Note, that the OK state includes state transitions of the operations described by both `B.write()` and `Q.enqueue()`. A similar relabelling and order can be shown for R, resulting in the following associated hierarchy, as described in Fig. 8: $R_1 \ > \ idle_R$.

Note, that transitions by W from state OOM to $idle_W$ do not increment $count_W$. To resolve this issue, we introduced a variable $tries$ which helps demonstrate the desired effect of monotonic decrease in $Progress$. See [27] for a more detailed explanation. The state transition induced by `B.write(x,i)` is encapsulated in OK. In order to distinguish the states before and after the write operation, we resort to tracking remaining writes, which is trivially $writesLeft = size(Data) - numEnqs$. We make use of this variable within the $Progress$ variant.

Overall, we use a lexicographical order over the following tuple for a state s:

$$\begin{aligned} Progress(s) = (&2n - s.count_W - s.count_R, \\ &writesLeft,\ K - s.tries,\ s.pc_R,\ s.pc_W) \end{aligned}$$

where K is a the upper bound[3] on the maximum number of retries via OOM, and $n = size(Data)$ is the number of elements to be transferred[4]. The $Progress$ tuple utilises lexicographical order implying highest rank of the left-most value.

We show that $Progress$ is strictly monotonically decreasing for each state transition, implying that any step taken towards completion should guarantee a decrease in rank. Since $Progress$ defines a global order over all processes, we are not required to introduce any additional fairness assumptions (e.g., strong / weak fairness) on the processes. Of course, since we prove progress under minimal fairness assumptions, the proof also holds in the presence of a strongly or weakly fair scheduler.

The minimal value for $Progress$ can now be formally written as:

$$Progress_{min}(s) = (0,\ 0,\ 0,\ idle_R,\ Finished_W)$$

The minimal value for $Progress$ is only obtainable when $count_R = n \wedge pc_R = idle_R$ (the reader has read the last $Data$ entry, and performed `B.release(x)`). Note that $Progress_{min}(s)$ implies $s.count_R = n$, which by $Safety$ guarantees our required progress property, i.e.,

$$\forall i \in \mathrm{dom}(Data).\ own_D(i) = R.$$

5.4 Mechanisation

The ring buffer has been modelled, and safety and progress properties (described in Sections 5.2 and 5.3) have been verified in Isabelle/HOL [27]. The program is modelled as a transition system, where control flow is tracked using program counters, and explicit auxiliary variables are used to encode ownership.

The safety proof proceeds via standard Owicki-Gries reasoning, which involves checking local correctness and interference freedom (global correctness) of each assertion. The progress proof involves checking whether each transition does indeed reduce the value of lexicographic order or not.

[3]If a retry only occurs when the reader moves T, we can use $K = size(Buffer)$ as an upper bound on the variable $tries$.
[4]There can be at most n number of writes and n number of reads performed by W and R, due to the size of the list $Data$

Once the required invariants and orders were encoded, the Isabelle proofs themselves were relatively straightforward. The strategy used was primarily splitting into cases for each line of code, followed by application of Isabelle's built in `sledgehammer` tool, which was able to solve the proof obligations that were generated.

6 Related Work

Notions of ownership are well-studied within programming languages and verification communities. Within the object-oriented literature, ownership is used to model encapsulation and information flow within the class hierarchy [2, 11]. These models are often supported by type systems [10, 26] to enable ownership tracking during compile time (see also [12]). Ownership is widely used in Concurrent Separation Logic and Rely Guarantee reasoning to aid in abstracting states, while maintaining the expressive nature of used variants [7].

In verification, ownership is primarily used to track availability of objects for use by other objects. Jacobs et al. [3, 20] implement verification of ownership based systems using ownership notation. Dietl and Müller [16] successfully extended use of ownership in Java Modelling Language environment. Boyapati et al. [4] implement an ownership type system to enforce a systematic approach to handling verification against deadlocks and data races in Java, recognising that ownership can help preserve variants of sub-objects, by focusing on hierarchical object ownership.

The idea of ownership has seen prominent use within concurrent separation logic [5, 6, 8, 9, 22, 25, 30], including variants that allow threads themselves to be owned [21]. Ownership has been used to aid in rely-guarantee reasoning [15], and works that combine rely-guarantee reasoning with separation logic [17, 29, 31, 32], with ownership being an integral part of verification. These logics have been applied to verifying a number of complex algorithms [28, 33] such as the flat combining paradigm [19]. However, separation logics are often specialised towards verifying particular examples; our aim is to incorporate ownership-based reasoning into an Owicki-Gries framework with a view to obtaining a more general solution.

The closest related works to ours is that of Haecki et al. [18], who use ownership to aid reasoning about correct address reservation in descriptor rings, using the Intel i82599 descriptor ring as the main example. They only reason about ownership at a single level (i.e., the addresses within the buffer), unlike our methods, which consider *Data* ownership and *Buffer* address ownership separately. As far as we are aware, full details of their verification techniques have not yet been released, and it will be interesting to compare their methods with ours once these details become available.

7 Conclusion and Future Work

This paper explores the idea of generalising an auxiliary variable, used in tracking thread access to a shared variable, to facilitate reasoning about concurrent programs both pen and paper, and using a theorem proving environment.

This paper analyses the proposed ring buffer structure by Amazon, and verifies the correctness and progress of parallel execution using a novel, ownership based, approach to access control, queuing and logical consequences [27]. The main reasons for choosing this problem, although could be pegged on its "Real World Problem" status, are the mechanism for synchronisation and the rules of the buffer. The existence of Queue could ensure interference-free parallel execution of Reads and Writes, however simple Owicki-Gries style informal arguments would quickly require much more than "simple" analysis to prove correctness and interference freedom. The extensive list of rules for acquiring/releasing Buffer bytes seemed like an interesting challenge for an ownership-based approach.

Acknowledgments

The authors thank Nathan Chong (Amazon) for introducing us to the ring buffer problem and for comments on an earlier version of this paper. This work is supported by EPSRC grants EP/V038915/1, EP/R032556/1 and EP/R025134/2; VeTSS; and ARC Grant DP190102142.

References

[1] [n. d.]. Amazon ring buffer source code. https://github.com/awslabs/aws-c-common/blob/master/source/ring_buffer.c Accessed Oct-2019.

[2] Anindya Banerjee and David A. Naumann. 2005. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM* 52, 6 (2005), 894–960.

[3] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. 2004. Verification of Object-Oriented Programs with Invariants. *J. Object Technol.* 3, 6 (2004), 27–56.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. *SIGPLAN Not.* 37, 11 (Nov. 2002), 211–230. https://doi.org/10.1145/583854.582440

[5] Stephen Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–34.

[6] Stephen Brookes. 2010. Fairness, Resources, and Separation. *Electr. Notes Theor. Comput. Sci.* 265 (2010), 177–195.

[7] Stephen Brookes. 2011. A Revisionist History of Concurrent Separation Logic. *Electr. Notes Theor. Comput. Sci.* 276 (09 2011), 5–28. https://doi.org/10.1016/j.entcs.2011.09.013

[8] Stephen D. Brookes. 2006. Variables as Resource for Shared-Memory Programs: Semantics and Soundness. In *MFPS (Electronic Notes in Theoretical Computer Science, Vol. 158)*, Stephen D. Brookes and Michael W. Mislove (Eds.). Elsevier, 123–150.

[9] C. Calcagno, P. W. O'Hearn, and H. Yang. 2007. Local Action and Abstract Separation Logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 366–378.

[10] David Clarke. 2001. *Object Ownership and Containment.* Technical Report.

[11] Dave Clarke, Sophia Drossopoulou, Peter Müller, James Noble, and Tobias Wrigstad. 2008. Aliasing, Confinement, and Ownership in Object-Oriented Programming. In *ECOOP (LNCS, Vol. 5475)*, Patrick Eugster (Ed.). Springer, 30–41.

[12] David G. Clarke, James Noble, and John M. Potter. 2001. Simple Ownership Types for Object Containment. In *ECOOP 2001 — Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 53–76.

[13] Robert Colvin and Brijesh Dongol. 2007. Verifying Lock-Freedom Using Well-Founded Orders. In *Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, Macau, China, September 26-28, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4711)*, Cliff B. Jones, Zhiming Liu, and Jim Woodcock (Eds.). Springer, 124–138. https://doi.org/10.1007/978-3-540-75292-9_9

[14] Robert Colvin and Brijesh Dongol. 2009. A general technique for proving lock-freedom. *Sci. Comput. Program.* 74, 3 (2009), 143–165. https://doi.org/10.1016/j.scico.2008.09.013

[15] Pedro Da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2015. Steps in Modular Specifications for Concurrent Modules (Invited Tutorial Paper). *Electronic Notes in Theoretical Computer Science* 319 (12 2015), 3–18. https://doi.org/10.1016/j.entcs.2015.12.002

[16] Werner Dietl and Peter Müller. 2005. Universes: Lightweight Ownership for JML. *JOURNAL OF OBJECT TECHNOLOGY* 4, 8 (2005), 5–32.

[17] Xinyu Feng. 2009. Local Rely-Guarantee Reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 315–327. https://doi.org/10.1145/1480881.1480922

[18] Roni Haecki, Lukas Humbel, Reto Achermann, David Cock, Daniel Schwyn, and Timothy Roscoe. 2019. CleanQ: a lightweight, uniform, formally specified interface for intra-machine data transfer. *ArXiv* abs/1911.08773 (2019).

[19] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Thira, Santorini, Greece) *(SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[20] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. 2005. Safe Concurrency for Aggregate Objects with Invariants. In *SEFM*, Bernhard K. Aichernig and Bernhard Beckert (Eds.). IEEE Computer Society, 137–147.

[21] Duy-Khanh Le, Wei-Ngan Chin, and Yong Meng Teo. 2015. Threads as Resource for Concurrency Verification. In *PEPM* (Mumbai, India) *(PEPM '15)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/2678015.2682540

[22] Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–67.

[23] Susan S. Owicki. 1978. Verifying parallel programs with resource allocation. In *Mathematical Studies of Information Processing, Proceedings of the International Conference, Kyoto, Japan, August 23-26, 1978 (Lecture Notes in Computer Science, Vol. 75)*, Edward K. Blum, Manfred Paul, and Satoru Takasu (Eds.). Springer, 151–164.

[24] S. S. Owicki and D. Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6 (1976), 319–340.

[25] M. Parkinson, R. Bornat, and C. Calcagno. 2006. Variables as Resource in Hoare Logics. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*. 137–146.

[26] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. 2006. Generic ownership for generic Java. In *OOPSLA*, Peri L. Tarr and William R. Cook (Eds.). ACM, 311–324.

[27] M. Semenyuk and B. Dongol. 2022. Isabelle/HOL files for "Ownership-Based Owicki-Gries Reasoning". https://doi.org/10.6084/m9.figshare.21762992.

[28] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *ICFP* (Boston, Massachusetts, USA) *(ICFP '13)*. ACM, New York, NY, USA, 377–390. https://doi.org/10.1145/2500365.2500600

[29] Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification*. Technical Report UCAM-CL-TR-726. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-726.pdf

[30] Viktor Vafeiadis. 2011. Concurrent separation logic and operational semantics. *Electronic Notes in Theoretical Computer Science* 276 (2011), 335–351.

[31] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. 2006. Proving Correctness of Highly-Concurrent Linearisable Objects. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, New York, USA) *(PPoPP '06)*. Association for Computing Machinery, New York, NY, USA, 129–136. https://doi.org/10.1145/1122971.1122992

[32] Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–271.

[33] Shuling Wang and Xu Wang. 2012. Proving Simpson's Four-Slot Algorithm Using Ownership Transfer. In *VERIFY-2010. 6th International Verification Workshop (EPiC Series in Computing, Vol. 3)*, Markus Aderhold, Serge Autexier, and Heiko Mantel (Eds.). EasyChair, 126–140. https://doi.org/10.29007/l2sp