Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis

ROLAND MEYER, TU Braunschweig, Germany
SEBASTIAN WOLFF, TU Braunschweig, Germany

Verification of concurrent data structures is one of the most challenging tasks in software verification. The topic has received considerable attention over the course of the last decade. Nevertheless, human-driven techniques remain cumbersome and notoriously difficult while automated approaches suffer from limited applicability. The main obstacle for automation is the complexity of concurrent data structures. This is particularly true in the absence of garbage collection. The intricacy of lock-free memory management paired with the complexity of concurrent data structures makes automated verification prohibitive.

In this work we present a method for verifying concurrent data structures and their memory management separately. We suggest two simpler verification tasks that imply the correctness of the data structure. The first task establishes an over-approximation of the reclamation behavior of the memory management. The second task exploits this over-approximation to verify the data structure without the need to consider the implementation of the memory management itself. To make the resulting verification tasks tractable for automated techniques, we establish a second result. We show that a verification tool needs to consider only executions where a single memory location is reused. We implemented our approach and were able to verify linearizability of Michael&Scott's queue and the DGLM queue for both hazard pointers and epoch-based reclamation. To the best of our knowledge, we are the first to verify such implementations fully automatically.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; **Program verification**; *Shared memory algorithms*; *Program specifications*; *Program analysis*;

Additional Key Words and Phrases: static analysis, lock-free data structures, verification, linearizability, safe memory reclamation, memory management

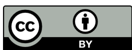
ACM Reference Format:
Roland Meyer and Sebastian Wolff. 2019. Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 58 (January 2019), 31 pages. https://doi.org/10.1145/3290371

1 INTRODUCTION

Data structures are a basic building block of virtually any program. Efficient implementations are typically a part of a programming language's standard library. With the advent of highly concurrent computing being available even on commodity hardware, concurrent data structure implementations are needed. The class of lock-free data structures has been shown to be particularly efficient. Using fine-grained synchronization and avoiding such synchronization whenever possible results in unrivaled performance and scalability.

Unfortunately, this use of fine-grained synchronization is what makes lock-free data structures also unrivaled in terms of complexity. Indeed, bugs have been discovered in published lock-free

Authors' addresses: Roland Meyer, TU Braunschweig, Germany, roland.meyer@tu-bs.de; Sebastian Wolff, TU Braunschweig, Germany, sebastian.wolff@tu-bs.de.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/1-ART58
https://doi.org/10.1145/3290371

data structures [Doherty et al. 2004a; Michael and Scott 1995]. This confirms the need for formal proofs of correctness. The de facto standard correctness notion for concurrent data structures is linearizability [Herlihy and Wing 1990]. Intuitively, linearizability provides the illusion that the operations of a data structure appear atomically. Clients of linearizable data structures can thus rely on a much simpler sequential specification.

Establishing linearizability for lock-free data structures is challenging. The topic has received considerable attention over the past decade (cf. Section 7). For instance, Doherty et al. [2004b] give a mechanized proof of a lock-free queue. Such proofs require a tremendous effort and a deep understanding of the data structure and the verification technique. Automated approaches remove this burden. Vafeiadis [2010a,b], for instance, verifies singly-linked structures fully automatically.

However, many linearizability proofs rely on a garbage collector. What is still missing are automated techniques that can handle lock-free data structures with manual memory management. The major obstacle in automating proofs for such implementations is that lock-free memory management is rather complicated—in some cases even as complicated as the data structure using it. The reason for this is that memory deletions need to be deferred until all unsynchronized, concurrent readers are done accessing the memory. Coping with lock-free memory management is an active field of research. It is oftentimes referred to as *Safe Memory Reclamation (SMR)*. The wording underlines its focus on safely reclaiming memory for lock-free programs. This results in the system design depicted in Figure 1. The clients of a lock-free data structure are unaware of how it manages its memory. The data structure uses an allocator to acquire memory, for example, using `malloc`. However, it does not `free` the memory itself. Instead, it delegates this task to an SMR algorithm which defers the `free` until it is *safe*. The deferral can be controlled by the data structure through an API the functions of which depend on the actual SMR algorithm.

In this paper we tackle the challenge of verifying lock-free data structures which use SMR. To make the verification tractable, we suggest a compositional approach which is inspired by the system design from Figure 1. We observe that the only influence the SMR implementation has on the data structure are the `free` operations it performs. So we introduce SMR specifications that capture when a `free` can be executed depending on the history of invoked SMR API functions. With such a specification at hand, we can verify that a given SMR implementation adheres to the specification. More importantly, it allows for a compositional verification of the data structure. Intuitively, we replace the SMR implementation with the SMR specification. If the SMR implementation adheres to the specification, then the specification over-approximates the frees of the implementation. Using this over-approximation for verifying the data structure is sound because frees are the only influence the SMR implementation has on the data structure.

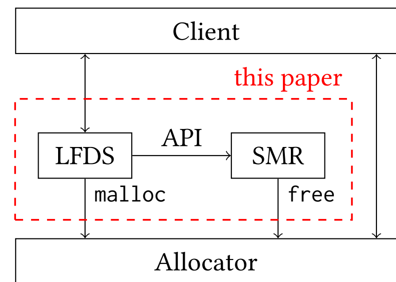


Fig. 1. Typical interaction between the components of a system. Lock-free data structures (LFDS) perform all their reclamation through an SMR component.

Although our compositional approach localizes the verification effort, it leaves the verification tool with a hard task: verifying shared-memory programs with memory reuse. Our second finding eases this task by taming the complexity of reasoning about memory reuse. We prove sound that it suffices to consider reusing a single memory location only. This result relies on data structures being invariant to whether or not memory is actually reclaimed and reused. Intuitively, this requirement boils down to ABA freedom and is satisfied by data structures from the literature.

To substantiate the usefulness of our approach, we implemented a linearizability checker which realizes the approaches presented in this paper, compositional verification and reduction of reuse to a single address. Our tool is able to establish linearizability of well-known lock-free data structures, such as *Treiber's stack* [Treiber 1986], *Michael&Scott's queue* [Michael and Scott 1996], and the *DGLM queue* [Doherty et al. 2004b], when using SMR, like *Hazard Pointers* [Michael 2002] and *Epoch-Based Reclamation* [Fraser 2004]. We remark that we needed both results for the benchmarks to go through. To the best of our knowledge, we are the first to verify lock-free data structures with SMR fully automatically. We are also the first to automatically verify the DGLM queue with any manual memory management.

Our contributions and the outline of our paper are summarized as follows:

- §4 introduces a means for specifying SMR algorithms and establishes how to perform compositional verification of lock-free data structures and SMR implementations,
- §5 presents a sound verification approach which considers only those executions of a program where at most a single memory location is reused,
- §6 evaluates our approach on well-known lock-free data structures and SMR algorithms, and demonstrates its practicality.

We illustrate our contributions informally in §2, introduce the programing model in §3, discuss related work in §7, and conclude the paper in §8. This paper comes with a companion technical report [Meyer and Wolff 2018] containing missing details.

2 THE VERIFICATION APPROACH ON AN EXAMPLE

The verification of lock-free data structures is challenging due to their complexity. One source of this complexity is the avoidance of traditional synchronization. This leads to subtle thread interactions and imposes a severe state space explosion. The problem becomes worse in the absence of garbage collection. This is due to the fact that lock-free memory reclamation is far from trivial. Due to the lack of synchronization it is typically impossible for a thread to judge whether or not certain memory will be accessed by other threads. Hence, naively deleting memory is not feasible. To overcome this problem, programmers employ SMR algorithms. While this solves the memory reclamation problem, it imposes a major obstacle for verification. For one, SMR implementations are oftentimes as complicated as the data structure using it. This makes the already hard verification of lock-free data structures prohibitive.

We illustrate the above problems on the lock-free queue from Michael and Scott [1996]. It is a practical example in that it is used for Java's `ConcurrentLinkedQueue` and C++ Boost's `lockfree::queue`, for instance. The implementation is given in Figure 2 (ignore the lines marked by `H` for a moment). The queue maintains a `NULL`-terminated singly-linked list of nodes. New nodes are enqueued at the end of that list. If the `Tail` pointer points to the end of the list, a new node is appended by linking `Tail->next` to the new node. Then, `Tail` is updated to point to the new node. If `Tail` is not pointing to the end of the list, the enqueue operation first moves `Tail` to the last node and then appends a new node as before. The dequeue operation on the other hand removes nodes from the front of the queue. To do so, it first reads the data of the second node in the queue (the first one is a dummy node) and then swings `Head` to the subsequent node. Additionally, dequeues ensure that `Head` does not overtake `Tail`. Hence, a dequeue operation may have to move `Tail` towards the end of the list before it moves `Head`. It is worth pointing out that threads read from the queue without synchronization. Updates synchronize on single memory words using atomic Compare-And-Swap (`CAS`).

In terms of memory management, the queue is flawed. It leaks memory because dequeued nodes are not reclaimed. A naive fix for this leak would be to uncomment the `delete head` statement in Line 40. However, other threads may still hold and dereference pointers to the then deleted node.

```
1  struct Node { data_t data; Node* next; };
2  shared Node* Head, Tail;
3  atomic init() { Head = new Node(); Head->next = NULL; Tail = Head; }
```

```
4     void enqueue(data_t input) {
5       Node* node = new Node();
6       node->data = input;
7       node->next = NULL;
8       while (true) {
9         Node* tail = Tail;
10 H      protect(tail, 0);
11 H      if (tail != Tail) continue;
12        Node* next = tail->next;
13        if (tail != Tail) continue;
14        if (next != NULL) {
15          CAS(&Tail, tail, next);
16          continue;
17        }
18        if (CAS(&tail->next, next, node))
19          break
20      }
21      CAS(&Tail, tail, node);
22 H    unprotect(0);
23    }
```

```
24    data_t dequeue() {
25      while (true) {
26        Node* head = Head;
27 H      protect(head, 0);
28 H      if (head != Head) continue;
29        Node* tail = Tail;
30        Node* next = head->next;
31 H      protect(next, 1);
32        if (head != Head) continue;
33        if (next == NULL) return EMPTY;
34        if (head == tail) {
35          CAS(&Tail, tail, next);
36          continue;
37        } else {
38          data_t output = next->data;
39          if (CAS(&Head, head, next)) {
40            // delete head;
41 H          retire(head);
42 H          unprotect(0); unprotect(1);
43            return output;
44    } } } }
```

Fig. 2. Michael&Scott's non-blocking queue [Michael and Scott 1996] extended with hazard pointers [Michael 2002] for safe memory reclamation. The modifications needed for using hazard pointers are marked with H. The implementation requires two hazard pointers per thread.

Such *use-after-free* dereference are *unsafe*. In C/C++, for example, the behavior is *undefined* and can result in a system crash due to a segfault.

To avoid both memory leaks and unsafe accesses, programmers employ SMR algorithms like *Hazard Pointers (HP)* [Michael 2002]. An example HP implementation is given in Figure 3. Each thread holds a HPRec record containing two single-writer multiple-reader pointers hp0 and hp1. A thread can use these pointers to protect nodes it will subsequently access without synchronization. Put differently, a thread requests other threads to defer the deletion of a node by protecting it. The deferred deletion mechanism is implemented as follows. Instead of explicitly deleting nodes, threads retire them. Retired nodes are stored in a thread-local retiredList and await reclamation. Eventually, a thread tries to reclaim the nodes collected in its list. Therefore, it reads the hazard pointers of all threads and copies them into a local protectedList. The nodes in the intersection of the two lists, retiredList ∩ protectedList, cannot be reclaimed because they are still protected by some thread. The remaining nodes, retiredList \ protectedList, are reclaimed.

Note that the HP implementation from Figure 3 allows for threads to join and part dynamically. In order to join, a thread allocates an HPRec and appends it to a shared list of such records. Afterwards, the thread uses the hp0 and hp1 fields of that record to issue protections. Subsequent reclaim invocations of any thread are aware of the newly added hazard pointers since reclaim traverses the shared list of HPRec records. To part, threads simply unprotect their hazard pointers. They do

```
45 struct HPRec { HPRec* next; Node* hp0; Node* hp1; }
46 shared HPRec* Records;
47 threadlocal HPRec* myRec;
48 threadlocal List<Node*> retiredList;
49 atomic init() { Records = NULL; }
```

```
50 void join() {
51   myRec = new HPRec();
52   while (true) {
53     HPRec* rec = Records;
54     myRec->next = rec;
55     if (CAS(Records, rec, myRec))
56       break;
57   }
58 }
59
60 void part() {
61   unprotect(0); unprotect(1);
62 }
63
64 void protect(Node* ptr, int i) {
65   if (i == 0) myRec->hp0 = ptr;
66   if (i == 1) myRec->hp1 = ptr;
67   assert(false);
68 }
69
70 void unprotect(int i) {
71   protect(NULL, i);
72 }
```

```
73 void retire(Node* ptr) {
74   if (ptr != NULL)
75     retiredList.add(ptr);
76   if (*) reclaim();
77 }
78
79 void reclaim() {
80   List<Node*> protectedList;
81   HPRec* cur = Records;
82   while (cur != NULL) {
83     Node* hp0 = cur->hp0;
84     Node* hp1 = cur->hp1;
85     protectedList.add(hp0);
86     protectedList.add(hp1);
87     cur = cur->next;
88   }
89   for (Node* ptr : retiredList) {
90     if (protectedList.contains(ptr))
91       continue;
92     retiredList.remove(ptr);
93     delete ptr;
94   }
95 }
```

Fig. 3. Simplified hazard pointer implementation [Michael 2002]. Each thread is equipped with two hazard pointers. Threads can dynamically join and part. Note that the record used to store a thread's hazard pointers is not reclaimed upon parting.

not reclaim their `HPRec` record [Michael 2002]. The reason for this is that reclaiming would yield the same difficulties that we face when reclaiming in lock-free data structures, as discussed before.

To use hazard pointers with Michael&Scott's queue we have to modify the implementation to `retire` dequeued nodes and to `protect` nodes that will be accessed without synchronization. The required modifications are marked by `H` in Figure 2. Retiring dequeued nodes is straight forward, as seen in Line 41. Successfully protecting a node is more involved. A typical pattern to do this is implemented by Lines 26 to 28, for instance. First, a local copy `head` of the shared pointer `Head` is created, Line 26. The node referenced by `head` is subsequently protected, Line 27. Simply issuing this protection, however, does not have the intended effect. Another thread could concurrently execute `reclaim` from Figure 3. If the reclaiming thread already computed its `protectedList`, i.e., executed `reclaim` up to Line 89, then it does not *see* the later protection and thus may reclaim the node referenced by `head`. The check from Line 28 safeguards the queue from such situations. It ensures that `head` has not been retired since the protection was issued.[1] Hence, no concurrent

[1] The reasoning is a bit more complicated. We discuss this in more detail in Section 5.3.

`reclaim` considers it for deletion. This guarantees that subsequent dereferences of `head` are safe. This pattern exploits a simple temporal property of hazard pointers, namely that *a retired node is not reclaimed if it has been protected continuously since before the retire* [Gotsman et al. 2013].

As we have seen, the verification of lock-free data structures becomes much more complex when considering SMR code. On the one hand, the data structure needs additional logic to properly use the SMR implementation. On the other hand, the SMR implementation is complex in itself. It is lock-free (as otherwise the data structure would not be lock-free) and uses multiple lists.

Our contributions make the verification tractable. First, we suggest a compositional verification technique which allows us to verify the data structure and the SMR implementation separately. Second, we reduce the impact of memory management for the two new verification tasks. We found that both contributions are required to automate the verification of data structures like Michael&Scott's queue with hazard pointers.

2.1 Compositional Verification

We propose a compositional verification technique. We split up the single, monolithic task of verifying a lock-free data structure together with its SMR implementation into two separate tasks: verifying the SMR implementation and verifying the data structure implementation without the SMR implementation. At the heart of our approach is a specification of the SMR behavior. Crucially, this specification has to capture the influence of the SMR implementation on the data structure. Our main observation is that it has *none*, as we have seen conceptually in Figure 1 and practically in Figures 2 and 3. More precisely, there is no *direct* influence. The SMR algorithm influences the data structure only *indirectly* through the underlying allocator: the data structure passes to-be-reclaimed nodes to the SMR algorithm, the SMR algorithm eventually reclaims those nodes using `free` of the allocator, and then the data structure can reuse the reclaimed memory with `malloc` of the allocator.

In order to come up with an SMR specification, we exploit the above observation as follows. We let the specification define when reclaiming retired nodes is allowed. Then, the SMR implementation is correct if the reclamations it performs are a subset of the reclamations allowed by the specification. For verifying the data structure, we use the SMR specification to over-approximate the reclamation of the SMR implementation. This way we over-approximate the influence the SMR implementation has on the data structure, provided that the SMR implementation is correct. Hence, our approach is sound for solving the original verification task.

Towards lightweight SMR specifications, we rely on the insight that SMR implementations, despite their complexity, implement rather simple temporal properties [Gotsman et al. 2013]. We have already seen that hazard pointers implement that *a retired node is not reclaimed if it has been protected continuously since before the retire*. These temporal properties are incognizant of the actual SMR implementation. Instead, they reason about those points in time when a call of an SMR function is invoked or returns. We exploit this by having SMR specifications judge when reclamation is allowed based on the *history* of SMR invocations and returns.

For the actual specification we use observer automata. A simplified specification for hazard pointers is given in Figure 4. The automaton $O_{HP}(t, a, i)$ is parametrized by a thread t, an address a, and an integer i. Intuitively, $O_{HP}(t, a, i)$ specifies when the i-th hazard pointer of t forces a `free` of a to be deferred. Technically, the automaton reaches an accepting state if a free is performed that should have been deferred. That is, we let observers specify *bad* behavior. We found this easier than to formalize the *good* behavior. For an example, consider the following histories:

$$h_1 = inv{:}\texttt{protect}(t_1, a, 0).\ ret{:}\texttt{protect}(t_1, a, 0).\ inv{:}\texttt{retire}(t_2, a).\ ret{:}\texttt{retire}(t_2, a).\ \texttt{free}(a) \ \text{ and}$$
$$h_2 = inv{:}\texttt{protect}(t_1, a, 0).\ inv{:}\texttt{retire}(t_2, a).\ ret{:}\texttt{protect}(t_1, a, 0).\ ret{:}\texttt{retire}(t_2, a).\ \texttt{free}(a)\ .$$

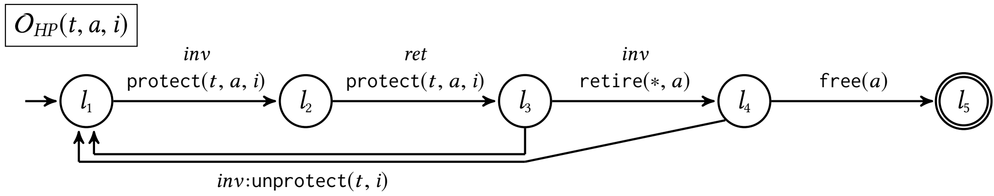


Fig. 4. Automaton for specifying *negative* HP behavior for thread t, address a, and index i. It states that if a was protected by thread t using hazard pointer i before a is retired by any thread (denoted by $*$), then freeing a must be deferred. Here, "must be deferred" is expressed by reaching a final state upon a free of a.

History h_1 leads $O_{HP}(t, a, i)$ to an accepting state. Indeed, that a is protected before being retired forbids a free of a. History h_2 does not lead to an accepting state because the retire is issued before the protection has returned. The free of a can be observed if the threads are scheduled in such a way that the protection of t_1 is not visible while t_2 computes its `retiredList`, as in the scenario described above for motivating why the check at Line 28 is required.

Now, we are ready for compositional verification. Given an observer, we first check that the SMR implementation is correct wrt. to that observer. Second, we verify the data structure. To that end, we strip away the SMR implementation and let the observer execute the frees. More precisely, we non-deterministically free those addresses which are allowed to be freed according to the observer.

Theorem 2.1 (Proven by Theorem 4.2). *Let $D(R)$ be a data structure D using an SMR implementation R. Let O be an observer. If R is correct wrt. O and if $D(O)$ is correct, then $D(R)$ is correct.*

A thorough discussion of the illustrated concepts is given in Section 4.

2.2 Taming Memory Management for Verification

Factoring out the implementation of the SMR algorithm and replacing it with its specification reduces the complexity of the data structure code under scrutiny. What remains is the challenge of verifying a data structure with manual memory management. As suggested by Abdulla et al. [2013]; Haziza et al. [2016] this makes the analysis scale poorly or even intractable. To overcome this problem, we suggest to perform verification in a simpler semantics. Inspired by the findings of the aforementioned works we suggest to avoid reallocations as much as possible. As a second contribution we prove the following theorem.

Theorem 2.2 (Proven by Theorem 5.20). *For a sound verification of safety properties it suffices to consider executions where at most a single address is reused.*

The rational behind this theorem is the following. From the literature we know that avoiding memory reuse altogether is not sound for verification [Michael and Scott 1996]. Put differently, correctness under garbage collection (GC) does not imply correctness under manual memory management (MM). The difference of the two program semantics becomes evident in the ABA problem. An ABA is a scenario where a pointer referencing address a is changed to point to address b and changed back to point to a again. Under MM a thread might erroneously conclude that the pointer has never changed if the intermediate value was not seen due to a certain interleaving. Typically, the root of the problem is that address a is removed from the data structure, deleted, reallocated, and reenters the data structure. Under GC, the exact same code does not suffer from this problem. A pointer referencing a would prevent it from being reused.

From this we learn that avoiding memory reuse does not allow for a sound analysis due to the ABA problem. So we want to check with little overhead to a GC analysis whether or not the program

$$
\begin{aligned}
cond &::= \; p = q \mid p \neq q \mid x = y \mid x \neq y \mid x < y \\
com &::= \; p := q \mid p := q.\mathtt{next} \mid p.\mathtt{next} := q \mid x := \mathtt{op}(x_1, \ldots, x_n) \mid x := q.\mathtt{data} \\
&\quad \mid p.\mathtt{data} := x \mid \mathtt{assert}\; cond \mid p := \mathtt{malloc} \mid \mathtt{enter}\; func(\bar{p}, \bar{x}) \mid \mathtt{exit} \mid smr \\
smr &::= \; \mathtt{free}(p) \mid \ldots
\end{aligned}
$$

Fig. 5. The syntax of atomic commands. Here, $x, y \in DVar$ are data variables, and $p, q \in PVar$ are pointer variables. We write $\bar{p}$ instead of $p_1, \ldots, p_n$ and similarly for $\bar{x}$. Besides `free`, SMR implementations may use not further specified commands *smr*.

under scrutiny suffers from the ABA problem. If not, correctness under GC implies correctness under MM. Otherwise, we reject the program as buggy.

To check whether a program suffers from ABAs it suffices to check for *first* ABAs. Fixing the address a of such a first ABA allows us to avoid reuse of any address except a while retaining the ability to detect the ABA. Intuitively, this is the case because the first ABA is the first time the program reacts differently on a reused address than on a fresh address. Hence, replacing reallocations with allocations of fresh addresses before the first ABA retains the behavior of the program.

A formal discussion of the presented result is given in Section 5.

3 PROGRAMS WITH SAFE MEMORY RECLAMATION

We define shared-memory programs that use an SMR library to reclaim memory. Here, we focus on the program. We leave unspecified the internal structure of SMR libraries (typically, they use the same constructs as programs), our development does not depend on it. We show in Section 4 how to strip away the SMR implementation for verification.

Memory. A memory m is a partial function $m : PExp \uplus DExp \rightharpoonup Adr \uplus \{\mathtt{seg}\} \uplus Dom$ which maps pointer expressions $PExp$ to addresses from $Adr \uplus \{\mathtt{seg}\}$ and data expressions $DExp$ to values from Dom, respectively. A pointer expression is either a pointer variable or a pointer selector: $PExp = PVar \uplus PSel$. Similarly, we have $DExp = DVar \uplus DSel$. The selectors of an address a are $a.\mathtt{next} \in PSel$ and $a.\mathtt{data} \in DSel$. A generalization to arbitrary selectors is straight forward. We use $\mathtt{seg} \notin Adr$ to denote undefined/uninitialized pointers. We write $m(e) = \bot$ if $e \notin dom(m)$. An address a is in-use if it is referenced by some pointer variable or if one of its selectors is defined. The set of such in-use addresses in m is $adr(m)$.

Programs. We consider computations of data structures D using an SMR library R, written $D(R)$. A computation τ is a sequence of actions. An action act is of the form $act = (t, com, up)$. Intuitively, act states that thread t executes command com resulting in the memory update up. An action stems either from executing D or from executing functions offered by R.

The commands are given in Figure 5. The commands of D include assignments, memory accesses, assertions, and allocations with the usual meaning. We make explicit when a thread enters and exits a library function with `enter` and `exit`, respectively. That is, we assume that computations are well-formed in the sense that no commands from D and all commands from R of a thread occur between `enter` and `exit`. Besides deallocations we leave the commands of R unspecified.

The memory resulting from a computation τ, denoted by m_τ, is defined inductively by its updates. Initially, pointer variables p are uninitialized, $m_\epsilon(p) = \mathtt{seg}$, and data variables x are default initialized, $m_\epsilon(x) = 0$. For a computation $\tau.act$ with $act = (t, com, up)$ we have $m_{\tau.act} = m_\tau[up]$. With the memory update $m' = m[e \mapsto v]$ we mean $m'(e) = v$ and $m'(e') = m(e')$ for all $e' \neq e$.

Semantics. The semantics of $D(R)$ is defined relative to a set $X \subseteq Adr$ of addresses that may be reused. It is the set of allowed executions, denoted by $[\![D(R)]\!]_X$. To make the semantics precise, let $fresh(\tau)$ and $freed(\tau)$ be those sets of addresses which have never been allocated and have been freed since their last allocation, respectively. Then, the semantics is defined inductively. In the base case we have the empty execution $\epsilon \in [\![D(R)]\!]_X$. In the induction step we have $\tau.act \in [\![D(R)]\!]_X$ if one of the following rules applies.

(Malloc) If $act = (t, p := \texttt{malloc}, [p \mapsto a, a.\texttt{next} \mapsto \texttt{seg}, a.\texttt{data} \mapsto d])$ where $d \in Dom$ is arbitrary and $a \in Adr$ is fresh or available for reuse, that is, $a \in fresh(\tau)$ or $a \in freed(\tau) \cap X$.
(FreePtr) If $act = (t, \texttt{free}(p), [a.\texttt{next} \mapsto \bot, a.\texttt{data} \mapsto \bot])$ with $m_\tau(p) = a \in Adr$.
(Enter) If $act = (t, \texttt{enter}\ func(\bar{p}, \bar{x}), \varnothing)$ with $\bar{p} = p_1, \ldots, p_k$ and $m_\tau(p_i) \in Adr$ for all $1 \leq i \leq k$.
(Exit) If $act = (t, \texttt{exit}, \varnothing)$.
(Assign1) If $act = (t, p := q, [p \mapsto m_\tau(q)])$.
(Assign2) If $act = (t, p.\texttt{next} := q, [a.\texttt{next} \mapsto m_\tau(q)])$ with $m_\tau(p) = a \in Adr$.
(Assign3) If $act = (t, p := q.\texttt{next}, [p \mapsto m_\tau(a.\texttt{next})])$ with $m_\tau(q) = a \in Adr$.
(Assign4) If $act = (t, x := \texttt{op}(y_1, \ldots, y_n), [x \mapsto d])$ with $d = \texttt{op}(m_\tau(y_1), \ldots, m_\tau(y_n))$.
(Assign5) If $act = (t, p.\texttt{data} := y, [a.\texttt{data} \mapsto m_\tau(y)])$ with $m_\tau(p) = a \in Adr$.
(Assign6) If $act = (t, x := q.\texttt{data}, [x \mapsto m_\tau(a.\texttt{data})])$ with $m_\tau(q) = a \in Adr$.
(Assert) If $act = (t, \texttt{assert}\ lhs \triangleq rhs, \varnothing)$ if $m_\tau(lhs) \triangleq m_\tau(rhs)$.

We assume that computations respect the control flow (program order) of threads. The control location after τ is denoted by $ctrl(\tau)$. We deliberately leave this unspecified as we will express only properties of the form $ctrl(\tau) = ctrl(\sigma)$ to state that after τ and σ the threads can execute the same commands.

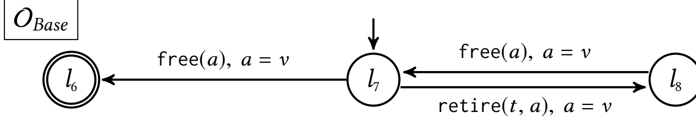
4 COMPOSITIONAL VERIFICATION

Our first contribution is a compositional verification approach for data structures $D(R)$ which use an SMR library R. The complexity of SMR implementations makes the verification of data structure and SMR implementation in a single analysis prohibitive. To overcome this problem, we suggest to verify both implementations independently of each other. More specifically, we (i) introduce a means for specifying SMR implementations, then (ii) verify the SMR implementation R against its specification, and (iii) verify the data structure D relative to the SMR specification rather than the SMR implementation. If both verification tasks succeed, then the data structure using the SMR implementation, $D(R)$, is correct.

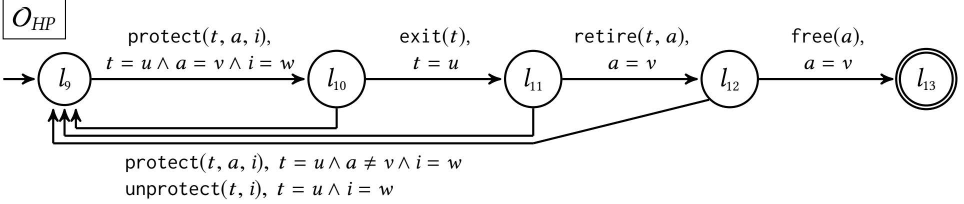
Our approach compares favorably to existing techniques. Manual techniques from the literature consider a monolithic verification task where both the data structure and the SMR implementaetion are verfied together [Fu et al. 2010; Gotsman et al. 2013; Krishna et al. 2018; Parkinson et al. 2007; Tofan et al. 2011]. Consequently, only simple implementations using SMR have been verified. Existing automated techniques rely on non-standard program semantics and support only simplistic SMR techniques [Abdulla et al. 2013; Haziza et al. 2016]. Refer to Section 7 for a more detailed discussion.

Towards our result, we first introduce observer automata for specifying SMR algorithms. Then we discuss the two new verification tasks and show that they imply the desired correctness result.

Observer Automata. An observer automaton $\mathcal{O}$ consists of observer locations, observer variables, and transitions. There is a dedicated initial location and some accepting locations. Transitions are of the form $l \xrightarrow{f(\bar{r}),\, g} l'$ with observer locations l, l', event $f(\bar{r})$, and guard g. Events $f(\bar{r})$ consist of a type f and parameters $\bar{r} = r_1, \ldots, r_n$. The guard is a Boolean formula over equalities of observer variables and the parameters $\bar{r}$. An observer state s is a tuple (l, φ) where l is a location and φ maps observer variables to values. Such a state is initial if l is initial, and similarly accepting if l is accepting. Then, $(l, \varphi) \xrightarrow{f(\bar{v})} (l', \varphi)$ is an observer step, if $l \xrightarrow{f(\bar{r}),\, g} l'$ is a transition and $\varphi(g[\bar{r} \mapsto \bar{v}])$



(a) Observer specifying that address v may be freed only if it has been retired and not been freed since.



(b) Observer specifying when HP defers frees. A retired cell v may not be freed if it has been protected continuously by the w-th hazard pointer of thread u since before being retired.

Fig. 6. Observer $O_{Base} \times O_{HP}$ characterizes the histories that violate the Hazard Pointer specification. Three observer variables, u, v, and w, are used to observe a thread, an address, and an integer, respectively. For better legibility we omit self-loops for every location and every event that is missing an outgoing transition from that location.

evaluates to *true*. With $\varphi(g[\bar{r} \mapsto \bar{v}])$ we mean g where the formal parameters $\bar{r}$ are replaced with the actual values $\bar{v}$ and where the observer variables are replaced by their φ-mapped values. Initially, the valuation φ is chosen non-deterministically; it is not changed by observer steps.

A *history* $h = f_1(\bar{v}_1) \ldots f_n(\bar{v}_n)$ is a sequence of events. We write $s \xrightarrow{h} s'$ if there are steps $s \xrightarrow{f_1(\bar{v}_1)} \cdots \xrightarrow{f_n(\bar{v}_n)} s'$. If s' is accepting, then we say that h is accepted by s. We use observers to characterize *bad behavior*. So we say h is in the specification of s, denoted by $h \in \mathcal{S}(s)$, if it is not accepted by s. Formally, the specification of s is the set $\mathcal{S}(s) := \{h \mid \forall s'.\ s \xrightarrow{h} s' \implies s' \text{ not final}\}$ of histories that are not accepted by s. The specification of O is the set of histories that are not accepted by any initial state of O, $\mathcal{S}(O) := \bigcap\{\mathcal{S}(s) \mid s \text{ initial}\}$. The cross-product $O_1 \times O_2$ denotes an observer with $\mathcal{S}(O_1 \times O_2) = \mathcal{S}(O_1) \cap \mathcal{S}(O_2)$.

SMR Specifications. To use observers for specifying SMR algorithms, we have to instantiate appropriately the histories they observe. Our instantiation crucially relies on the fact that programmers of lock-free data structures rely solely on simple temporal properties that SMR algorithms implement [Gotsman et al. 2013]. These properties are typically incognizant of the actual SMR implementation. Instead, they allow reasoning about the implementation's behavior based on the temporal order of function invocations and responses. With respect to our programming model, `enter` and `exit` actions provided the necessary means to deduce from the data structure computation how the SMR implementation behaves.

We instantiate observers for specifying SMR as follows. As event types we use (i) $func_1, \ldots, func_n$, the functions offered by the SMR algorithm, (ii) `exit`, and (iii) `free`. The parameters to the events are (i) the executing thread and the parameters to the call in case of type $func_i$, (ii) the executing thread for type `exit`, and (iii) the parameters to the call for type `free`. Here, type $func_i$ represents the corresponding `enter` command of a call to $func_i$. The corresponding `exit` event is uniquely defined because both $func_i$ and `exit` events contain the executing thread.

For an example, consider the hazard pointer specification $O_{Base} \times O_{HP}$ from Figure 6. It consists of two observers. First, O_{Base} specifies that no address must be freed that has not been retired. Second,

O_{HP} implements the temporal property that no address must be freed if it has been protected continuously since before the retire. For observer $O_{Base} \times O_{HP}$ we assume that no address is retired multiple times before being reclaimed (freed) by the SMR implementation. This avoids handling such *double-retire* scenarios in the observer, keeping it smaller and simpler. The assumption is reasonable because in a setting where SMR is used a double-retire is the analogue of a double-free and thus avoided. Our experiments confirm this intuition.

With an SMR specification in form of an observer O_{SMR} at hand, our task is to check whether or not a given SMR implementation R satisfies this specification. We do this by converting a computation τ of R into its induced history $\mathcal{H}(\tau)$ and check if $\mathcal{H}(\tau) \in \mathcal{S}(O_{SMR})$. The induced history $\mathcal{H}(\tau)$ is a projection of τ to `enter`, `exit`, and `free` actions. This projection replaces the formal parameters in τ with their actual values. For example, $\mathcal{H}(\tau.(t, \texttt{protect}(p, x), up)) = \mathcal{H}(\tau).\texttt{protect}(t, m_\tau(p), m_\tau(x))$. Then, τ satisfies O_{SMR} if $\mathcal{H}(\tau) \in \mathcal{S}(O_{SMR})$. The SMR implementation R satisfies O_{SMR} if every possible usage of R produces a computation that satisfies O_{SMR}. To generate all such computations, we use a most general client (MGC) for R which concurrently executes arbitrary sequences of SMR functions.

Definition 4.1 (SMR Correctness). An SMR implementation R *is correct wrt. a specification* O_{SMR}, denoted by $R \models O_{SMR}$, if for every $\tau \in [\![MGC(R)]\!]_{Adr}$ we have $\mathcal{H}(\tau) \in \mathcal{S}(O_{SMR})$.

From the above definition follows the first new verification task: prove that the SMR implementation R cannot possibly violate the specification O_{SMR}. Intuitively, this boils down to a reachability analysis of accepting states in the cross-product of $MGC(R)$ and O_{SMR}. Since we can understand R as a lock-free data structure itself, this task is similar to our next one, namely verifying the data structure relative to O_{SMR}. In the remainder of the paper we focus on this second task because it is harder than the first one. The reason for this lies in that SMR implementations typically do not reclaim the memory they use. This holds true even if the SMR implementation supports dynamic thread joining and parting [Michael 2002] (cf. `part()` from Figure 3). The absence of reclamation allows for a simpler[2] and more efficient analysis. We confirm this in our experiments where we automatically verify the Hazard Pointer implementation from Figure 3 wrt. $O_{Base} \times O_{HP}$.

Compositionality. The next task is to verify the data structure $D(R)$ avoiding the complexity of R. We have already established correctness of R wrt. a specification O_{SMR}. Intuitively, we now replace R by O_{SMR}. Because O_{SMR} is an observer, and not program code like R, we cannot just execute O_{SMR} in place of R. Instead, we remove the SMR implementation from $D(R)$. The result is $D(\epsilon)$ the computations of which correspond to the ones of $D(R)$ with SMR-specific actions between `enter` and `exit` being removed. To account for the frees that R executes, we introduce *environment steps*. We non-deterministically check for every address a whether or not O_{SMR} allows freeing it. If so, we free the address. Formally, the semantics $[\![D(O_{SMR})]\!]_X$ corresponds to $[\![D(\epsilon)]\!]_X$ as defined in Section 3 plus a new rule for frees from the environment.

(Free) If $\tau \in [\![D(\epsilon)]\!]_X$ and $a \in Adr$ can be freed, i.e., $\mathcal{H}(\tau).\texttt{free}(a) \in \mathcal{S}(O_{SMR})$, then we have $\tau.act \in [\![D(\epsilon)]\!]_X$ with $act = (t, \texttt{free}(a), [a.\texttt{next} \mapsto \bot, a.\texttt{data} \mapsto \bot])$.

Note that $\texttt{free}(a)$ from the environment has the same update as $\texttt{free}(p)$ from R if p points to a. With this definition, $D(O_{SMR})$ performs more frees than $D(R)$ provided $R \models O_{SMR}$.

With the semantics of data structures with respect to an SMR specification rather than an SMR implementation set up, we can turn to the main result of this section. It states that the correctness of R wrt. O_{SMR} and the correctness of $D(O_{SMR})$ entail the correctness of the original program $D(R)$. Here, we focus on the verification of safety properties. It is known that this reduces to control

[2]In terms of Section 5, the absence of reclamation results in SMR implementations being free from pointer races and harmful ABAs since pointers do not become invalid. Intuitively, this allows us to combine our results with ones from Haziza et al. [2016] and verify the SMR implementation in a garbage-collected semantics.

location reachability [Vardi 1987]. So we can assume that there is a dedicated *bad* control location in D the unreachability of which is equivalent to the correctness of $D(R)$. To establish the result, we require that the interaction between D and R follows the one depicted in Figure 1 and discussed on an example in Section 2. That is, the only influence R has on D are frees. In particular, this means that R does not modify the memory accessed by D. We found this restriction to be satisfied by many SMR algorithms from the literature. We believe that our development can be generalized to reflect memory modifications performed by the SMR algorithm. A proper investigation of the matter, however, is beyond the scope of this paper.

Theorem 4.2 (Compositionality). *Let $R \models O_{SMR}$. If $[\![D(O_{SMR})]\!]_{Adr}$ is correct, so is $[\![D(R)]\!]_{Adr}$.*

Compositionality is a powerful tool for verification. It allows us to verify the data structure and the SMR implementation independently of each other. Although this simplifies the verification, reasoning about lock-free programs operating on a shared memory remains hard. In Section 5 we build upon the above result and propose a sound verification of $[\![D(O_{SMR})]\!]_{Adr}$ which considers only executions reusing a single addresses.

5 TAMING MEMORY REUSE

As a second contribution we demonstrate that *one can soundly verify a data structure $D(O_{SMR})$ by considering only those computations where at most a single cell is reused.* This avoids the need for a state space exploration of full $[\![D(O_{SMR})]\!]_{Adr}$. Such explorations suffer from a severe state space explosion. In fact, we were not able to make our analysis from Section 6 go through without this second contribution. Previous works [Abdulla et al. 2013; Haziza et al. 2016; Holík et al. 2017] have not required such a result since they did not consider fully fledged SMR implementations like we do. For a thorough discussion of related work refer to Section 7.

Our results are independent of the actual safety property and the actual observer O_{SMR} specifying the SMR algorithm. To achieve this, we establish that for every computation from $[\![D(O_{SMR})]\!]_{Adr}$ there is a *similar* computation which reuses only a single address. We construct such a similar computation by eliding reuse in the original computation. With elision we mean that we replace in a computation a freed address with a fresh one. This allows a subsequent allocation to `malloc` the elided address fresh instead of reusing it. Our notion of similarity makes sure that in both computations the threads reach the same control locations. This allows for verifying safety properties.

The remainder of the section is structured as follows. Section 5.1 introduces our notion of similarity. Section 5.2 formalizes requirements on $D(O_{SMR})$ such that the notion of similarity suffices to prove the desired result. Section 5.3 discusses how the ABA problem can affect soundness of our approach and shows how to detect those cases. Section 5.4 presents the reduction result.

5.1 Similarity of Computations

Our goal is to mimic a computation τ where memory is reused arbitrarily with a computation σ where memory reuse is restricted. As noted before, we want that the threads in τ and σ reach the same control locations in order to verify safety properties of τ in σ. We introduce a *similarity relation* among computations such that τ and σ are similar if they can execute the same actions. This results in both reaching the same control locations as desired. However, control location equality alone is insufficient for σ to mimic subsequent actions of τ, that is, to preserve similarity for subsequent actions. This is because most actions involve memory interaction. Since σ reuses memory differently than τ, the memory of the two computations is not equal. Similarity requires a non-trivial correspondence wrt. the memory. Towards a formal definition let us consider an example.

Example 5.1. Let τ_1 be a computation of a data structure $D(O_{Base} \times O_{HP})$ using hazard pointers:

$$\tau_1 = (t, p := \texttt{malloc}, [p \mapsto a, \dots]).(t, \texttt{retire}(p), \varnothing).(t, \texttt{free}(a), [\dots]).(t, \texttt{exit}, \varnothing).$$
$$(t, q := \texttt{malloc}, [q \mapsto a, \dots]) .$$

In this computation, thread t uses pointer p to allocate address a. The address is then retired and freed. In the subsequent allocation, t acquires another pointer q to a; a is reused.

If σ_1 is a computation where a shall not be reused, then σ_1 is not able to execute the exact same sequence of actions as τ_1. However, it can mimic τ_1 as follows:

$$\sigma_1 = (t, p := \texttt{malloc}, [p \mapsto b, \dots]).(t, \texttt{retire}(p), \varnothing).(t, \texttt{free}(b), [\dots]).(t, \texttt{exit}, \varnothing).$$
$$(t, q := \texttt{malloc}, [q \mapsto a, \dots]) ,$$

where σ_1 coincides with τ_1 up to replacing the first allocation of a with another address b. We say that σ_1 elides the reuse of a. Note that the memories of τ_1 and σ_1 differ on p and agree on q.

In the above example, p is a dangling pointer. Programmers typically avoid using such pointers because it is unsafe. For a definition of similarity, this practice suggests that similar computations must coincide only on the non-dangling pointers and may differ on the dangling ones. To make precise which pointers in a computation are dangling, we use the notion of *validity*. That is, we define a set of valid pointers. The dangling pointers are then the complement of the valid pointers. We take this detour because we found it easier to formalize the valid pointers.

Initially, no pointer is valid. A pointer becomes valid if it receives its value from an allocation or another valid pointer. A pointer becomes invalid if its referenced memory location is deleted or it receives its value from an invalid pointer. A deletion of a memory cell makes invalid its pointer selectors and all pointers to that cell. A subsequent reallocation of that cell makes valid only the receiving pointer; all other pointers to that cell remain invalid. Assertions of the form $p = q$ validate p if q is valid, and vice versa. We omit the formal definition of the valid pointers in a computation τ, denoted by $valid_\tau$.

Example 5.2 (Continued). In both τ_1 and σ_1 from the previous example, the last allocation renders valid pointer q. On the other hand, the free to a in τ_1 renders p invalid. The reallocation of a does not change the validity of p, it remains invalid. In σ_1, address b is allocated and freed rendering p invalid. It remains invalid after the subsequent allocation of a. That is, both τ_1 and σ_1 agree on the validity of q and the invalidity of p. Moreover, τ_1 and σ_1 agree on the valuation of the valid q and disagree (here by chance) on the valuation of the invalid p.

The above example illustrates that eliding reuse of memory leads to a different memory valuation. However, the elision can be performed in such a way that the valid memory is not affected. So we say that two computations are similar if they agree on the resulting control locations of threads and the valid memory. The valid memory includes the valid pointer variables, the valid pointer selectors, the data variables, and the data selectors of addresses that are referenced by a valid pointer variable/selector. Formally, this is a restriction of the entire memory to the valid pointers, written $m_\tau|_{valid_\tau}$.

Definition 5.3 (Restrictions). A restriction of m to a set $P \subseteq PExp$, denoted by $m|_P$, is a new m' with $dom(m') := P \cup DVar \cup \{a.\texttt{data} \in DExp \mid a \in m(P)\}$ and $m(e) = m'(e)$ for all $e \in dom(m')$.

We are now ready to formalize the notion of similarity among computations. Two computations are similar if they agree on the control location of threads and the valid memory.

Definition 5.4 (Computation Similarity). Two computations τ and σ are *similar*, denoted by $\tau \sim \sigma$, if $ctrl(\tau) = ctrl(\sigma)$ and $m_\tau|_{valid_\tau} = m_\sigma|_{valid_\sigma}$.

If two computations τ and σ are similar, then each action enabled after τ can be mimicked in σ. An action $act = (t, com, up)$ can be mimicked by another action $act' = (t, com, up')$. Both actions agree on the executing thread and the executed command but may differ in the memory update. The reason for this is that similarity does not relate the invalid parts of the memory. This may give another update in σ if *com* involves invalid pointers.

Example 5.5 (Continued). Consider the following continuation of τ_1 and σ_1:

$$\tau_2 = \tau_1 . (t, p := p, up) \qquad \text{and} \qquad \sigma_2 = \sigma_1 . (t, p := p, up') ,$$

where we append an assignment of p to itself. The prefixes τ_1 and σ_1 are similar, $\tau_1 \sim \sigma_1$. Nevertheless, the updates up and up' differ because they involve the valuation of the invalid pointer p which differs in τ_1 and σ_1. The updates are $up = [p \mapsto a]$ and $up' = [p \mapsto b]$. Since the assignment leaves p invalid, similarity is preserved by the appended actions, $\tau_2 \sim \sigma_2$. We say that act' mimics act.

Altogether, similarity does not guarantee that the exact same actions are executable. It guarantees that every action can be mimicked such that similarity is preserved.

In the above we omitted an integral part of the program semantics. Memory reclamation is not based on the control location of threads but on an observer examining the history induced by a computation. The enabledness of a `free` is not preserved by similarity. On the one hand, this is due to the fact that invalid pointers can be (and in practice are) used in SMR calls which lead to different histories. On the other hand, similar computations end up in the same control location but may perform different sequences of actions to arrive there, for instance, execute different branches of conditionals. That is, to mimic `free` actions we need to correlate the behavior of the observer rather than the behavior of the program. We motivate the definition of an appropriate relation.

Example 5.6 (Continued). Consider the following continuation of τ_2 and σ_2:

$$\begin{aligned} \tau_3 &= \tau_2 . (t, \texttt{protect}(p, i), \emptyset) . (t, \texttt{exit}, \emptyset) . (t, \texttt{retire}(q), \emptyset) . (t, \texttt{exit}, \emptyset) \\ \text{and} \quad \sigma_3 &= \sigma_2 . (t, \texttt{protect}(p, i), \emptyset) . (t, \texttt{exit}, \emptyset) . (t, \texttt{retire}(q), \emptyset) . (t, \texttt{exit}, \emptyset) , \end{aligned}$$

where t issues a protection and a retirement using p and q, respectively. The histories induced by those computations are:

$$\begin{aligned} \mathcal{H}(\tau_3) &= \mathcal{H}(\tau_1) . \texttt{protect}(t, a, i) . \texttt{exit}(t) . \texttt{retire}(t, a) . \texttt{exit}(t) \\ \text{and} \quad \mathcal{H}(\sigma_3) &= \mathcal{H}(\sigma_1) . \texttt{protect}(t, b, i) . \texttt{exit}(t) . \texttt{retire}(t, a) . \texttt{exit}(t) . \end{aligned}$$

Recall that τ_2 and σ_2 are similar. Similarity guarantees that the events of the `retire` call coincide because q is valid. The events of the `protect` call differ because the valuations of the invalid p differ. That is, SMR calls do not necessarily emit the same event in similar computations. Consequently, the observer states after τ_3 and σ_3 differ. More precisely, O_{HP} from Figure 6b has the following runs from the initial observer state (l_9, φ) with $\varphi = \{u \mapsto t, v \mapsto a, w \mapsto i\}$:

$$\begin{aligned} & (l_9, \varphi) \xrightarrow{\mathcal{H}(\tau_1)} (l_9, \varphi) \xrightarrow{\texttt{protect}(t,a,i)} (l_{10}, \varphi) \xrightarrow{\texttt{exit}(t)} (l_{11}, \varphi) \xrightarrow{\texttt{retire}(t,a)} (l_{12}, \varphi) \xrightarrow{\texttt{exit}(t)} (l_{12}, \varphi) \\ \text{and} \quad & (l_9, \varphi) \xrightarrow{\mathcal{H}(\sigma_1)} (l_9, \varphi) \xrightarrow{\texttt{protect}(t,b,i)} (l_9, \varphi) \xrightarrow{\texttt{exit}(t)} (l_9, \varphi) \xrightarrow{\texttt{retire}(t,a)} (l_9, \varphi) \xrightarrow{\texttt{exit}(t)} (l_9, \varphi) \ . \end{aligned}$$

This prevents a from being freed after τ_3 (a `free`(a) would lead to the final state (l_{13}, φ) and is thus not enabled) but allows for freeing it after σ_3.

The above example shows that eliding memory addresses to avoid reuse changes observer runs. The affected runs involve freed addresses. Like for computation similarity, we define a relation among computations which captures the observer behavior on the *valid addresses*, i.e., those addresses that are referenced by valid pointers, and ignores all other addresses. Here, we do not use an equivalence relation. That is, we do not require observers to reach the exact same state for valid addresses. Instead, we express that the mimicking σ allows for more observer behavior on the valid

addresses than the mimicked τ does. We define an *observer behavior inclusion* among computations. This is motivated by the above example. There, address a is valid because it is referenced by a valid pointer, namely q. Yet the observer runs for a differ in τ_3 and σ_3. After σ_3 more behavior is possible; σ_3 can free a while τ_3 cannot.

To make this intuition precise, we need a notion of *behavior on an address.* Recall that the goal of the desired behavior inclusion is to enable us to mimic `free`s. Intuitively, the behavior allowed by O_{SMR} on address a is the set of those histories that *lead* to a free of a.

Definition 5.7 (Observer Behavior). The behavior allowed by O_{SMR} on address a after history h is the set $\mathcal{F}(h, a) := \{h' \mid h.h' \in \mathcal{S}(O_{SMR}) \wedge frees(h') \subseteq a\}$.

Note that $h' \in \mathcal{F}(h, a)$ contains `free` events for address a only. This is necessary because an address may become invalid before being freed if, for instance, the address becomes unreachable from valid pointers. The mimicking computation σ may have already freed such an address while τ has not, despite similarity. Hence, the free is no longer allowed after σ but still possible after τ. To prevent such invalid addresses from breaking the desired inclusion on valid addresses, we strip from $\mathcal{F}(h, a)$ all `free`s that do not target a. Note that we do not even retain frees of valid addresses here. This way, only actions which emit an event influence $\mathcal{F}(h, a)$.

The observer behavior inclusion among computations is defined such that σ includes at least the behavior of τ on the valid addresses. Formally, the valid addresses in τ are $adr(m_\tau|_{valid_\tau})$.

Definition 5.8 (Observer Behavior Inclusion). Computation σ *includes the (observer) behavior* of τ, denoted by $\tau \lessdot \sigma$, if $\mathcal{F}(\tau, a) \subseteq \mathcal{F}(\sigma, a)$ holds for all $a \in adr(m_\tau|_{valid_\tau})$.

5.2 Preserving Similarity

The development in Section 5.1 is idealized. There are cases where the introduced relations do not guarantee that an action can be mimicked. All such cases have in common that they involve the usage of invalid pointers. More precisely, (i) the computation similarity may not be strong enough to mimic actions that dereference invalid pointers, and (ii) the observer behavior inclusion may not be strong enough to mimic calls involving invalid pointers. For each of those cases we give an example and restrict our development. We argue throughout this section that our restrictions are reasonable. Our experiments confirm this. We begin with the computation similarity.

Example 5.9 (Continued). Consider the following continuation of τ_3 and σ_3:

$$\tau_4 = \tau_3 \,.\, (t, q.\mathtt{next} := q, [a.\mathtt{next} \mapsto a]) \,.\, (t, p.\mathtt{next} := p, [a.\mathtt{next} \mapsto a])$$
$$\text{and} \quad \sigma_4 = \sigma_3 \,.\, (t, q.\mathtt{next} := q, [a.\mathtt{next} \mapsto a]) \,.\, (t, p.\mathtt{next} := p, [b.\mathtt{next} \mapsto b]) \,.$$

The first appended action updates a.`next` in both computations to a. Since q is valid after both τ_3 and σ_3 this assignment renders valid a.`next`. The second action assigns to a.`next` in τ_4. This results in a.`next` being invalid after τ_4 because the right-hand side of the assignment is the invalid p. In σ_4 the second action updates b.`next` which is why a.`next` remains valid. That is, the valid memories of τ_4 and σ_4 differ. We have executed an action that cannot be mimicked on the valid memory despite the computations being similar.

The problem in the above example is the dereference of an invalid pointer. The computation similarity does not give any guarantees about the valuation of such pointers. Consequently, it cannot guarantee that an action using invalid pointers can be mimicked. To avoid such problems, we forbid programs to dereference invalid pointers.

The rational behind this is as follows. Recall that an invalid pointer is dangling. That is, the memory it references has been freed. If the memory has been returned to the underlying operating system, then a subsequent dereference is unsafe, that is, prone to a system crash due to a *segfault.*

Hence, such dereferences should be avoided. The dereference is only safe if the memory is guaranteed to be accessible. To decide this, the invalid pointer needs to be compared with a definitely valid pointer. As we mentioned in Section 5.1, such a comparison renders valid the invalid pointer. This means that dereferences of invalid pointers are always unsafe. We let verification fail if unsafe accesses are performed. That performance-critical and lock-free code is free from unsafe accesses was validated experimentally by Haziza et al. [2016] and is confirmed by our experiments.

Definition 5.10 (Unsafe Access). A computation $\tau.(t, com, up)$ performs an *unsafe access* if *com* contains p.`data` or p.`next` with $p \notin valid_\tau$.

Forbidding unsafe accesses makes the computation similarity strong enough to mimic all desired actions. A discussion of cases where the observer behavior inclusion cannot be preserved is in order. We start with an example.

Example 5.11 (Continued). Consider the following continuations of τ_1 and σ_1 from Example 5.1:

$$\begin{aligned} & \tau_5 = \tau_1 . (t, \texttt{retire}(p), \varnothing) \quad \text{with} \quad \mathcal{H}(\tau_5) = \mathcal{H}(\tau_1) . \texttt{retire}(t, a) \\ \text{and} \quad & \sigma_5 = \sigma_1 . (t, \texttt{retire}(p), \varnothing) \quad \text{with} \quad \mathcal{H}(\sigma_5) = \mathcal{H}(\sigma_1) . \texttt{retire}(t, b) \,. \end{aligned}$$

The observer behavior of τ_1 is included in σ_1, $\tau_1 \lessdot \sigma_1$. After τ_5 a deletion of a is possible because it was retired. After σ_5 a deletion of a is prevented by O_{Base} because a was not retired. Technically, we have $\texttt{free}(a) \in \mathcal{F}(\tau_5, a)$ and $\texttt{free}(a) \notin \mathcal{F}(\sigma_5, a)$. However, a is a valid address because it is referenced by the valid pointer q. That is, the behavior inclusion among τ_1 and σ_1 is not preserved by the subsequent action.

The above example showcases that calls to the SMR algorithm can break the observer behavior inclusion. This is the case because an action can emit different events in similar computations. The event emitted by an SMR call differs only if it involves invalid pointers.

The naive solution would prevent using invalid pointers in calls altogether. In practice, this is too strong a requirement. As discussed in Section 2, a common pattern for protecting an address (cf. Figure 2, Lines 26 to 28) is to (i) read a pointer p into a local variable q, (ii) issue a protection using q, and (iii) repeat the process if p and q do not coincide. After reading into q and before protecting q the referenced memory may be freed. Hence, the protection is prone to use invalid pointers. Forbidding such protections would render our theory inapplicable to lock-free data structures using hazard pointers.

To fight this problem, we forbid only those calls involving invalid pointers which are prone to *break* the observer behavior inclusion. Intuitively, this is the case if a call with the values of the invalid pointers replaced arbitrarily allows for more behavior on the valid addresses than the original call. Actually, we keep precise the address the behavior of which is under consideration. This allows us to support more scenarios where invalid pointers are used.

Definition 5.12 (Racy SMR Calls). A computation $\tau.act$ with $act = (t, \texttt{enter}\,func(\bar{p}, \bar{x}), \varnothing)$, $\mathcal{H}(\tau) = h$, $m_\tau(\bar{p}) = \bar{a}$, and $m_\tau(\bar{x}) = \bar{d}$ performs a *racy call* if:

$$\exists\, c\ \exists\, \bar{b}.\ \big(\forall i.\ (a_i = c \lor p_i \in valid_\tau) \implies a_i = b_i\big) \land \mathcal{F}(h.func(t, \bar{b}, \bar{d}), c) \not\subseteq \mathcal{F}(h.func(t, \bar{a}, \bar{d}), c) \,.$$

It follows immediately that calls containing valid pointers only are not racy. In practice, `retire` is always called using valid pointers, thus avoiding the problematic scenario from Example 5.11. The rational behind this is that freeing invalid pointers may lead to system crashes, just like dereferences. For `protect` calls of hazard pointers one can show that they never race. We have already seen this in Example 5.6. There, a call to `protect` with invalid pointers has not caused the observer relation to break. Instead, the mimicking computation (σ_3) could perform (strictly) more frees than the computation it mimicked (τ_3).

We uniformly refer to the above situations where the usage of an invalid pointer can break the ability to mimic an actions as a *pointer race*. It is a race indeed because the usage and the free of a pointer are not properly synchronized.

Definition 5.13 (Pointer Race). A computation $\tau.act$ is a *pointer race* if *act* performs (i) an unsafe access, or (ii) a racy SMR call.

With pointer races we restrict the class of supported programs. The restriction to pointer race free programs is reasonable in that we can handle common non-blocking data structures from the literature as shown in our experiments. Since we want to give the main result of this section in a general fashion that does not rely on the actual observer used to specify the SMR implementation, we have to restrict the class of supported observers as well.

We require that the observer supports the elision of reused addresses, as done in Example 5.1. Intuitively, elision is a two-step process the observer must be insensitive to. First, an address a is replaced with a fresh address b upon an allocation where a should be reused but cannot. In the resulting computation, a is fresh and thus the allocation can be performed without reusing a. The process of replacing a with b must not affect the behavior of the observer on addresses other than a and b. Second, the observer must allow for more behavior on the fresh address than on the reused address. This is required to preserve the observer behavior inclusion because the allocation of a renders it a valid address.

Additionally, we require a third property: the observer behavior on an address must not be influenced by frees to another address. This is needed because computation similarity and behavior inclusion do not guarantee that frees of invalid addresses can be mimicked, as discussed before. Since such frees do not affect the valid memory they need not be mimicked. The observer has to allow us to do so, that is, simply *skip* such frees when mimicking a computation.

For a formal definition of our intuition we write $h[a/b]$ to denote the history that is constructed from h by replacing every occurrence of a with b.

Definition 5.14 (Elision Support). The observer O_{SMR} *supports elision of memory reuse* if

(i) for all h_1, h_2, a, b, c with $a \neq c \neq b$ and $h_2 = h_1[a/b]$ we have $\mathcal{F}(h_1, c) = \mathcal{F}(h_2, c)$,

(ii) for all h_1, h_2, a, b with $\mathcal{F}(h_1, a) \subseteq \mathcal{F}(h_2, a)$ and $b \in \mathit{fresh}(h_2)$ we have $\mathcal{F}(h_1, b) \subseteq \mathcal{F}(h_2, b)$, and

(iii) for all h, a, b with $a \neq b$ we have $\mathcal{F}(h.\texttt{free}(a), b) = \mathcal{F}(h, b)$.

We found this definition practical in that the observers we use for our experiments support elision (cf. Section 6.1). The hazard pointer observer $O_{Base} \times O_{HP}$, for instance, supports elision.

5.3 Detecting ABAs

So far we have introduced restrictions, namely pointer race freedom and elision support, to rule out cases where our idea of eliding memory reuse would not work, that is, break the similarity or behavior inclusion. If those restrictions were strong enough to carry out our development, then we could remove any reuse from a computation and get a similar one where no memory is reused. That the resulting computation does not reuse memory means, intuitively, that it is executed under garbage collection. As shown in the literature [Michael and Scott 1996], the ABA problem is a subtle bug caused by manual memory management which is prevented by garbage collection. So eliding all reuses jeopardizes soundness of the analysis—it could miss ABAs which result in a safety violation. With this observation, we elide all reuses except for one address per computation. This way we analyse a semantics that is close to garbage collection, can detect ABA problems, and is much simpler than full $[\![D(O_{SMR})]\!]_{Adr}$.

The semantics that we suggest to analyse is $[\![D(O_{SMR})]\!]_{one} := \bigcup_{a \in Adr} [\![D(O_{SMR})]\!]_{\{a\}}$. It is the set of all computations that reuse at most a single address. A single address suffices to detect the

ABA problem. The ABA problem manifests as an assertion of the form `assert` $p = q$ where the addresses held by p and q coincide but stem from different allocations. That is, one of the pointers has received its address, the address was freed and then reallocated, before the pointer is used in the assertion. Note that this implies that for an assertion to be ABA one of the involved pointers must be invalid. Pointer race freedom does not forbid this. Nor do we want to forbid such assertions. In fact, most programs using hazard pointers contain ABAs. They are written in a way that ensures that the ABA is *harmless*. Consider an example.

Example 5.15 (ABAs in Michael&Scott's queue using hazard pointers). Consider the code of Michael&Scott's queue from Figure 2. More specifically, consider Lines 26 to 28. In Line 26 the value of the shared pointer `Head` is read into the local pointer `head`. Then, a hazard pointer is used in Line 27 to protect `head` from being freed. In between reading and protecting `head`, its address could have been deleted, reused, and reentered the queue. That is, when executing Line 28 the pointers `Head` and `head` can coincide although the `head` pointer stems from an earlier allocation. This scenario is an ABA. Nevertheless, the queue's correctness is not affected by this ABA. The ABA prone assertion is only used to guarantee that the address protected in Line 27 is indeed protected after Line 28. Wrt. to the observer $\mathcal{O}_{HP}$ from Figure 6, the assertion guarantees that the protection was issued before a retirement (after the latest reallocation) so that $\mathcal{O}_{HP}$ is guaranteed to be in l_{11} and thus prevent future retirements from freeing the protected memory. The ABA does not void this guarantee, it is harmless.

The above example shows that lock-free data structures may perform ABAs which do not affect their correctness. To soundly verify such algorithms, our approach is to detect every ABA and decide whether it is harmless indeed. If so, our verification is sound. Otherwise, we report to the programmer that the implementation suffers from a harmful ABA problem.

A discussion of how to detect ABAs is in order. Let $\tau \in [\![D(\mathcal{O}_{SMR})]\!]_{Adr}$ and $\sigma \in [\![D(\mathcal{O}_{SMR})]\!]_{\{a\}}$ be two similar computations. Intuitively, σ is a computation which elides the reuses from τ except for some address a. The address a can be used in σ in exactly the same way as it is used in τ. Let $act = (t, \texttt{assert}\ p = q, \varnothing)$ be an ABA assertion which is enabled after τ. To detect this ABA under $[\![D(\mathcal{O}_{SMR})]\!]_{\{a\}}$ we need act to be enabled after σ. We seek to have $\sigma.act \in [\![D(\mathcal{O}_{SMR})]\!]_{\{a\}}$. This is not guaranteed. Since act is an ABA it involves at least one invalid pointer, say p. Computation similarity does not guarantee that p has the same valuation in both τ and σ. However, if p points to a in τ, then it does so in σ because a is (re)used in σ in the same way as in τ. Thus, we end up with $m_\tau(p) = m_\sigma(p)$ although p is invalid. In order to guarantee this, we introduce a *memory equivalence* relation. We use this relation to precisely track how the reusable address a is used.

Definition 5.16 (Memory Equivalence). Two computations τ and σ are *memory equivalent wrt. address a*, denoted by $\tau \simeq_a \sigma$, if

$$\begin{aligned}
&\forall\, p \in PVar.\ m_\tau(p) = a \iff m_\sigma(p) = a \\
\text{and}\quad &\forall\, b \in m_\tau(valid_\tau).\ m_\tau(b.\texttt{next}) = a \iff m_\sigma(b.\texttt{next}) = a \\
\text{and}\quad &a \in fresh(\tau) \cup freed(\tau) \iff a \in fresh(\sigma) \cup freed(\sigma) \\
\text{and}\quad &\mathcal{F}(\tau, a) \subseteq \mathcal{F}(\sigma, a)\ .
\end{aligned}$$

The first line in this definition states that the same pointer variables in τ and σ are pointing to a. Similarly, the second line states this for the pointer selectors of valid addresses. We have to exclude the invalid addresses here because τ and σ may differ on the in-use addresses due to eliding reuse. The third line states that a can be allocated in τ iff it can be allocated in σ. The last line states that the observer allows for more behavior on a in σ than in τ. These properties combined guarantee that σ can mimic actions of τ involving a no matter if invalid pointers are used.

The memory equivalence lets us detect ABAs in $[\![D(O_{SMR})]\!]_{one}$. Intuitively, we can only detect *first* ABAs because we allow for only a single address to be reused. Subsequent ABAs on different addresses cannot be detected. To detect ABA sequences of arbitrary length, an arbitrary number of reusable addresses is required. To avoid this, i.e., to avoid an analysis of full $[\![D(O_{SMR})]\!]_{Adr}$, we formalize the idea of *harmless ABA* from before. We say that an ABA is harmless if executing it leads to a system state which can be explored without performing the ABA. That the system state can be explored without performing the ABA means that every ABA is also a first ABA. Thus, any sequence of ABAs is explored by considering only first ABAs. Note that this definition is independent of the actual correctness notion.

Definition 5.17 (Harmful ABA). $[\![D(O_{SMR})]\!]_{one}$ is *free from harmful ABAs* if the following holds:

$$\forall\, \sigma_a.act \in [\![D(O_{SMR})]\!]_{\{a\}}\ \forall\, \sigma_b \in [\![D(O_{SMR})]\!]_{\{b\}}\ \exists\, \sigma'_b \in [\![D(O_{SMR})]\!]_{\{b\}}.$$
$$\sigma_a \sim \sigma_b\ \wedge\ act = (_, \texttt{assert}\ _, _) \implies \sigma_a.act \sim \sigma'_b\ \wedge\ \sigma_b \simeq_b \sigma'_b\ \wedge\ \sigma_a.act \lessdot \sigma'_b\ .$$

To understand how the definition implements our intuition, consider some $\tau.act \in [\![D(O_{SMR})]\!]_{Adr}$ where *act* performs an ABA on address a. Our goal is to mimic $\tau.act$ in $[\![D(O_{SMR})]\!]_{\{b\}}$, that is, we want to mimic the ABA without reusing address a (for instance, to detect subsequent ABAs on address b). Assume we are given some $\sigma_b \in [\![D(O_{SMR})]\!]_{\{b\}}$ which is similar and memory equivalent wrt. b to τ. This does not guarantee that *act* can be mimicked after σ_b; the ABA may not be enabled because it involves invalid pointers the valuation of which may differ in τ and σ_b. However, we can construct a computation σ_a which is similar and memory equivalent wrt. a to τ. After σ_a the ABA is enabled, i.e., we have $\sigma_a.act \in [\![D(O_{SMR})]\!]_{\{a\}}$. For those two computations $\sigma_a.act$ and σ_b we invoke the above definition. It yields another computation $\sigma'_b \in [\![D(O_{SMR})]\!]_{\{b\}}$ which, intuitively, coincides with σ_b but where the ABA has already been executed. Put differently, σ'_b is a computation which mimics the execution of *act* after σ_b (although *act* is not enabled).

Example 5.18 (Continued). Consider the following computation of Michael&Scott's queue:

$$\tau = \tau_6\,.\,(t, \mathsf{head} := \mathsf{Head}, [\mathsf{head} \mapsto a])\,.\,\tau_7\,.\,\texttt{free}(a)\,.\,\tau_8\,.$$
$$(t, \texttt{enter protect}(\mathsf{head}, 0), \varnothing)\,.\,(t, \texttt{exit}, \varnothing)\,.\,(t, \texttt{assert}\ \mathsf{head} = \mathsf{Head}, \varnothing)\ .$$

This computation resembles a thread t executing Lines 26 to 28 while an interferer frees address a referenced by head. Note that the `assert` resembles the conditional from Line 28 and states that the condition evaluates to *true*. That is, the last action in τ is an ABA.

Reusing address a allows us to mimic τ with a computation $\sigma_a \in [\![D(O_{SMR})]\!]_{\{a\}}$ which detects the ABA. For simplicity, assume $\tau = \sigma_a$. Mimicking τ with another computation $\sigma_b \in [\![D(O_{SMR})]\!]_{\{b\}}$ is not possible. In $\sigma_b \in [\![D(O_{SMR})]\!]_{\{b\}}$ the first allocation of a will be elided such that the assertion is not enabled. However, rescheduling the actions gives rise to $\sigma'_b \in [\![D(O_{SMR})]\!]_{\{b\}}$ which coincides with σ_b but where the assertion has also been executed:

$$\sigma'_b = \tau_6\,.\,\tau_7\,.\,\texttt{free}(a)\,.\,\tau_8\,.\,(t, \mathsf{head} := \mathsf{Head}, [\mathsf{head} \mapsto a])\,.$$
$$(t, \texttt{enter protect}(\mathsf{head}, 0), \varnothing)\,.\,(t, \texttt{exit}, \varnothing)\,.\,(t, \texttt{assert}\ \mathsf{head} = \mathsf{Head}, \varnothing)\ .$$

Requiring the existence of such a σ'_b guarantees that an analysis can *see past* ABAs on address a, although a is not reused.

A key aspect of the above definition is that checking for harmful ABAs can be done in the simpler semantics $[\![D(O_{SMR})]\!]_{one}$. Altogether, this means that we can rely on $[\![D(O_{SMR})]\!]_{one}$ for both the actual analysis and a soundness (absence of harmful ABAs) check. Our experiments show that the above definition is practical. There were no harmful ABAs in the benchmarks we considered.

5.4 Reduction Result

We show how to exploit the concepts introduced so far to soundly verify safety properties in the simpler semantics $[\![D(O_{SMR})]\!]_{one}$ instead of full $[\![D(O_{SMR})]\!]_{Adr}$.

Lemma 5.19. *Let $[\![D(O_{SMR})]\!]_{one}$ be free from pointer races and harmful ABAs, and let O_{SMR} support elision. For all $\tau \in [\![D(O_{SMR})]\!]_{Adr}$ and $a \in Adr$ there is $\sigma \in [\![D(O_{SMR})]\!]_{\{a\}}$ with $\tau \sim \sigma$, $\tau \lessdot \sigma$, and $\tau \simeq_a \sigma$.*

Proof Sketch. We construct σ inductively by mimicking every action from τ and eliding reuses as needed. For the construction, consider $\tau.act \in [\![D(O_{SMR})]\!]_{Adr}$ and assume we have already constructed, for every $a \in Adr$, an appropriate $\sigma_a \in [\![D(O_{SMR})]\!]_{\{a\}}$. Consider some address $a \in Adr$. The task is to mimic *act* in σ_a. If *act* is an assignment or an SMR call, then pointer race freedom guarantees that we can mimic *act* by executing the same command with a possibly different update. We discussed this in Section 5.2. The interesting cases are ABAs, frees, and allocations.

First, consider the case where *act* executes an ABA assertion `assert` $p = q$. That the assertion is an ABA means that at least one of the pointers is invalid, say p. That is, *act* may not be enabled after σ_a. Let p point to b in τ. By induction, we have already constructed σ_b for τ. The ABA is enabled after σ_b. This is due to $\tau \simeq_b \sigma_b$. It implies that p points to b in τ iff p points to b in σ_b (independent of the validity), and likewise for q. That is, the comparison has the same outcome in both computations. Now, we can exploit the absence of harmful ABAs to find a computation mimicking $\tau.act$ for a. Applying Definition 5.17 to $\sigma_b.act$ and σ_a yields some σ'_a that satisfies the required properties.

Second, consider the case of *act* performing a `free`(b). If *act* is enabled after σ_a nothing needs to be shown. In particular, this is the case if b is a valid address or $a = b$. Otherwise, b must be an invalid address. Freeing an invalid address does not change the valid memory. It also does not change the control location of threads as frees are performed by the environment. Hence, we have $\tau.act \sim \sigma_a$. By the definition of elision support, Definition 5.14iii, the `free` does not affect the behavior of the observer on other addresses. So we get $\tau.act \lessdot \sigma_a$. With the same arguments we conclude $\tau.act \simeq_a \sigma_a$. That is, we do not need to mimic frees of invalid addresses.

Last, consider *act* executing an allocation $p :=$ `malloc` of address b. If b is fresh in σ_a or $a = b$, then *act* is enabled. The allocation makes b a valid address. That $\lessdot$ holds for this address follows from elision support, Definition 5.14ii. Otherwise, *act* is not enabled because b cannot be reused. We replace in σ_a every occurrence of b with a fresh address c. Let us denote the result with $\sigma_a[b/c]$. Relying on elision support, Definition 5.14i, one can show $\sigma_a \prec \sigma_a[b/c]$ and thus $\tau \prec \sigma_a[b/c]$ for all $\prec \in \{\sim, \lessdot, \simeq_a\}$. Since b is fresh in $\sigma_a[b/c]$, we conclude by enabledness of *act*. □

From the above follows the second result of the paper. It states that for every computation there is a similar one which reuses at most a single address. Since similarity implies control location equality, our result allows for a much simpler verification of safety properties. We stress that the result is independent of the actual observer used.

Theorem 5.20. *If $[\![D(O_{SMR})]\!]_{one}$ is pointer race free, supports elision, and is free from harmful ABAs, then $[\![D(O_{SMR})]\!]_{Adr} \sim [\![D(O_{SMR})]\!]_{one}$.*

One can generalize the above results to a strictly weaker premise, see [Meyer and Wolff 2018].

6 EVALUATION

We implemented our approach in a tool. It is a thread-modular analysis for (i) verifying linearizability of singly-linked lock-free data structures using SMR specifications, and (ii) verifying SMR implementations against their specification. In the following, we elaborate on the SMR implementations used for our benchmarks and the implemented analysis, and evaluate our tool.

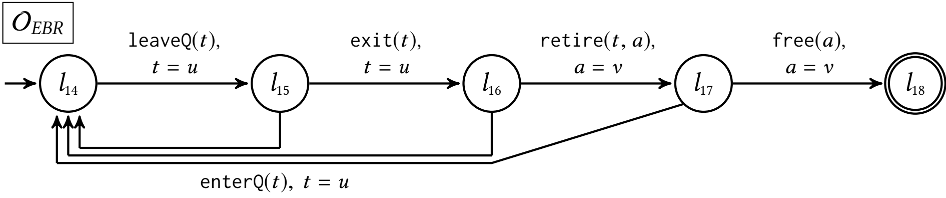


Fig. 7. Observer specifying when EBR/QSBR defers deletion using two variables, u and v, to observe a thread and an address, respectively. The observer implements the property that a cell v retired during the non-quiescent phase of a thread u may not be freed until the thread becomes quiescent. The full specification of EBR/QSBR is the observer $O_{Base} \times O_{EBR}$.

6.1 SMR Algorithms

For our experiments, we consider two well-known SMR algorithms: *Hazard Pointers (HP)* and *Epoch-Based Reclamation (EBR)*. Additionally, we include as a baseline for our experiments a simplistic *GC* SMR algorithm which does not allow for memory to be reclaimed. We already introduced HP and gave a specification (cf. Figure 6) in form of the observer $O_{Base} \times O_{HP}$. We briefly introduce EBR.

Epoch-based reclamation [Fraser 2004] relies on two assumption: (i) threads cannot have pointers to any node of the data structure in-between operation invocations, and (ii) nodes are retired only after being removed from the data structure, i.e., after being made unreachable from the shared variables. Those assumptions imply that no thread can acquire a pointer to a removed node if every thread has been in-between an invocation since the removal. So it is safe to delete a retired node if every thread has been in-between an invocation since the retire. Technically, EBR introduces *epoch counters*, a global one and one for each thread. Similar to hazard pointers, thread epochs are single-writer multiple-reader counters. Whenever a thread invokes an operation, it reads the global epoch e and announces this value by setting its thread epoch to e. Then, it scans the epochs announced by the other threads. If they all agree on e, the global epoch is set to $e + 1$. The fact that all threads must have announced the current epoch e for it to be updated to $e + 1$ means that all threads have invoked an operation after the epoch was changed from $e - 1$ to e. That is, all threads have been in-between invocations. Thus, deleting nodes retired in the global epoch $e - 1$ becomes safe from the moment when the global epoch is updated from e to $e + 1$. To perform those deletions, every thread keeps a list of retired nodes for every epoch and stores nodes passed to `retire` in the list for the current thread epoch. For the actual deletion it is important to note that the thread-local epoch may lack behind the global epoch by up to 1. As a consequence, a thread may put a node retired during the global epoch e into its retire-list for epoch $e - 1$. So for a thread during its local epoch e. it is not safe to delete the nodes in the retired-list for $e - 1$ because it may have been retired during the global epoch e. It is only safe to delete the nodes contained in the retired-list for epochs $e - 2$ and smaller. Hence, it suffices to maintaining three retire-lists. Progressing to epoch $e + 1$ allows for deleting the nodes from the local epoch $e - 2$ and to reuse that retire-list for epoch $e + 1$.

Quiescent-State-Based Reclamation (QSBR) [McKenney and Slingwine 1998] generalizes EBR by allowing the programmer to manually identify when threads are *quiescent*. A thread is quiescent if it does not hold pointers to any node from the data structure. Threads signal this by calling `enterQ` and `leaveQ` upon entering and leaving a quiescent phase, respectively.

We use the observer $O_{Base} \times O_{EBR}$ from Figure 7 for specifying both EBR and QSBR (they have the same specification). As for HP, we use O_{Base} to ensure that only those addresses are freed that have been retired. Observer O_{EBR} implements the actual EBR/QSBR behavior described above.

To use HP and EBR with our approach for restricting reuse during an analysis, we have to show that their observers support elision. This is established by the following lemma. We consider it future work to extend our tool such that it performs the appropriate checks automatically.

LEMMA 6.1. *The observers $O_{Base} \times O_{HP}$ and $O_{Base} \times O_{EBR}$ support elision.*

6.2 Thread-Modular Linearizability Analysis

Proving a data structure correct for an arbitrary number of client threads requires a thread-modular analysis [Berdine et al. 2008; Jones 1983]. Such an analysis abstracts a system state into so-called *views*, partial configurations reflecting a single thread's perception of the system state. A view includes a thread's program counter and, in the case of shared-memory programs, the memory reachable from the shared and thread-local variables. An analysis then saturates a set V of reachable views. This is done by computing the least solution to the recursive equation $V = V \cup seq(V) \cup int(V)$. Function *seq* computes a *sequential step*, the views obtained from letting each thread execute an action on its own views. Function *int* accounts for *interference* among threads. It updates the shared memory of views by actions from other threads. We follow the analysis from Abdulla et al. [2013, 2017]. There, *int* is computed by combining two views, letting one thread perform an action, and projecting the result to the other thread. More precisely, computing $int(V)$ requires for every pair of views $v_1, v_2 \in V$ to (i) compute a combined view ω of v_1 and v_2, (ii) perform for ω a sequential step for the thread of v_2, and (iii) project the result of the sequential step to the perception of the thread from v_1. This process is required only for views v_1 and v_2 that *match*, i.e., agree on the shared memory both views have in common. Otherwise, the views are guaranteed to reflect different system states. Thus, interference is not needed for an exhaustive state space exploration.

To check for linearizability, we assume that the program under scrutiny is annotated with linearization points, points at which the effect of operations take place logically and become visible to other threads. Whether or not the sequence of emitted linearization points is indeed linearizable can be checked using an observer automaton implementing the desired specification [Abdulla et al. 2013, 2017], in our case the one for stacks and queues. The state of this automaton is stored in the views. If a final state is reached, verification fails.

For brevity, we omit a discussion of the memory abstraction we use. It is orthogonal to the analysis. For more details, we refer the reader to [Abdulla et al. 2013, 2017]. We are not aware of another memory abstraction which can handle reuse and admits automation.

We extend the above analysis by our approach to integrate SMR and restrict reuse to a single address. To integrate SMR, we add the necessary observers to views. Note that observers have a pleasant interplay with thread-modularity. In a view for thread t only those observer states are required where t is observed. For the hazard pointer observer $O_{Base} \times O_{HP}$ this means that only observer states with u capturing t need to be stored in the view for t. Similarly, the memory abstraction induces a set of addresses that need to be observed (by v). However, we do not keep observer states for *shared* addresses, i.e., addresses that are reachable from the shared variables. Instead, we maintain the invariant that they are never retired nor freed. For the analysis, we then assume that the ignored observer states are arbitrary (but not in observer locations implying retiredness or freedness of shared addresses). We found that all benchmark programs satisfied this invariant and that the resulting precision allowed for successful verification. Altogether, this keeps the number of observer states per view small in practice.

As discussed in Section 4, observers $O_{Base} \times O_{HP}$ and $O_{Base} \times O_{EBR}$ assume that a client does not perform double-retires. We integrate a check for this invariant, relying on observer O_{Base}: if O_{Base} is in a state (l_8, φ), then a double-retire occurs if an event of the form `retire`$(_, \varphi(v))$ is emitted.

To guarantee that the restriction of reuse to a single cell is sound, we have to check for pointer races and harmful ABAs. To check for pointer races we annotate views with validity information. This information is updated accordingly during sequential and interference steps. If a pointer race is detected, verification fails. For this check, we rely on Lemma 6.2 below and deem racy any invocation of `retire` with invalid pointers. That is, the pointer race check boils down to scanning dereferences and `retire` invocations for invalid pointers.

Lemma 6.2. *If a call is racy wrt.* $O_{Base} \times O_{EBR}$ *or* $O_{Base} \times O_{HP}$*, then it is a call of function* `retire` *using an invalid pointer.*

Last, we add a check for harmful ABAs on top of the state space exploration. This check has to implement Definition 5.17. That a computation $\sigma_a.act$ contains a harmful ABA can be detected in the view v_a for thread t which performs act. Like for computations, the view abstraction v_b of σ_b for t cannot perform the ABA. To establish that the ABA is harmless, we seek a v_b' which is similar to v_a, memory equivalent to v_b, and includes the observer behavior of v_b. (The relations introduced in Section 5 naturally extend from computations to views.) If no such v_b' exists, verification fails.

In the thread-modular setting one has to be careful with the choice of v_b'. It is not sufficient to find *just some* v_b' satisfying the desired relations. The reason lies in that we perform the ABA check on a thread-modular abstraction of computations. To see this, assume the view abstraction of σ_b is $\alpha(\sigma_b) = \{v_b, v\}$ where v_b is the view for thread t which performs the ABA in $\sigma_a.act$. For *just some* v_b' it is not guaranteed that there is a computation σ_b' such that $\alpha(\sigma_b') = \{v_b', v\}$. The sheer existence of v_b' and v in V does not guarantee that there is a computation the abstraction of which yields those two views. Put differently, we cannot construct computations from views. Hence, a simple search for v_b' cannot prove the existence of the required σ_b'.

To overcome this problem, we use a method to search for a v_b' that guarantees the existence of σ_b'; in terms of the above example, guarantees that there is σ_b' with $\alpha(\sigma_b') = \{v_b', v\}$. We take the view v_b that cannot perform the ABA. We apply sequential steps to v_b until it *comes back* to the same program counter. The rational behind is that ABAs are typically conditionals that restart the operation if the ABA is not executable. Restarting the operation results in reading out pointers anew (this time without interference from other threads). Consequently, the ABA is now executable. The resulting view is a candidate for v_b'. If it does not satisfy Definition 5.17, verification fails. Although simple, this approach succeeded in all benchmarks.

6.3 Linearizability Experiments

We implemented the above analysis in a `C++` tool.[3] We evaluated the tool on singly-linked lock-free data structures from the literature, like Treiber's stack [Michael 2002; Treiber 1986], Michael&Scott's lock-free queue [Michael 2002; Michael and Scott 1996], and the DGLM lock-free queue [Doherty et al. 2004b]. The findings are listed in Table 1. They include (i) the time taken for verification, i.e., to explore exhaustively the state space and check linearizability, (ii) the size of the explored state space, i.e., the number of reachable views, (iii) the number of ABA prone views, i.e., views where a thread is about to perform an `assert` containing an invalid pointer, (iv) the time taken to establish that no ABA is harmful, and (v) the verdict of the linearizability check. The experiments were conducted on an Intel Xeon X5650@2.67GHz running Ubuntu 16.04 and using Clang version 6.0.

Our approach is capable of verifying lock-free data structures using HP and EBR. We were able to automatically verify Treiber's stack, Michael&Scott's queue, and the DGLM queue. To the best of our knowledge, we are the first to verify data structures using the aforementioned SMR algorithms

[3] Available at: https://github.com/Wolff09/TMRexp/releases/tag/POPL19-chkds

Table 1. Experimental results for verifying singly-linked data structures using SMR. The experiments were conducted on an Intel Xeon X5650@2.67GHz running Ubuntu 16.04 and using Clang 6.0.

Program[a]	SMR	Time Verif.	States	ABAs	Time ABA	Linearizable
Coarse stack	GC	0.44*s*	300	0	0*s*	yes
	None	0.5*s*	300	0	0*s*	yes
Coarse queue	GC	1.7*s*	300	0	0*s*	yes
	None	1.7*s*	300	0	0*s*	yes
Treiber's stack	GC	3.2*s*	806	0	0*s*	yes
	EBR	16*s*	1822	0	0*s*	yes
	HP	19*s*	2606	186	0.06*s*	yes
Opt. Treiber's stack	HP	0.8*s*	—	—	—	no[b]
Michael&Scott's queue	GC	414*s*	2202	0	0*s*	yes
	EBR	2630*s*	7613	0	0*s*	yes
	HP	7075*s*	19028	536	0.9*s*	yes
DGLM queue	GC	714*s*	9934	0	0*s*	yes[c]
	EBR	3754*s*	27132	0	0*s*	yes[c]
	HP	7010*s*	41753	2824	26*s*	yes[c]

[a] The code for the benchmark programs can be found in [Meyer and Wolff 2018].
[b] Pointer race due to an ABA in push: the `next` pointer of the new node becomes invalid and the CAS succeeds.
[c] Imprecision in the memory abstraction required hinting that `Head` cannot overtake `Tail` by more than one node.

fully automatically. Moreover, we are also the first to verify automatically the DGLM queue under any manual memory management technique.

An interesting observation throughout the entire test suite is that the number of ABA prone views is rather small compared to the total number of reachable views. Consequently, the time needed to check for harmful ABAs is insignificant compared to the verification time. This substantiates the usefulness of *ignoring* ABAs during the actual analysis and checking afterwards that no harmful ABA exists.

Our tool could not establish linearizability for the optimized version of Treiber's stack with hazard pointers by Michael [2002]. The reason for this is that the push operation does not use any hazard pointers. This leads to pointer races and thus verification failure although the implementation is correct. To see why, consider the code of push from Figure 8. The operation allocates a new node, reads the top-of-stack pointer into a local variable `top` in Line 102, links the new node to the top-of-stack in Line 103, and swings the top-of-stack pointer to the new node in Line 104. Between Line 102 and Line 103 the node referenced by `top` can be popped, reclaimed, reused, and reinserted by an interferer. That is, the CAS in Line 104 is ABA prone. The reclamation of the node referenced by `top` renders both the `top` pointer and the `next` field of the new node invalid. As discussed

```
96  struct Node {/* ... */}
97  shared Node* ToS;
98  void push(data_t input) {
99    Node* node = new Node();
100   node->data = input;
101   while (true) {
102     Node* top = ToS;
103     node->next = top;
104     if (CAS&ToS, top, next)
105       break;
106   }
```

Fig. 8. The push operation of Treiber's lock-free stack [Treiber 1986].

Table 2. Experimental results for verifying SMR implementations against their observer specifications. The experiments were conducted on an Intel Xeon X5650@2.67GHz running Ubuntu 16.04 and using Clang 6.0.

SMR Implementation[a]	Specification	Verification Time	States	Correct
Hazard Pointers	$\mathcal{O}_{Base} \times \mathcal{O}_{HP}$	$1.5s$	5437	yes
Epoch-Based Reclamation	$\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$	$11.2s$	11528	yes

[a]The code for the benchmark programs can be found in [Meyer and Wolff 2018].

in Section 5, the comparison of the valid `ToS` with the invalid `top` makes `top` valid again. However, the `next` field of the new node remains invalid. That is, the `push` succeeds and leaves the stack in a state with `ToS->next` being invalid. This leads to pointer races because no thread can acquire valid pointers to the nodes following `ToS`. Hence, reading out data of such subsequent nodes in the `pop` procedure, for example, raises a pointer race.

To solve this issue, the `CAS` in Line 104 has to validate the pointer `node->next`. One could annotate the `CAS` with an invariant `Tos == node-> next`. Treating invariants and assertions alike would then result in the `CAS` validating `node->next` (cf. Section 5.1) as desired. That the annotation is an invariant indeed, could be checked during the analysis. We consider a proper investigation as future work.

For the DGLM queue, our tool required hints. The DGLM queue is similar to Michael&Scott's queue but allows the `Head` pointer to overtake the `Tail` pointer by at most one node. Due to imprecision in the memory abstraction, our tool explored states with malformed lists where `Head` overtook `Tail` by more than one node. We implemented a switch to increase the precision of the abstraction and ignore cases where `Head` overtakes `Tail` by more than one node. This allowed us to verifying the DGLM queue. While this change is ad hoc, it does not jeopardize the principledness of our approach because it affects only the memory abstraction which we took from the literature.

6.4 Verifying SMR Implementations

It remains to verify that a given SMR implementation is correct wrt. an observer $\mathcal{O}_{SMR}$. As noted in Section 4, an SMR implementation can be viewed as a lock-free data structure where the stored *data* are pointers. Consequently, we can reuse the above analysis. We extended our implementation with an abstraction for (sets of) data values.[4] The main insight for a concise abstraction is that it suffices to track a single observer state per view. If the SMR implementation is not correct wrt. $\mathcal{O}_{SMR}$, then by definition there is $\tau \in [\![MGC(R)]\!]_{Adr}$ with $\mathcal{H}(\tau) \notin \mathcal{S}(\mathcal{O}_{SMR})$. Hence, there must be some observer state s with $\mathcal{H}(\tau) \notin \mathcal{S}(s)$. Consider the observers $\mathcal{O}_{Base} \times \mathcal{O}_{HP}$ and $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ where s is of the form $s = (l, \{u \mapsto t, v \mapsto a, w \mapsto i\})$. This single state s induces a simple abstraction of data values d: either $d = a$ or $d \neq a$. Similarly, an abstraction of sets of data values simply tracks whether or not the set contains a.

To gain adequate precision, we retain in every view the thread-local pointers of t. Wrt. Figure 3, this keeps the thread-t-local `HPRec` in every view. It makes the analysis recognize that t has indeed protected a. Moreover, we store in every view whether or not the last `retire` invocation stems from the thread of that view. With this information, we avoid unnecessary matches during interference of views v_1 and v_2: if both threads t_1 of v_1 and t_2 of v_2 have performed the last `retire` invocation, then t_1 and t_2 are the exact same thread. Hence, interference is not needed as threads have unique identities. We found this extension necessary to gain the precision required to verify our benchmarks.

[4]Available at: https://github.com/Wolff09/TMRexp/releases/tag/POPL19-chksmr

Table 2 shows the experimental results for the HP implementation from Figure 3 and an EBR implementation. Both SMR implementations allow threads to dynamically join and part. We conducted the experiments in the same setup as before. As noted in Section 4, the verification is simpler and thus more efficient than the previous one. The reason for this is the absence of memory reclamation.

7 RELATED WORK

We discuss the related work on SMR implementations and on the verification of linearizability.

Safe Memory Reclamation. Besides HP and EBR further SMR algorithms have been proposed in the literature. *Free-lists* is the simplest such mechanism. Retired nodes are stored in a thread-local free-list. The nodes in this list are never reclaimed. Instead, a thread can reuse nodes instead of allocating new ones. *Reference Counting (RC)* adds to nodes a counter representing the number of pointers referencing that node. Updating such counters safely in a lock-free fashion, however, requires the use of hazard pointers [Herlihy et al. 2005] or double-word CAS [Detlefs et al. 2001], which is not available on most hardware. Besides free-lists and RC, most SMR implementations from the literature combine techniques. For example, *DEBRA* [Brown 2015] is an optimized QSBR implementation. Harris [2001] extends EBR by adding epochs to nodes to detect when reclamation is safe. *Cadence* [Balmau et al. 2016], the work by Aghazadeh et al. [2014], and the work by Dice et al. [2016] are HP implementations improving on the original implementation due to Michael [2002]. *ThreadScan* [Alistarh et al. 2015], *StackTrack* [Alistarh et al. 2014], and *Dynamic Collect* [Dragojevic et al. 2011] borrow the mechanics of hazard pointers to protect single cells at a time. *Drop the Anchor* [Braginsky et al. 2013], *Optimistic Access* [Cohen and Petrank 2015b], *Automatic Optimistic Access* [Cohen and Petrank 2015a], *QSense* [Balmau et al. 2016], *Hazard Eras* [Ramalhete and Correia 2017], and *Interval-Based Reclamation* [Wen et al. 2018] are combinations of EBR and HP. *Beware&Cleanup* [Gidenstam et al. 2005] is a combination of HP and RC. *Isolde* [Yang and Wrigstad 2017] is an implementation of EBR and RC. We omit *Read-Copy-Update (RCU)* here because it does not allow for non-blocking deletion [McKenney 2004]. While we have implemented an analysis for EBR and HP only, we believe that our approach can handle most of the above works with little to no modifications. We refer the reader to [Meyer and Wolff 2018] for a more detailed discussion.

Linearizability. Linearizability of lock-free data structures has received considerable attention over the last decade. The proposed techniques for verifying linearizability can be classified roughly into (i) testing, (ii) non-automated proofs, and (iii) automated proofs. Linearizability testing [Burckhardt et al. 2010; Cerný et al. 2010; Emmi and Enea 2018; Emmi et al. 2015; Horn and Kroening 2015; Liu et al. 2009, 2013; Lowe 2017; Travkin et al. 2013; Vechev and Yahav 2008; Yang et al. 2017; Zhang 2011] enumerates an incomplete portion of the state space and checks whether or not the discovered computations are linearizable. This approach is useful for bug-hunting and for analyzing huge code bases. However, it cannot prove an implementation linearizable as it might miss non-linearizable computations.

In order to prove a given implementation linearizable, one can conduct a manual (pen&paper) or a mechanized (tool-supported, not automated) proof. Such proofs are cumbersome and require a human to have a deep understanding of both the implementation under scrutiny and the verification technique used. Common verification techniques are program logics and simulation relations. Since our focus lies on automated proofs, we do not discuss non-automated proof techniques in detail. For a survey refer to [Dongol and Derrick 2014].

Interestingly, most non-automated proofs of lock-free code rely on a garbage collector [Bäumler et al. 2011; Bouajjani et al. 2017; Colvin et al. 2005, 2006; Delbianco et al. 2017; Derrick et al. 2011; Doherty and Moir 2009; Elmas et al. 2010; Groves 2007, 2008; Hemed et al. 2015; Jonsson 2012;

Khyzha et al. 2017; Liang and Feng 2013; Liang et al. 2012, 2014; O'Hearn et al. 2010; Sergey et al. 2015a,b] to avoid the complexity of memory reclamation. Henzinger et al. [2013]; Schellhorn et al. [2012] verify a lock-free queue by Herlihy and Wing [1990] which does not reclaim memory.

There is less work on manual verification in the presence of reclamation. Doherty et al. [2004b] verify a lock-free queue implementation using tagged pointers and free-lists. Dodds et al. [2015] verify a time-stamped stack using tagged pointers. Krishna et al. [2018] verify Harris' list [Harris 2001] with reclamation. Finally, there are works [Fu et al. 2010; Gotsman et al. 2013; Parkinson et al. 2007; Tofan et al. 2011] which verify implementations using safe memory reclamation. With the exception of [Gotsman et al. 2013], they only consider implementations using HP. Gotsman et al. [2013] in addition verify implementations using EBR. With respect to lock-free data structures, these works prove linearizability of stacks. Unlike in our approach, data structure and SMR code are verified together. In theory, those works are not limited to such simple data structures. In practice, however, we are not aware of any work that proves linearizable more complicated implementations using HP. The (temporal) specifications for HP and EBR from Gotsman et al. [2013] are reflected in our observer automata.

Alglave et al. [2013]; Desnoyers et al. [2013]; Kokologiannakis and Sagonas [2017]; Liang et al. [2018] test RCU implementations. Gotsman et al. [2013]; Tassarotti et al. [2015] specify RCU and verify data structures using it non-automatically. We do not discuss such approaches because memory reclamation in RCU is blocking.

Automated approaches relieve a human checker from the complexity of manual/mechanized proofs. In turn, they have to automatically synthesize invariants and finitely encode all possible computations. The state-of-the-art methodology to do so is thread-modularity [Berdine et al. 2008; Jones 1983]. We have already discussed this technique in Section 6.2. Its main advantage is the ability to verify library code under an unbounded number of concurrent clients.

Similar to non-automated verification, most automated approaches assume a garbage collector [Abdulla et al. 2016; Amit et al. 2007; Berdine et al. 2008; Segalov et al. 2009; Sethi et al. 2013; Vafeiadis 2010a,b; Vechev et al. 2009; Zhu et al. 2015]. Garbage collection has the advantage of ownership: an allocation is always owned by the allocating thread, it cannot be accessed by any other thread. This allows for guiding thread-modular techniques, resulting in faster convergence times and more precise analyses. Despite this, there are many techniques that suffer from poor scalability. To counteract the state space explosion, they neglect SMR code making the programs under scrutiny simpler (and shorter).

Few works [Abdulla et al. 2013; Haziza et al. 2016; Holík et al. 2017] address the challenge of verifying lock-free data structures under manual memory management. These works assume that accessing deleted memory is safe and that tag fields are never overwritten with non-tag values. Basically, they integrate free-lists into the semantics. Their tools are able to verify Treiber's stack and Micheal&Scott's queue using tagged pointers. For our experiments we implemented an analysis based on [Abdulla et al. 2013; Haziza et al. 2016]. For our theoretical development, we borrowed the observer automata from Abdulla et al. [2013] as a specification means for SMR algorithms. From Haziza et al. [2016] we borrowed the notion of validity and pointer races. We modified the definition of validity to allow for ABAs and lifted the pointer race definition to SMR calls. This allowed us to verify lock-free data structures without the complexity of SMR implementations and without considering all possible reallocations. With our approach we could verify the DGLM queue which has not been verified automatically for manual memory management before.

To the best of our knowledge, there are no works which propose an automated approach for verifying linearizability of lock-free data structures similar to ours. We are the first to (i) propose a method for automatically verifying linearizability which decouples the verification of memory

reclamation from the verification of the data structure, and (ii) propose a method for easing the verification task by avoiding reuse of memory except for a single memory location.

8 CONCLUSION AND FUTURE WORK

In this paper, we have shown that the way lock-free data structures and their memory reclamation are implemented allows for compositional verification. The memory reclamation can be verified against a simple specification. In turn, this specification can be used to verify the data structure without considering the implementation of the memory reclamation. This breaks verification into two tasks each of which has to consider only a part of the original code under scrutiny.

However, the resulting tasks remain hard for verification tools. To reduce their complexity and make automation tractable, we showed that one can rely on a simpler semantics without sacrificing soundness. The semantics we proposed is simpler in that, instead of arbitrary reuse, only a single memory location needs to be considered for reuse. To ensure soundness, we showed how to check in such a semantics for ABAs. We showed how to tolerate certain, harmless ABAs to handle SMR implementations like hazard pointers. That is, we need to give up verification only if there are harmful ABAs, that is, true bugs.

We evaluated our approach in an automated linearizability checker. Our experiments confirmed that our approach can handle complex data structures with SMR the verification of which is beyond the capabilities of existing automatic approaches.

As future work we would like to extend our implementation to support more data structures and more SMR implementations from the literature. As stated before, this may require some generalizations of our theory.

ACKNOWLEDGMENTS

We thank the POPL'19 reviewers for their valuable feedback and suggestions for improvements.

REFERENCES

Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. 2013. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. In *TACAS (LNCS)*, Vol. 7795. Springer, 324–338. https://doi.org/10.1007/978-3-642-36742-7_23

Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. 2017. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. *STTT* 19, 5 (2017), 549–563. https://doi.org/10.1007/s10009-016-0415-4

Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. 2016. Automated Verification of Linearization Policies. In *SAS (LNCS)*, Vol. 9837. Springer, 61–83. https://doi.org/10.1007/978-3-662-53413-7_4

Zahra Aghazadeh, Wojciech M. Golab, and Philipp Woelfel. 2014. Making objects writable. In *PODC*. ACM, 385–395. https://doi.org/10.1145/2611462.2611483

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *CAV (LNCS)*, Vol. 8044. Springer, 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. 2014. StackTrack: an automated transactional approach to concurrent memory reclamation. In *EuroSys*. ACM, 25:1–25:14. https://doi.org/10.1145/2592798.2592808

Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. 2015. ThreadScan: Automatic and Scalable Memory Reclamation. In *SPAA*. ACM, 123–132. https://doi.org/10.1145/2755573.2755600

Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. 2007. Comparison Under Abstraction for Verifying Linearizability. In *CAV (LNCS)*, Vol. 4590. Springer, 477–490. https://doi.org/10.1007/978-3-540-73368-3_49

Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *SPAA*. ACM, 349–359. https://doi.org/10.1145/2935764.2935790

Simon Bäumler, Gerhard Schellhorn, Bogdan Tofan, and Wolfgang Reif. 2011. Proving linearizability with temporal logic. *Formal Asp. Comput.* 23, 1 (2011), 91–112. https://doi.org/10.1007/s00165-009-0130-y

Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. 2008. Thread Quantification for Concurrent Shape Analysis. In *CAV (LNCS)*, Vol. 5123. Springer, 399–413. https://doi.org/10.1007/978-3-540-70545-1_37

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving Linearizability Using Forward Simulations. In *CAV (2) (LNCS)*, Vol. 10427. Springer, 542–563. https://doi.org/10.1007/978-3-319-63390-9_28

Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. Drop the anchor: lightweight memory management for non-blocking data structures. In *SPAA*. ACM, 33–42. https://doi.org/10.1145/2486159.2486184

Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *PODC*. ACM, 261–270. https://doi.org/10.1145/2767386.2767436

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *PLDI*. ACM, 330–340. https://doi.org/10.1145/1806596.1806634

Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. 2010. Model Checking of Linearizability of Concurrent List Implementations. In *CAV (LNCS)*, Vol. 6174. Springer, 465–479. https://doi.org/10.1007/978-3-642-14295-6_41

Nachshon Cohen and Erez Petrank. 2015a. Automatic memory reclamation for lock-free data structures. In *OOPSLA*. ACM, 260–279. https://doi.org/10.1145/2814270.2814298

Nachshon Cohen and Erez Petrank. 2015b. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *SPAA*. ACM, 254–263. https://doi.org/10.1145/2755573.2755579

Robert Colvin, Simon Doherty, and Lindsay Groves. 2005. Verifying Concurrent Data Structures by Simulation. *Electr. Notes Theor. Comput. Sci.* 137, 2 (2005), 93–110. https://doi.org/10.1016/j.entcs.2005.04.026

Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. 2006. Formal Verification of a Lazy Concurrent List-Based Set Algorithm. In *CAV (LNCS)*, Vol. 4144. Springer, 475–488. https://doi.org/10.1007/11817963_44

Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2017. Concurrent Data Structures Linked in Time. In *ECOOP (LIPIcs)*, Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 8:1–8:30. https://doi.org/10.4230/LIPIcs.ECOOP.2017.8

John Derrick, Gerhard Schellhorn, and Heike Wehrheim. 2011. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.* 33, 1 (2011), 4:1–4:43. https://doi.org/10.1145/1889997.1890001

Mathieu Desnoyers, Paul E. McKenney, and Michel R. Dagenais. 2013. Multi-core systems modeling for formal verification of parallel algorithms. *Operating Systems Review* 47, 2 (2013), 51–65. https://doi.org/10.1145/2506164.2506174

David Detlefs, Paul Alan Martin, Mark Moir, and Guy L. Steele Jr. 2001. Lock-free reference counting. In *PODC*. ACM, 190–199. https://doi.org/10.1145/383962.384016

Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *ISMM*. ACM, 36–45. https://doi.org/10.1145/2926697.2926699

Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. A Scalable, Correct Time-Stamped Stack. In *POPL*. ACM, 233–246. https://doi.org/10.1145/2676726.2676963

Simon Doherty, David Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul Alan Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. 2004a. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA*. ACM, 216–224. https://doi.org/10.1145/1007912.1007945

Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2004b. Formal Verification of a Practical Lock-Free Queue Algorithm. In *FORTE (LNCS)*, Vol. 3235. Springer, 97–114. https://doi.org/10.1007/978-3-540-30232-2_7

Simon Doherty and Mark Moir. 2009. Nonblocking Algorithms and Backward Simulation. In *DISC (LNCS)*, Vol. 5805. Springer, 274–288. https://doi.org/10.1007/978-3-642-04355-0_28

Brijesh Dongol and John Derrick. 2014. Verifying linearizability: A comparative survey. *CoRR* abs/1410.6268 (2014). http://arxiv.org/abs/1410.6268

Aleksandar Dragojevic, Maurice Herlihy, Yossi Lev, and Mark Moir. 2011. On the power of hardware transactional memory to simplify memory management. In *PODC*. ACM, 99–108. https://doi.org/10.1145/1993806.1993821

Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. 2010. Simplifying Linearizability Proofs with Reduction and Abstraction. In *TACAS (LNCS)*, Vol. 6015. Springer, 296–311. https://doi.org/10.1007/978-3-642-12002-2_25

Michael Emmi and Constantin Enea. 2018. Sound, complete, and tractable linearizability monitoring for concurrent collections. *PACMPL* 2, POPL (2018), 25:1–25:27. https://doi.org/10.1145/3158113

Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Monitoring refinement via symbolic reasoning. In *PLDI*. ACM, 260–269. https://doi.org/10.1145/2737924.2737983

Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR (LNCS)*, Vol. 6269. Springer, 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. 2005. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. In *ISPAN*. IEEE Computer Society, 202–207. https://doi.org/10.1109/ISPAN.2005.42

Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *ESOP (LNCS)*, Vol. 7792. Springer, 249–269. https://doi.org/10.1007/978-3-642-37036-6_15

Lindsay Groves. 2007. Reasoning about Nonblocking Concurrency using Reduction. In *ICECCS*. IEEE Computer Society, 107–116. https://doi.org/10.1109/ICECCS.2007.39

Lindsay Groves. 2008. Verifying Michael and Scott's Lock-Free Queue Algorithm using Trace Reduction. In *CATS (CRPIT)*, Vol. 77. Australian Computer Society, 133–142. http://crpit.com/abstracts/CRPITV77Groves.html

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC (LNCS)*, Vol. 2180. Springer, 300–314. https://doi.org/10.1007/3-540-45414-4_21

Frédéric Haziza, Lukás Holík, Roland Meyer, and Sebastian Wolff. 2016. Pointer Race Freedom. In *VMCAI (LNCS)*, Vol. 9583. Springer, 393–412. https://doi.org/10.1007/978-3-662-49122-5_19

Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular Verification of Concurrency-Aware Linearizability. In *DISC (LNCS)*, Vol. 9363. Springer, 371–387. https://doi.org/10.1007/978-3-662-48653-5_25

Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. Aspect-Oriented Linearizability Proofs. In *CONCUR (LNCS)*, Vol. 8052. Springer, 242–256. https://doi.org/10.1007/978-3-642-40184-8_18

Maurice Herlihy, Victor Luchangco, Paul A. Martin, and Mark Moir. 2005. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.* 23, 2 (2005), 146–196. https://doi.org/10.1145/1062247.1062249

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Lukás Holík, Roland Meyer, Tomás Vojnar, and Sebastian Wolff. 2017. Effect Summaries for Thread-Modular Analysis - Sound Analysis Despite an Unsound Heuristic. In *SAS (LNCS)*, Vol. 10422. Springer, 169–191. https://doi.org/10.1007/978-3-319-66706-5_9

Alex Horn and Daniel Kroening. 2015. Faster Linearizability Checking via P-Compositionality. In *FORTE (LNCS)*, Vol. 9039. Springer, 50–65. https://doi.org/10.1007/978-3-319-19195-9_4

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619. https://doi.org/10.1145/69575.69577

Bengt Jonsson. 2012. Using refinement calculus techniques to prove linearizability. *Formal Asp. Comput.* 24, 4-6 (2012), 537–554. https://doi.org/10.1007/s00165-012-0250-7

Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. 2017. Proving Linearizability Using Partial Orders. In *ESOP (LNCS)*, Vol. 10201. Springer, 639–667. https://doi.org/10.1007/978-3-662-54434-1_24

Michalis Kokologiannakis and Konstantinos Sagonas. 2017. Stateless model checking of the Linux kernel's hierarchical read-copy-update (tree RCU). In *SPIN*. ACM, 172–181. https://doi.org/10.1145/3092282.3092287

Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the flow: compositional abstractions for concurrent data structures. *PACMPL* 2, POPL (2018), 37:1–37:31. https://doi.org/10.1145/3158125

Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *PLDI*. ACM, 459–470. https://doi.org/10.1145/2462156.2462189

Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*. ACM, 455–468. https://doi.org/10.1145/2103656.2103711

Hongjin Liang, Xinyu Feng, and Ming Fu. 2014. Rely-Guarantee-Based Simulation for Compositional Verification of Concurrent Program Transformations. *ACM Trans. Program. Lang. Syst.* 36, 1 (2014), 3:1–3:55. https://doi.org/10.1145/2576235

Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. 2018. Verification of tree-based hierarchical read-copy update in the Linux kernel. In *DATE*. IEEE, 61–66. https://doi.org/10.23919/DATE.2018.8341980

Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. 2009. Model Checking Linearizability via Refinement. In *FM (LNCS)*, Vol. 5850. Springer, 321–337. https://doi.org/10.1007/978-3-642-05089-3_21

Yang Liu, Wei Chen, Yanhong A. Liu, Jun Sun, Shao Jie Zhang, and Jin Song Dong. 2013. Verifying Linearizability via Optimized Refinement Checking. *IEEE Trans. Software Eng.* 39, 7 (2013), 1018–1039. https://doi.org/10.1109/TSE.2012.82

Gavin Lowe. 2017. Testing for linearizability. *Concurrency and Computation: Practice and Experience* 29, 4 (2017). https://doi.org/10.1002/cpe.3928

Paul E. McKenney. 2004. *Exploiting Deferred Destruction: an Analysis of Read-Copy-Update Techniques in Operating System Kernels.* Ph.D. Dissertation. Oregon Health & Science University.

Paul E. McKenney and John D. Slingwine. 1998. Read-copy Update: Using Execution History to Solve Concurrency Problems.

Roland Meyer and Sebastian Wolff. 2018. Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis. *CoRR* abs/1810.10807 (2018). http://arxiv.org/abs/1810.10807

Maged M. Michael. 2002. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC*. ACM, 21–30. https://doi.org/10.1145/571825.571829

Maged M. Michael and Michael L. Scott. 1995. *Correction of a Memory Management Method for Lock-Free Data Structures.* Technical Report. Rochester, NY, USA.

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*. ACM, 267–275. https://doi.org/10.1145/248052.248106

Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *PODC*. ACM, 85–94. https://doi.org/10.1145/1835698.1835722

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular verification of a non-blocking stack. In *POPL*. ACM, 297–302. https://doi.org/10.1145/1190216.1190261

Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *SPAA*. ACM, 367–369. https://doi.org/10.1145/3087556.3087588

Gerhard Schellhorn, Heike Wehrheim, and John Derrick. 2012. How to Prove Algorithms Linearisable. In *CAV (LNCS)*, Vol. 7358. Springer, 243–259. https://doi.org/10.1007/978-3-642-31424-7_21

Michal Segalov, Tal Lev-Ami, Roman Manevich, Ganesan Ramalingam, and Mooly Sagiv. 2009. Abstract Transformers for Thread Correlation Analysis. In *APLAS (LNCS)*, Vol. 5904. Springer, 30–46. https://doi.org/10.1007/978-3-642-10672-9_5

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015a. Mechanized verification of fine-grained concurrent programs. In *PLDI*. ACM, 77–87. https://doi.org/10.1145/2737924.2737964

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015b. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *ESOP (LNCS)*, Vol. 9032. Springer, 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Divjyot Sethi, Muralidhar Talupur, and Sharad Malik. 2013. Model Checking Unbounded Concurrent Lists. In *SPIN (LNCS)*, Vol. 7976. Springer, 320–340. https://doi.org/10.1007/978-3-642-39176-7_20

Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying read-copy-update in a logic for weak memory. In *PLDI*. ACM, 110–120. https://doi.org/10.1145/2737924.2737992

Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. 2011. Formal Verification of a Lock-Free Stack with Hazard Pointers. In *ICTAC (LNCS)*, Vol. 6916. Springer, 239–255. https://doi.org/10.1007/978-3-642-23283-1_16

Oleg Travkin, Annika Mütze, and Heike Wehrheim. 2013. SPIN as a Linearizability Checker under Weak Memory Models. In *Haifa Verification Conference (LNCS)*, Vol. 8244. Springer, 311–326. https://doi.org/10.1007/978-3-319-03077-7_21

R.Kent Treiber. 1986. *Systems programming: coping with parallelism.* Technical Report RJ 5118. IBM.

Viktor Vafeiadis. 2010a. Automatically Proving Linearizability. In *CAV (LNCS)*, Vol. 6174. Springer, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

Viktor Vafeiadis. 2010b. RGSep Action Inference. In *VMCAI (LNCS)*, Vol. 5944. Springer, 345–361. https://doi.org/10.1007/978-3-642-11319-2_25

Moshe Y. Vardi. 1987. Verification of Concurrent Programs: The Automata-Theoretic Framework. In *LICS*. IEEE Computer Society, 167–176.

Martin T. Vechev and Eran Yahav. 2008. Deriving linearizable fine-grained concurrent objects. In *PLDI*. ACM, 125–135. https://doi.org/10.1145/1375581.1375598

Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2009. Experience with Model Checking Linearizability. In *SPIN (LNCS)*, Vol. 5578. Springer, 261–278. https://doi.org/10.1007/978-3-642-02652-2_21

Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based memory reclamation. In *PPOPP*. ACM, 1–13. https://doi.org/10.1145/3178487.3178488

Albert Mingkun Yang and Tobias Wrigstad. 2017. Type-assisted automatic garbage collection for lock-free data structures. In *ISMM*. ACM, 14–24. https://doi.org/10.1145/3092255.3092274

Xiaoxiao Yang, Joost-Pieter Katoen, Huimin Lin, and Hao Wu. 2017. Verifying Concurrent Stacks by Divergence-Sensitive Bisimulation. *CoRR* abs/1701.06104 (2017). http://arxiv.org/abs/1701.06104

Shao Jie Zhang. 2011. Scalable automatic linearizability checking. In *ICSE*. ACM, 1185–1187. https://doi.org/10.1145/1985793.1986037

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *CAV (2) (LNCS)*, Vol. 9207. Springer, 3–19. https://doi.org/10.1007/978-3-319-21668-3_1