Effect Summaries for Thread-Modular Analysis *

Sound Analysis despite an Unsound Heuristic

Lukáš Holík[1], Roland Meyer[2,**], Tomáš Vojnar[1], and Sebastian Wolff[2,3]

[1] FIT BUT, IT4Innovations Centre of Excellence
[2] TU Braunschweig
[3] Fraunhofer ITWM

Abstract We propose a novel guess-and-check principle to increase the efficiency of thread-modular verification of lock-free data structures. We build on a heuristic that guesses candidates for stateless effect summaries of programs by searching the code for instances of a copy-and-check programming idiom common in lock-free data structures. These candidate summaries are used to compute the interference among threads in linear time. Since a candidate summary need not be a sound effect summary, we show how to fully automatically check whether the precision of candidate summaries is sufficient. We can thus perform sound verification despite relying on an unsound heuristic. We have implemented our approach and found it up to two orders of magnitude faster than existing ones.

1 Introduction

Verification of concurrent, lock-free data structures has recently received considerable attention [2,3,14,28,29]. Such structures are both of high practical relevance and, at the same time, difficult to write. A common correctness notion in this context is *linearizability* [15], which requires that every concurrent execution can be linearized to an execution that could also occur sequentially. For many data structures, linearizability reduces to checking control-flow reachability in a variant of the data structure that is augmented with observer automata [2]. This control-flow reachability problem, in turn, is often solved by means of *thread-modular analysis* [4,19]. Our contribution is on improving thread-modular analyses for verifying linearizability of lock-free data structures.

Thread-modular analyses compute the least solution to a recursive equation

$$X \;=\; X \cup \mathit{seq}(X) \cup \mathit{interfere}(X)\,.$$

The domain of X are sets of *views*, partial configurations reflecting the perception of a single thread about the shared heap. Crucially, thread-modular analyses abstract away from the correlation among the views of different threads. Function $\mathit{seq}(X)$ computes a sequential step, the views obtained from X by letting each thread execute a command

* This work was supported by the Czech Science Foundation project 16-24707Y, the BUT FIT project FIT-S-17-4014, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), and by the German Science Foundation (DFG) project R2M2.
The full version is available as technical report [16].
** A part of the work was done when the author was at Aalto University.

on its own views. This function, however, does not reflect the fact that a thread may change a part of the shared heap seen by others. Such interference steps are computed by $interfere(X)$. It is this function that we improve on. Before turning to the contribution, we recall the existing approaches and motivate the need for more work.

In the *merge-and-project* approach to interference (e.g., [4,11,19,22]), a merge operation is applied on every two views in X to determine all merged views consistent with the given ones. On each of the consistent views, one thread performs a sequential step, and the result is projected to what is seen by the other thread. The approach has problems with efficiency. The number of merge operations is exactly the square of the number of views in the fixed point. In addition, every merge of two views is expensive. It has to consider all consistent views whose number can be exponential in the size of the views.

The *learning* approach to interference [34,24] derives, via symbolic execution, a symbolic update pattern for the shared heap. The learning process is integrated into the fixed-point computation, which incurs an overhead. Moreover, the number of update patterns to be learned is bounded only by the number of reachable views. An interference step applies the learned update patterns to all views, which again is quadratic in the number of views. Moreover, although update patterns abstract away from thread-local information, computing each application still requires a potentially expensive matching. There are, however, fragments of separation logic with efficient entailment [7].

What is missing is an *efficient* approach to computing interferences among threads.

Main ideas of the contribution. We propose to compute $interfere(X)$ by means of so-called *effect summaries*. An effect summary for a method M is a *stateless* program Q_M which over-approximates the effects that M has on the shared heap. With such summaries at hand, the interference step can be computed in linear time by executing the method summaries Q_M for all methods M on the views in the current set X. This is a substantial improvement in efficiency over merge-and-project and learning techniques, which require time roughly quadratic in the size of the fixed-point approximant, X, and possibly exponential in the size of views.

Technically, *statelessness* is defined as atomicity and absence of persistent local state. We found both requirements typically satisfied by methods of lock-free data structures. For our approach, this means stateless summaries are likely to exist (which is confirmed by our experiments). The reason why the atomicity requirement holds is that the methods have to preserve the integrity of the data structure under interleavings. The absence of persistent state holds since interference by other threads may invalidate local state at any time.

We propose a heuristic to compute, from a method M, a stateless program Q_M which is a candidate for being an effect summary of M. Whether or not this candidate is indeed a summary of M is checked on top of the actual analysis, as discussed below. Our heuristic is based on looking for occurrences of a programming idiom common in lock-free data structures which we call *copy-and-check blocks*. Such a block is a piece of code that, despite lock-free execution, appears to be executed atomically. Roughly, we identify each such block and turn it into an atomic program.

Programmers achieve the above mentioned atomicity of copy-and-check blocks by first creating a local copy of a shared variable, performing some computation over it,

checking whether the copy is still up-to-date and, if so, publishing the results of the computation to the shared heap. A classic implementation of such blocks is based on *compare-and-swap* (CAS) instructions. In this case, for a local variable `t` and a shared variable `T`, the copy-and-check block typically starts with an assignment `t=T` and finishes with executing `CAS(T,t,x)` which atomically checks whether `t==T` holds and, if so, changes the value of `T` to `x`. Hereafter, we will denote such blocks as *CAS blocks*, and we will concentrate on them since they are rather common in practice [31,23,8]. However, we note that the same principle can be used to handle other kinds of copy-and-check blocks, e.g., those based on the *load-link/store-conditional* (LL/SC) mechanism.

The idea of program analyses to employ the intended behavior of CAS blocks by treating them as atomic is quite natural. The reason why it is not common practice is that this approach is not sound in general. The atomicity may be introduced too coarsely, and, as a result, an $interfere(X)$ implementation based on the guessed candidate summaries may miss interleavings present in the actual program. For our analysis, this means that its soundness is conditional upon the fact that the candidate summaries used are indeed proper effect summaries. It must be checked that they are stateless and that they cover all effects on the shared heap. We propose a fully automatic and efficient way of performing those checks. To the best of our knowledge, we are the first to propose such checks.

To check whether candidate summaries indeed cover the effects of the methods for which they were constructed, the idea is to let the methods execute under any number of interferences with the candidate summaries and see if some effect not covered by the candidate summaries can be obtained. Formally, we use the program $Q = \bigoplus_i Q_{M_i}$, which executes a non-deterministically chosen candidate summary Q_{M_i} of a method M_i, execute the Kleene iteration Q^* in parallel with each method M, and check whether the following inclusions holds:

$$\mathit{Effects}(M \parallel Q^*) \subseteq \mathit{Effects}(Q^*).$$

If this inclusion holds, Q^* covers the actual interference all methods may cause. Hence, our novel implementation of $interfere(X)$ explores all possible interleavings. The cost of the inclusion test is asymptotically covered by that of computing the fixed point, and practically negligible. It can be checked in linear time (in the size of the fixed point) by performing, for every view in X, a sequential step and testing whether the effect of the step can be mimicked by the candidate summaries. It is worth pointing out the cyclic nature of our reasoning: we use the candidate summaries to prove their own correctness.

Statelessness is an important aspect in the above process. It guarantees that the sequential iteration of Q^* explores the overall interference the methods of the data structure cause. As we are interested in parametric verification, the overall interference is, in fact, the one produced by an unbounded number of concurrent method invocations. Hence, computing this interference using candidate summaries requires us to analyse the program $\prod^\infty Q$, which is a parallel composition of arbitrarily many Q instances. However, statelessness guarantees that each of these instances executes atomically without retaining any local state. While the atomicity ensures that the concurrent Q instances cannot overlap, the absence of local state ensures that Q instances cannot influence each other, even if executed consecutively by the same thread. Hence, we can use a single thread executing the iteration Q^* in order to explore the interference caused by $\prod^\infty Q$. This justifies the usage of Q^* for the effect coverage above. The check for

statelessness is similar to the one of effect coverage. If both tests succeed, the analysis information is guaranteed to be sound.

Overview of the approach, its advantages, and experimental evaluation. Overall, our thread-modular analysis proceeds as follows. We employ the CAS block heuristic to compute candidate summaries. We use these candidates to determine the interferences in the fixed-point computation. Once the fixed point has been obtained, we check whether the candidates are valid summaries. If so, the fixed point contains sound information, and can be used for verification (or, an on-the-fly computed verification result can be used). Otherwise, verification fails. Currently, we do not have a refinement loop because it was not needed in our experiments.

Our method overcomes the limitations of the previous approaches as follows. The summary program, Q, is quadratic in the syntactic size of the program—not in the size of the fixed point. The interference step executes the summary on all views in the current set X, which means an effort linear rather than quadratic in the fixed-point approximant. Moreover, Q is often acyclic and hence needs linear time to execute, as opposed to the worst case exponential merge or match. In our benchmarks, we needed at most 5 very short summaries, usually around 3–5 lines of code each. The computation of candidate summaries (based on cheap and standard static analyses) and their check for validity are separated from the fixed point, and the cost of both operations is negligible. We stress that our fixed point as well as the validity check do not rely on the actual algorithm used to compute the summary.

We implemented our thread-modular analysis with effect summaries on top of our state-of-the-art tool [14] based on thread-modular reasoning with merge-and-project. We applied the implementation to verify linearizability in a number of concurrent list implementations. Compared to [14], we obtain a speed-up of two orders of magnitude. Moreover, we managed to infer stateless effect summaries for all our case studies except the DGLM queue [8] under explicit memory management (where one needs to go beyond statelessness). However, we are not aware of any automatic approach that would be able to verify linearizability of this algorithm.

2 Effect Summaries on an Example

The main complication for writing lock-free algorithms is to guarantee robustness under interleavings. The key idea to tackle this issue is to use a specific update pattern, namely the CAS-blocks discussed in Section 1. We now show how CAS blocks are employed in Treiber's lock-free stack implementation under garbage collection, the code of which is given in Listing 1. The `push` method implements a CAS block by: (1) copying the top of stack pointer, `top=ToS`, (2) linking the `node` to be inserted to the current top of stack, `node.next=top`, and (3) making `node` the new top of stack in case no other thread changed the shared state, `CAS(ToS,top,node)`. Similarly, `pop` proceeds by: (1) copying the top of stack pointer, `top=ToS`, (2) querying its successor, `next=top.next`, and (3) swinging `ToS` to that successor in case the stack did not change, `CAS(ToS,top,next)`.

Following the CAS-block idiom, the only statements modifying the shared heap in Treiber's stack are the CAS operations. Hence, we identify three types of effects on the shared heap. First, a successful CAS in `push` makes `ToS` point to a newly allocated

```
struct Node { data_t data; Node next; }
shared Node ToS;

void push(data_t in) {
  Node node = new Node(in);
  while (true) {
    Node top = ToS;
    node.next = top;
    if(CAS(ToS, top, node)){
      return;
} } }

S1: atomic {
  /* push */
  Node node = new Node(*);
  node.next = ToS;
  ToS = node;
}
```

```
bool pop(data_t& out) {
  while (true) {
    Node top = ToS;
    if(top == NULL){
      return false;
    }
    Node next = top.next;
    if(CAS(ToS, top, next)){
      out = top.data;
      return true;
} } }

S2: atomic {
  /* pop */
  assume(ToS != NULL);
  ToS = ToS.next;
}
S3: atomic { /* skip */ }
```

Listing 1. Pseudo code of the Treiber's lock-free stack [31] and its effect summaries.

cell that, in turn, points to the previous value of `ToS`. Second, a successful CAS in `pop` moves `ToS` to its successor `ToS.next`. Since we assume garbage collection, the removed element is not freed but remains in the shared heap until collected. Third, the effect of any other statement on the shared heap is the identity.

With the effects of Treiber's stack identified, we can turn towards finding an approximation. For that, consider the program fragments from Listing 1: `S1` covers the effects of the CAS in `push`, `S2` covers the effects of the CAS in `pop`, and, lastly, `S3` produces the identity-effect covering all remaining statements. Then, the summary program is $Q = \texttt{S1} \oplus \texttt{S2} \oplus \texttt{S3}$.

To obtain the non-trivial summaries `S1` and `S2`, it suffices to concentrate on the block of code between the `top=ToS` assignment and the subsequent `CAS(ToS,top,_)` statement. Without going into details (which will be provided in Section 5), the summaries result from considering the code between the two statements atomic, performing simplification of the code under this atomicity assumption, and including some purely local initialization and finalization code (such as the allocation in the `push` method).

3 Programming Model

A concurrent program P is a parallel composition of threads T. The threads are while-programs formed using sequential composition, non-deterministic choice, loops, atomic blocks, skip, and primitive commands. The syntax is as follows:

$$P ::= T \mid P \parallel P \qquad T ::= T_1; T_2 \mid T_1 \oplus T_2 \mid T^* \mid \texttt{atomic}\, T \mid \texttt{skip} \mid C\,.$$

We use Thrd for the set of all threads. We also write P^* to mean a program P with the Kleene star applied to all threads. The syntax and semantics of the commands in C are orthogonal to our development. We comment on the assumptions we need in a moment.

We assume programs whose threads implement methods from the interface of the lock-free data structure which is to be verified. The fact that, at runtime, we may find an arbitrary (finite) number of instances of each of the threads corresponds to an arbitrary number of concurrent method invocations. The verification task is then formulated as proving a designated shared heap unreachable in all instantiations of the program. Since thread-modular analyses simultaneously reason over all instantiations of the program, we

refrain from making this parameterization more explicit. Instead, we consider program instances simply as programs with more copies of the same threads.

We model heaps as partial and finite functions $h\colon \mathit{Var} \cup \mathbb{N} \rightharpoonup \mathbb{N}$. Hence, we do not distinguish between the stack and the heap, and let the heap provide valuations for both the program variables from Var and the memory cells from $\mathbb{N}$. We use $\mathbb{H}$ for the set of all heaps. Initially, the heap is empty, denoted by emp with $\mathit{dom}(\mathit{emp}) = \varnothing$. We write $\bot$ if a partial function is undefined for an argument: $h(e) = \bot$ if $e \notin \mathit{dom}(h)$.

We assume each thread has an identifier from $\mathit{Tid} \subseteq \mathbb{N}$. A *program state* is a pair $(s,\ \mathit{cf})$ where $s \in \mathbb{H}$ is the shared heap and $\mathit{cf} : \mathit{Tid} \to \mathit{Thrd} \times \mathbb{H}$ maps the thread identifiers to *thread configurations*. A thread configuration is of the form $(T,\ o)$ with $T \in \mathit{Thrd}$ and $o \in \mathbb{H}$ being a heap owned by T. If $\mathit{cf} = \{i \to (T,\ o)\}$ contains a single mapping, we write simply $(s,\ (T,\ o))$.

Our development crucially relies on having a notion of separation between the shared heap s and the owned heap o of a thread T. However, the actual definitions of what is owned and what shared are a parameter to our development. We just require the separation to respect disjointness of the shared and owned heaps and to be defined such that it is preserved across execution of program statements. The latter is formalized below in Assumption 1. To render disjointness formally, we say that a state $(s,\ \mathit{cf})$ is *separated*, denoted by $\mathit{separate}(s,\ \mathit{cf})$, if, for every $i_1, i_2 \in \mathit{dom}(\mathit{cf})$ with $\mathit{cf}(i_j) = (T_{i_j},\ o_{i_j})$ and $i_1 \neq i_2$, we have $\mathit{dom}(s) \cap \mathit{dom}(o_{i_j}) \cap \mathbb{N} = \varnothing$ and $\mathit{dom}(o_{i_1}) \cap \mathit{dom}(o_{i_2}) \cap \mathbb{N} = \varnothing$. Note that, in order to allow for thread-local variables, the heaps need to be disjoint only on memory cells (but not on variables), thus the additional intersection with $\mathbb{N}$.

We use $\to$ to denote *program steps*. The sequential semantics of threads is as expected for sequential composition, choice, loops, and skip. An atomic block `atomic` T summarizes a computation of the underlying thread T into a single program step. The semantics of primitive commands depends on the actual set C. We do not make it precise but require it to preserve separation in the following sense.

Assumption 1. *For every step* $(s,\ (T,\ o)) \to (s',\ (T',\ o'))$ *with* $\mathit{separate}(s,\ (T,\ o))$, *we have* $\mathit{separate}(s',\ (T',\ o'))$.

The semantics of a concurrent program incorporates the requirement for separation into its transition rule. A thread may only update the shared heap and those parts of the heap it owns. No other parts can be modified. Therefore, we let threads execute in isolation and ensure that the combined resulting state is separated:

$$\frac{(s,\ \mathit{cf}(i)) \to (s',\ \mathit{cf}'') \qquad \mathit{cf}' = \mathit{cf}[i \to \mathit{cf}''] \qquad \mathit{separate}(s',\ \mathit{cf}')}{(s,\ \mathit{cf}) \to (s',\ \mathit{cf}')}\ (\textsc{par})$$

Although a precise notion of separation is not needed for the development of our approach in Section 4, we give, for illustration, the notion we use in our implementation and experiments. In the case of garbage collection (like in Java), the owned heap of a thread includes, as usual, its local variables and cells accessible from these variables, which were allocated by the thread but never made accessible through the shared variables. The shared heap then contains the shared variables, all cells that were once made accessible from them, as well as cells waiting for garbage collection. For the case of explicit memory management, we need a more complicated mechanism of ownership

transfer where a shared cell can become owned again. We propose such a mechanism in Section 6.

We assume the computation of the program under scrutiny to start from an initial state $init_P = (\mathrm{s}_{init},\ cf_{init,P})$ where s_{init} is the result of an initialization procedure. The initial thread configurations, denoted by $cf_{init,T}$, are of the form $(T,\ emp)$. The initialization procedure is assumed to be part of the input program. We are interested in the shared heaps reachable by program P from its initial state:

$$SH(P) := \{s \mid \exists\ cf.\ init_P \rightarrow^* (s,\ cf)\}.$$

In what follows, we assume that the correctness of a program P can be read of its reachable shared heaps, $SH(P)$. For this, some instrumentation of P might be needed. Such instrumentations are possible for a variety of properties. In particular, the instrumentation with observer automata from Section 1 allows one to check for linearizability.

4 Interference via Summaries

We now present our new approach to computing the effect of thread interference steps on the shared heap (corresponding to evaluating the expression $interfere(X)$ from Section 1 for a set of views X) in a way which is suitable for concurrency libraries. In particular, we introduce a notion of a *stateless effect summary* Q: a program whose repeated execution is able to produce all the effects on the shared heap that the program under scrutiny, P, can produce. With a stateless effect summary Q at hand, one can compute $interfere(X)$ by repeatedly applying Q on the views in X until a fixed point is reached. Here, statelessness assures that Q is applicable repeatedly without any need to track its local state.

Later, in Section 5, we provide a heuristic for deriving *candidates* for stateless effect summaries. Though our experiments show that the heuristic we propose is very effective in practice, the candidate summary that it produces is not guaranteed to be an effect summary, i.e., it is not guaranteed to produce all the effects on the shared heap that P can produce. A candidate summary which is not an effect summary is called *unsound*. To guarantee soundness of our approach even when the obtained candidate summary is unsound, we provide a test of soundness of candidate summaries. Interestingly, as we prove, it is the case that even (potentially) unsound candidate summaries can be used to check their own soundness—although this step appears to be cyclic reasoning.

4.1 Stateless Effect Summaries

We start by formalizing the notion of statelessness. Intuitively, a thread is stateless if it terminates after a single step and disposes its local heap. Formally, we say that a thread T of a program Q is *stateless* if, for all reachable shared heaps $s \in SH(Q^*)$ and all transitions $(s,\ cf_{init,T}) \rightarrow (s',\ cf)$, we have $cf = (\texttt{skip},\ emp)$. A program Q is stateless if so are all its threads. Note that statelessness should hold from all reachable shared heaps rather than from just all heaps. While an atomic execution to `skip` would be easy to achieve from all heaps, a clean-up yielding emp can only be achieved if we have control over the thread-local heap. Also note that statelessness basically requires a thread to consist of a top-level atomic block to ensure termination in a single step.

For an example, consider the summary `S1` of `push` in Treiber's stack from Listing 1. It is stateless because (1) the top-level `atomic` block ensures execution in a single step, and (2) the allocated node is published, i.e., moved from the owned heap to the shared heap.

Next, we define the *effects* of a program P, denoted by $EF(P) \subseteq \mathbb{H} \times \mathbb{H}$, to be the set $EF(P) = \{(s,\, s') \mid init_P \rightarrow^* (s,\, cf) \rightarrow (s',\, cf')\}$. This set generalizes the reachable shared heaps, $SH(P)$: it contains all atomic (single-step) updates P performs on the heaps from $SH(P)$.

In Treiber's stack, as discussed in Section 2, the updates performed by the CAS statements are effects. The remaining statements also yield effects. However, since they do not modify the shared heap, they produce the identity effect.

Altogether, a program Q is a *(stateless) effect summary* of P if it is stateless and $EF(T \parallel Q^*) \subseteq EF(Q^*)$ holds for all threads $T \in P$. We refer to this inclusion as the *effect inclusion*. Intuitively, it states that Q^* subsumes all the effects T may have under interference with Q^*. The lemma below shows that the effect inclusion can be used to check whether a candidate summary is indeed an effect summary. Moreover, the check can deal with the different threads separately.

Lemma 1. *If Q is stateless and $EF(T \parallel Q^*) \subseteq EF(Q^*)$ holds for all $T \in P$, then we have $EF(P) \subseteq EF(P \parallel Q^*) \subseteq EF(Q^*)$.*

In what follows, we describe our novel thread-modular analysis based on effect summaries. We assume that, in addition to the program P under scrutiny, we have a program Q which is a candidate for being a summary of P (obtained, e.g., by the heuristic that we provide in Section 5). In Section 4.2, we first provide a fixed-point computation where the interference step is implemented by a repeated application of the candidate summary Q. We show that if the candidate summary Q is an effect summary, then the fixed point we compute is a conservative over-approximation of the reachable shared heaps of P. Next, in Section 4.3, we show that the fact whether or not Q is indeed an effect summary of P can be checked efficiently on top of the computed fixed point (even though the fixed point need not over-approximate the reachable shared heaps of P).

In the case that the test of Section 4.3 fails, Q is not an effect summary of P, and our verification fails with no definite answer. As future work, one could think of proposing ways of patching the summaries based on feedback from the failed test. Then, along the lines of [5,6], the previously computed, unsound state space can be reused: one applies the newly added summaries to the already explored states and continues with the analysis afterwards. However, in our experiments, using the heuristic computation of candidate summaries proposed in Section 5, this situation has not happened for any program where a stateless effect summary exists. In the only experiment where our approach failed (the DGLM queue under explicit memory management, which has not been verified by any other fully automatic tool), the notion of stateless effect summaries itself is not strong enough. Hence, a perhaps more interesting question for future work is how to further generalize the notion of effect summaries.

4.2 Summaries in the Fixed-Point Computation

To explore the reachable shared heaps of a program P, we suggest a thread-modular analysis which explores the reachable states of the threads $T \in P$ in isolation. To account

for the possible thread interleavings of the original program, we apply interference steps to the threads T by executing the provided summary Q. Conceptually, this process corresponds to exploring the state space of the two-thread programs $T \parallel Q^*$ for all syntactically different threads $T \in P$. Technically, we collect the reachable states of those programs in the following least fixed point:

$$\begin{aligned} X_0 &= \{(\mathrm{s}_{init},\, (T,\, emp)) \mid T \in P\} \\ X_{i+1} &= X_i \,\cup\, seq(X_i) \,\cup\, interfere(X_i)\ . \end{aligned}$$

Since Q^* has no internal state, the analysis only keeps the thread-local configurations of the threads T. Functions $seq(\cdot)$ and $interfere(\cdot)$ compute sequential steps (steps of T) and interference steps (steps of Q^*), respectively, as follows:

$$\begin{aligned} seq(X_i) &= \{(s',\, cf') \mid \exists\, (s,\, cf) \in X_i.\ (s,\, cf) \rightarrow (s',\, cf')\} \\ interfere(X_i) &= \{(s',\, cf\ ) \mid separate(s',\, cf) \,\wedge\, \exists\, s, cf'. \\ &\qquad\qquad (s,\, cf) \in X_i \,\wedge\, (s,\, cf_{\,init,Q}) \rightarrow (s',\, cf')\}\ . \end{aligned}$$

Function $seq(X_i)$ is standard. For $interfere(X_i)$ we apply Q to each configuration $(s,\, cf) \in X_i$ by letting it start from the shared heap s and its initial thread-local configuration $cf_{\,init,Q}$. Then we extract the updated shared heap, s', resulting in the post configuration $(s',\, cf)$. Altogether, this procedure applies to the views in X_i the shared heap updates dictated by Q. The thread-local configurations, cf, of threads T are not changed by interference. This locality follows from the separation.

The following lemma states that the set of shared heaps collected from the above fixed point is indeed the set of reachable shared heaps of all $T \parallel Q^*$. Let X_k be the fixed point and define $\mathcal{R} = \{s \mid \exists cf.\, (s,\, cf) \in X_k\}$.

Lemma 2. *If Q is a summary of P, then $\mathcal{R} = \bigcup_{T \in P} SH(T \parallel Q^*)$.*

With the state space exploration in place, we can turn towards a soundness result of our method: given an appropriate summary Q, the fixed-point computation over-approximates the reachable shared heaps of P.

Theorem 1. *If Q is a summary of P, then we have $SH(P) \subseteq SH(Q^*) = \mathcal{R}$.*

The rationale behind the theorem is as follows. Relying on Q being a summary of P provides the effect inclusion. So, Lemma 1 yields $EF(P \parallel Q^*) \subseteq EF(Q^*)$. From the definition of effects we can then conclude $SH(P \parallel Q^*) \subseteq SH(Q^*)$. Thus, we have $SH(P) \subseteq SH(Q^*)$ because $SH(P) \subseteq SH(P \parallel Q^*)$ is always true. This shows the first inclusion. Similarly, the effect inclusion gives $SH(T \parallel Q^*) \subseteq SH(Q^*)$ by the definition of reachability. Hence, we conclude using Lemma 2.

4.3 Soundness of Summarization

Soundness of our method, as stated by Theorem 1 above, is conditioned by Q being a summary of P. In our framework, Q is heuristically constructed and there is no guarantee that it really summarizes P. Hence, for our analysis to be sound, we have to check summarization; we have to establish (1) the effect inclusion, and (2) statelessness of Q.

To that end, we check that (1) every update T performs on the shared heap in the system $T \parallel Q^*$ can be mimicked by Q, and that (2) every execution of Q terminates in a single step and does not retain persistent local state. We implement those checks on top of the fixed point, X_k, as follows:

$$\forall\, (s,\, cf) \in X_k\, \forall\, s',\, cf',\, i\, \exists cf''.$$
$$(s,\ cf) \to (s',\ cf') \implies (s,\ cf_{\,init,Q}) \to (s',\ cf'') \wedge \qquad \text{(CHK-MIMIC)}$$
$$(s,\ cf_{\,init,Q}(i)) \to (s',\ cf') \implies cf' = (\texttt{skip},\ emp) \qquad \text{(CHK-STATELESS)}$$

The above properties indeed capture our intuition. The former, (CHK-MIMIC), states that, for every explored T-step of the form $(s,\ cf) \to (s',\ cf')$, the effect $(s,\ s')$ is also an effect of Q. That is, executing Q starting from s yields s'. This establishes the effect inclusion as required by Lemma 1. The latter check, (CHK-STATELESS), states that every thread of Q must terminate in a single step and dispose its owned heap. This constraint is relaxed to those shared heaps which have been explored during the fixed-point computation. That is, it ensures statelessness of Q on all heaps from $\mathcal{R}$. The key aspect is to guarantee that $\mathcal{R}$ includes $SH(Q^*)$ as required by the definition. We show that this inclusion follows from the check.

The above checks rely on the fixed point, which, in turn, is computed using the candidate summary Q. That is, we use Q to prove its own correctness. Nevertheless, our development results in a sound analysis as stated by the following theorem.

Theorem 2. *The fixed point X_k satisfies* (CHK-MIMIC) *and* (CHK-STATELESS) *if and only if Q is a summary of P.*

5 Computing Effect Summaries

We now provide our heuristic for computing effect summaries. It is based on CAS blocks between an assignment `t=T`, denoted as *checked assignment*, and a CAS statement `CAS(T,t,x)`, denoted as *checking CAS* below. Since we compute a summary for each such block, the number of summaries is at most quadratic in the size of the input.

In what follows, consider some method M given by its control-flow graph (CFG) $G = (V, E, v_{init}, v_{final})$. The CFG has a unique initial and a unique final state, which we will use in our construction. Return commands are assumed to lead to the final state. As we are only interested in the effect on the shared heap, we drop return values from return commands. Likewise, we skip assignments to output parameters unless they are important for the flow of control in M. We assume the summaries to execute with non-deterministic input values, and so we replace every input parameter with a symbolic value $*$. Conditionals, loops, and CAS commands are represented by two edges, for the successful and failing execution, respectively. Let $e_{asgn} := (v_{asgns}, \texttt{t=T}, v_{asgnt})$ be the CFG edge of the checked assignment, and let the successful branch of the checking CAS be $e_{cas} := (v_{cas}, \texttt{CAS(T,t,x)}, v_{cassuc})$. Next, let $e_{asgn'} := (v_{asgns}, \texttt{t=T}, v_{asgnt'})$ be a copy of the checked assignment to be used as the beginning of the CAS block, and let $e_{cas'} := (v_{cas}, \texttt{CAS(T,t,x)}, v_{cassuc'})$ be a copy of the checking CAS to be used as the end of the CAS block. Here, $v_{asgnt'}$ and $v_{cassuc'}$ are fresh nodes.

To give a concise description of effect summaries, the following shortcuts will be helpful. We write $rand(G)$ for the CFG obtained from G by replacing each occurrence of a shared variable by a non-deterministic value $*$. By $G - S$, we mean the CFG obtained from G by dropping all edges carrying commands from the set S. Given nodes v_1 and v_2, we denote by $G(v_1, v_2)$ the CFG obtained from G by making v_1/v_2 the initial/final node, respectively. Given two CFGs G and G', we define $G; G'$ as their disjoint union where the single final state of G is merged with the single initial state of G'. Finally, we allow compositions $e; G$ and $G; e$ of a CFG G with a single edge e, by viewing e as a CFG consisting of a single edge with the initial/final nodes being the initial/final nodes of e, respectively.

The construction of the summary proceeds in two steps. First we identify the CAS block and create the control-flow structure, then we clean it up using data flow analysis and generate the final code of the summary. Note that the clean-up step is optional but generates a concise form beneficial for verification.

Step 1: Control-flow structure. A summary consists of an initialization phase, followed by the CAS block, and a finalization phase. The first step results in the CFG

$$G_{init}; G_{block}; G_{final} \ .$$

The guiding theme of the construction is to preserve all sequences of commands that may lead through the CAS block.

In the initialization phase, which is intended for purely local initialization, the method is assumed to be interrupted by other threads in the sense that the values of shared pointers may spontaneously change. Therefore, we replace all dependencies on shared variables by non-deterministic assignments. Moreover, all return commands are removed since we have not yet passed the CAS block. Eventually, when arriving at the v_{asgns} location, the summary non-deterministically guesses that the CAS block should begin, and so the control is transferred to it via the $e_{asgn'}$ edge. Hence, the initialization is:

$$G_{init} \ := \ (rand(G) - \{\texttt{return}\})(v_{init}, v_{asgns}) \ .$$

The CAS block begins with the $e_{asgn'}$ edge, i.e., with the checked assignment, and ends with the $e_{cas'}$ edge, i.e., the checking CAS statement. From the CAS block, we remove all control-flow edges with assignments `t=T` as we fixed the checked assignment when entering the CAS block (other assignments of the form `t=T`, if present, will give rise to other CAS blocks; and a repeated execution of the same checked assignment then corresponds to a repeated execution of the summary). We also remove the return commands as the finalization potentially still has to free owned heap. Failing executions of the checking CAS do not leave the CAS block (and typically get stuck due to the removed checked assignments). Successful executions may leave the CAS block, but do not have to. Eventually, the summary guesses the last successful execution of the checking CAS and enters the finalization phase. Hence, we get the following code:

$$G_{block} \ := \ e_{asgn'}; ((G - \{\texttt{return}, \texttt{t=T}\})(v_{asgnt}, v_{cas})); e_{cas'} \ .$$

Sometimes, the checked assignment can use local variables assigned prior to the checked assignment. In such a case, we add edges with these assignments before the

```
// Initialization
while (true) {
  if (*) goto L1;
  Node top=*;
  if (top==NULL){
    STOP;
  }
  Node next=top.next;
  if(CAS(*,top,next)){
    out=top.data;
    STOP;
  }
}
STOP;
```

```
// CAS block
L1:Node top=ToS;
goto L2;
while (true) {
  STOP;
  L2:if(top==NULL){
    STOP;
  }
  Node next=top.next;
  if(*) goto L3;
  if(CAS(ToS,top,next)){
    out=top.data;
    STOP;
  }
}
STOP;
L3:if(CAS(ToS,top,next))
   goto L4;
```

```
// Finalization
while (true) {
  Node top=*;
  if(top==NULL){
    return;
  }
  Node next=top.next;
  if(CAS(*,top,next)){
    L4:out=top.data;
    return;
  }
}
```

Figure 1. Step 1 in the summary computation for the `pop` method in Treiber's stack.

$e_{asgn'}$ edge. This happens, e.g., in the `enqueue` procedure of Michael&Scott's lock-free queue where the sequence `tail=Tail;next=tail.next` is used. If the checked assignment is `next=tail.next`, we start G_{block} with edges containing `tail=Tail` and `next=tail.next`.

The finalization phase, again, cannot rely on shared variables. However, here, we preserve the return statements to terminate the execution:

$$G_{final} \;:=\; rand(G)(v_{cassuc}, v_{final})\;.$$

Figure 1 illustrates the construction on the `pop` method in Treiber's stack. Instead of a CFG, we give the source code. `STOP` represents deleted edges and the fact that we cannot move from one phase to another not using the new edges.

Step 2: Cleaning-up and summary generation. We perform *copy propagation* using a must analysis that propagates an assignment `y=x` to subsequent assignments `z=y`, resulting in `z=x`. That it is a must analysis means the propagation is done only if `z=y` definitely has to use the value of `y` that stems from the assignment `y=x`. Moreover, we perform the copy propagation assuming that the entire summary executes atomically. For the initialization phase, the result is that the non-deterministic values for shared variables propagate through the code. Similarly, for the CAS block, the shared variables themselves propagate through the code. For the finalization phase, non-deterministic values propagate only in the case when a local variable does not receive its value from the CAS block. As a result, after the copy propagation, the CAS and the finalization block may contain conditionals that are constantly true or constantly false. We replace those that evaluate to true by skip and remove the edges that evaluate to false. The result of the copy propagation is illustrated in Figure 2.

Subsequently, we perform a *live variables analysis*. A variable is live if it may occur in a subsequent conditional or on the right-hand side of a subsequent assignment. Otherwise, it is dead. We remove all assignments to dead variables including output parameters. In our running example, all assignments to local variables as well as to the output parameter can be removed.

```
// Initialization
while (true) {
  if (*) goto L1;
  Node top=*;
  if(*){
    STOP;
  }
  Node next=*;
  if(*){
    out=*;
    STOP;
  }
}
STOP;
```

```
// CAS block
L1:Node top=ToS;
   goto L2;
while (true) {
  STOP;
  L2:if(ToS==NULL){
    STOP;
  }
  Node next=ToS.next;
  if(*) goto L3;
  ToS=ToS.next;
  out=ToS.data;
  STOP;
}
STOP;
L3:if(CAS(ToS,ToS,ToS.next))
   goto L4;
```

```
// Finalization
while (true) {
  Node top=*;
  if(*){
    return;
  }
  Node next=*;
  if(*){
    L4:out=ToS.data;
    return;
  }
}
```

Figure 2. Copy propagation within the summary computation for `pop` in Treiber's stack.

Next, we remove code that is *unreachable*, *dead*, or *useless*. Unreachable code can appear due to the modifications of the CFG. Dead code does not lead to the final location. Useless code does not have any impact on the values of the variables used, which can concern even (possibly infinite) useless loops.

Finally, the resulting code is wrapped into an `atomic` block, and conditionals are replaced by `assume` statements. For the `pop` method in Treiber's stack, we get the summary `S2` given in Listing 1.

6 Generalization to Explicit Memory Management

We now generalize our approach to explicit memory management. The problem is that the separation between the shared and owned heap is difficult to define and establish in this case. Ownership as understood in garbage collection, where no other thread can access a cell that was allocated by a thread but not made shared, does not exist any more. Memory can be freed and *reallocated*, with other threads still holding (dangling) pointers to it. These threads can read and modify that memory, hence the allocating thread does not have strong guarantees of exclusivity. However, programmers usually try to prevent effects of accidental reallocations: threads are designed to *respect ownership*. That is, a thread should be allowed to execute *as if* it had exclusive access to the memory it owns.

Our development is parameterized by a notion of separation between the shared and owned heap. To generalize the results, we provide a new notion of ownership suitable for explicit memory management. However, the new notion is not guaranteed to be preserved by the semantics. Instead, we include into our fixed-point computation a check that the program respects this ownership, and give up the analysis if the check fails.

To understand how the heap separation is influenced by basic pointer manipulations, we consider the following set of commands C:

$$x = \texttt{malloc},\ x = \texttt{free},\ x = y,\ x = y.sel_i,\ x.sel_i = y\ .$$

Here, x, y are program variables and $sel_0, \ldots, sel_n$ are selectors, from which the first, say m, are *pointer selectors* and the rest are *data selectors*. Command $x = \texttt{malloc}$ allocates a *record*, a free block of addresses $a + 0, \ldots, a + n$, and sets $h(x)$ to a. Command $x = \texttt{free}$ frees the record $h(x) + 0, \ldots, h(x) + n$. Selectors correspond to field accesses: $x.sel_i$ refers to the content of $a + i$ if x points to a. The remaining commands have the expected meaning.

6.1 Heap Separation

We work with a three-way partitioning of the heap into *shared*, *owned*, and *free* addresses. Free are all addresses that are fresh or have been freed and not reallocated. Shared is every address that is *reachable* from the shared variables and not free. The reachability predicate, however, requires care. First, we must obviously generalize reachability from the first memory cell of a record to the whole record. Second, we must not use *undefined* pointers for reachability. A pointer is undefined if it was propagated from uninitialized or uncontrolled memory. Letting the shared heap propagate through such values would make it possible for the entire allocated heap to be shared (since undefined pointers can have an arbitrary value). Then, owned is the memory which is not shared nor free. The owned memory is partitioned into disjoint blocks that are *owned by individual threads*. A thread gains ownership by moving memory into the owned part, and loses it when the memory is removed from the owned part. The actions by which a thread can gain ownership are (1) allocation and (2) breaking reachability from shared variables by an update of a pointer or a shared variable (*ownership transfer*). An *ownership violation* is then a modification of a thread's owned memory by another thread. This can in particular be (1) freeing or publishing the owned memory or (2) an update of a pointer therein. A program respects ownership if it cannot reach an ownership violation.

Let us discuss these concepts formally. We use $\bot$ to identify free cells. That is, in a heap h address a is free if $h(a) = \bot$ (also written $a \notin dom(h)$). A record is free if so are all its cells. Consequently, the `free` command sets all cells of a record to $\bot$. The shared heap is identified by reachability through defined pointers starting from the shared variables. For undefined pointers we use the symbolic value `udef`. Initially, all variables are undefined. Moreover, we let allocations initialize the selectors of records to `udef`. We use a value distinct from $\bot$ to detect ownership violations by checking whether $\bot$ is reachable from the shared heap (see below). Value `udef` is explicitly allowed to be reachable (this may be needed for list implementations where selectors of sentinel nodes are not initialized). Let $Ptrs$ be the shared pointer variables. Then, the addresses of the shared records in a heap h, denoted by $records(h) \subseteq \mathbb{N} \cup \{\bot\}$, are collected by the following fixed point (where the *address of a record* is its lowest address):

$$S_0 = \{a \mid \exists x \in Ptrs\,.\, h(x) = a \neq \texttt{udef}\}$$
$$S_{i+1} = \{b \mid \exists\, a \in S_i\, \exists\, k.\ a \neq \bot\ \wedge\ 0 \leq k < m\ \wedge\ h(a+k) = b \neq \texttt{udef}\}$$

All addresses within the shared records are shared. The remaining cells, i.e., those that are neither free nor shared, are owned. This definition establishes a sufficient separation for Assumption 1. It is automatically lifted to the concurrent setting by Rule (PAR) following the intuition from above.

It remains to detect ownership violations, which occur whenever a thread modifies cells owned by other threads. Due to the separation integrated into Rule (PAR), threads execute with only the shared heap and their owned heap being visible. The remainder of the heap is cut away. By choice of $\bot$ to identify free cells, the cut away part appears free to the acting thread. In particular, the parts owned by other threads appear free. Hence, in order to avoid ownership violations, a thread must not modify free cells. To that end, an ownership violation occurs if (A) a free cell is freed again, (B) a free cell is written to, or (C) a free cell is published to the shared heap. For (A) and (B) we extend

the semantics of commands to raise an ownership violation if a free cell is manipulated. For (C) we check for every program step whether it results in a shared heap where $\perp$ is made reachable.

Formally, we have the following rules.

$$\frac{\exists\, sel\,.\,(s \uplus o)(x).sel \notin dom(s \uplus o)}{(s,\,(x = \texttt{free},\,o)) \to violation}\ (\text{A}) \qquad \frac{(s \uplus o)(x).sel \notin dom(s \uplus o)}{(s,\,(x.sel = y,\,o)) \to violation}\ (\text{B})$$

$$\frac{(s,\,cf) \to (s',\,cf') \qquad \perp \in records(s')}{(s,\,cf) \to violation}\ (\text{C})$$

Note that reading out free cells is allowed by the above rules. This is necessary because lock-free algorithms typically perform speculating reads and check only later whether the result of the read is safe to use. Moreover, note that our detection of ownership violations can yield false-positives. A cell may not be owned, yet an ownership violation is raised because it appears free to the thread. We argue that such false-positives are *desired* as they access truly free memory. Put differently: an ownership violation detected by the above rules is either indeed an ownership violation or an unsafe access of free memory, that is, a bug.

6.2 Ownership Transfer

The above separation is different from the one used under garbage collection in the earlier sections. When an address becomes unreachable from the shared variables, it is *transferred* into the acting thread's owned heap (although other threads may still have pointers to it). We introduce this ownership transfer to simplify the construction of summaries. The idea is best understood on an example.

Under explicit memory management, threads free cells that they made unreachable from the shared variables to avoid memory leaks. Consider, for example, the method `pop` in Treiber's stack (Listing 1). There, a thread updates the `ToS` variable making the former top of stack, say a, unreachable from the shared heap. In the version for explicit memory management, a is then freed before returning. If ownership was not transferred and address a stayed shared, then two summaries would be needed: one for the update of `ToS` and one for freeing a. However, a stateless version of the latter summary could not learn which address to free since it starts with the empty local heap and with a unreachable from the shared heap. If, on the other hand, ownership of a is transferred to the acting thread, then the former summary can include freeing a (which does not change the shared heap). Moreover, it is even forced to free a in order to remain stateless since a would otherwise persist in its owned heap.

We stress that our framework can be instantiated with other notions of separation, like an analogue of the one for garbage collection or the one of [14], which both do not have ownership transfer. This would complicate the reasoning in Section 4, but could lead to a more robust analysis (ownership transfer is prone to ownership violations).

6.3 ABA Prevention

Additionally, synchronization mechanisms can be incorporated into our approach. For instance, lock-free data structures may use *version counters* to prevent the *ABA problem* [23]: a variable leaves and returns to the same address, and an observer incorrectly

Program		Thread-Modular [14]	Thread Summaries
Coarse stack	GC	0.29s / 343	0.03s / 256
	MM	1.89s / 1287	0.19s / 1470
Coarse queue	GC	0.49 / 343	0.05s / 256
	MM	2.34s / 1059	0.98s / 2843
Treiber's stack [31]	GC	1.99s / 651	0.06s / 458
	MM	25.5s / 3175	1.64s / 2926
Michael&Scott's queue [23]	GC	11.0s / 1530	0.39s / 1552
	MM	11700s / 19742	102s / 27087
DGLM queue [8]	GC	9.56s / 1537	0.37s / 1559
	MM	unsafe (spurious)	violation

Table 1. Experimental results: a speed-up of up to two orders of magnitude.

concludes that the variable has never changed. A well-known scenario of this type causes stack corruption in a naive extension of Treiber's stack to explicit memory management [23]. To give the observer a means of detecting that a variable has been changed, pointers are associated with a counter that increases with every update.

In our analysis, such version counters must be persistent in the shared memory. Since this is an exception from the above definition of separation, a presence of version counters must be indicated by the user (e. g., the user specifies that the version counter of a pointer a is always stored at address $a+1$). The semantics is then adapted in such a way that (1) version counters remain in the shared heap upon freeing, (2) are retained in case of reallocations, and (3) are never transferred to a thread's owned heap. The modifications can be easily implemented, and are detailed in [16]. Last, the thread-modular abstraction has to be adjusted since keeping all counters ever allocated in every thread view is not feasible. One solution is to remember only the values of those counters that are attached to the allocated shared and the thread's own heap.

7 Experiments and Discussion

To substantiate our claim for practical benefits of the proposed method, we implemented the techniques from Sections 4 and 6.[4] Therefore, we modified our previous linearizability checker [14] to perform our novel fixed-point computation. The modifications were straightforward leveraging the existing infrastructure.

Our findings are listed in Table 1. Experiments were conducted on an Intel Xeon E5-2670 running at 2.60GHz. The table includes the running times (averaged over ten runs) and the number of explored views (the size of set X from Section 1). Our benchmarks include well-known data structures such as Treiber's lock-free stack [31], Michael&Scott's lock-free queue [23], and the lock-free DGLM queue [8]. We do not include lock-free set implementations due to limitations of the tool in handling data—not due to limitations of our approach. We ran each benchmark under garbage collection (GC), and explicit memory management (MM) with version counters. Additionally, we include for each benchmark a comparison between our novel fixed point using summaries and the optimized version of the classical thread-modular fixed point from [14].

[4] Available at: `https://github.com/Wolff09/TMRexp/releases/SAS17/`

Our experiments show that summaries provide a significant performance boost compared to classical interference. This holds true for both garbage collection and explicit memory management. For garbage collection, we experience a speed-up of one order of magnitude throughout the entire test suite. Although comparisons among different implementations are inherently unfair, we note that our tool compares favorably to competitors [2,3,33,34]. Under explicit memory management, the same speed-up is present for simple algorithms, like Treiber's stack. For slightly more complex implementations, like Michael&Scott's queue, we observe a more eminent speed-up of over two orders of magnitude. This speed-up is present even though the analysis explores a way larger search space than its classical counterpart. This confirms that our approach of reducing the complexity of interference steps rather than reducing the search space is beneficial for verification.

Unfortunately, we could not establish correctness of the DGLM queue under explicit memory management with neither of the fixed points. For the classical one, the reason was imprecision in the underlying shape analysis which resulted in spurious unsafe memory accesses. For our novel fixed point, the tool detected an ownership violation according to Section 6. While being correct, the DGLM queue indeed features such a violation. The update pattern in the `deque` method can result in freeing nodes that were made unreachable by other threads. The problematic scenario only occurs when the head of the queue overtakes the tail. Despite the similarity, this behavior is not present in Michael&Scott's queue which is why it does not suffer from such a violation.

As hinted in Section 6, one could generalize our theory in such a way that no ownership transfer is required. Without ownership transfer, however, freeing cells becomes an effect of the shared heap which cannot be mimicked: a stateless summary cannot acquire a pointer to an unreachable cell and thus not mimic the free. Consequently, one has to relax the assumption of statelessness. This inflicts major changes on the fixed point from Section 4. Besides program threads, it would need to include threads executing stateful summaries. Moreover, one would need to reintroduce interference steps. However, only such interference steps are required where stateful summaries appear as the interfering thread. Hence, the number of interference steps is expected to be significantly lower than for ordinary interference. We consider a proper investigation of these issues an interesting subject for future work.

8 Related Work

We already commented on the two approaches of computing interference steps. The merge-and-project approach [4,11,19,22] suffers form low scalability and precision due to computing too many merge-compatible heaps. To improve precision of interference, works like [12,30,34] track additional thread correlations; ownership, for instance. However, keeping more information within thread states usually has a negative impact on scalability. Moreover, for the programs of our interest, those techniques were not applicable in the case of explicitly managed memory which does not provide exclusivity guarantees. Instead, [4,2] proposed to maintain views of two threads, allowing one to infer the context in which a views occurs. Since this again jeopardizes scalability, [14] tailored ownership towards explicit memory management. Still, computing interference remained quadratic in the size of the fixed point. Our approach improves dramatically on the efficiency of [14] while keeping its precision.

The learning approach in [32,34,35] and [24,25,26] performs a variant of rely/guarantee reasoning [18] paired with symbolic execution and abstract interpretation, respectively. In a fixed point, the interference produced by a thread is recorded and applied to other threads in consecutive iterations. This computes a symbolic representation of the inteference which is as precise as the underlying abstract domain (although the precision may be relaxed by further abstraction and hand-crafted joins). Our method improves on this in various aspects. First, we never compute the most precise interference information. Our summaries can be understood as a form of interpolant between the most precise approximation and the complement of the bad states. Second, our summaries are syntactic objects (program code) which are independent of the actual verification procedure and thus reusable. The learned interference may be reused only in the same abstract domain it was computed in. Third, we show how to lift our approach to explicit memory management what has not been done before. Fourth, our results are independent of the actual program semantics relying only on a small core language. Our development required to formulate the principles that libraries rely on (statelessness) which have not been made explicit elsewhere.

Another approach to make the verification of low-level implementations tractable is atomicity abstraction [1,10,9,20,28,27]. The core idea is to translate a given program into its specification by introducing and enlarging atomic blocks. The code transformations must be provably sound, with the soundness arguments oftentimes crafted for a particular semantics only. While generating summaries is closely related to making the program under scrutiny more atomic, we pursue a different approach. Our rewriting rules (i.e. the computation of summaries) do not need to be, and indeed are not, provably sound, which allows for much more freedom. Nevertheless, we guarantee a sound analysis. Our sanity checks can be understood as an efficient, fully automatic procedure to check whether or not the applied atomicity abstraction was sound. Additionally, we do not rely on a particular memory semantics.

Simulation relations are widely used for linearizability proofs [8,9,29,36] and verified compilation [17,21]. There, one establishes a simulation relation between a low-level program and a high-level program stating that the latter preserves the behaviors of the former. Verifying properties of the low-level program then reduces to verifying the same property for the high-level program. Establishing simulation relations, however, suffers from the same shortcomings as atomicity abstraction.

Finally, [13] introduces *grace periods*, an idiom similar to CAS blocks. It reflects the protocol used by a program to prohibit data corruption. During a grace period, it is guaranteed that a thread's memory is not freed. However, no method for checking conformance to such periods is given. That is, soundness of the analysis results cannot be checked when relying on grace periods whereas our sanity checks can efficiently detect unsound verification results.

9 Conclusion

We proposed a new approach for verifying lock-free data structures. The approach builds on the so-called CAS blocks (or, more generally, copy-and-check code blocks) which are commonly used when implementing lock-free data structures. We proposed a heuristic that builds stateless program summaries from such blocks. By avoiding many

expensive merge-and-project operations, the approach can greatly increase the efficiency of thread-modular verification. This was confirmed by our experimental results showing that the implementation of our approach compares favorably with other competing tools. Moreover, our approach naturally combines with recently proposed reasoning about ownership to improve the precision of thread-modular reasoning, which allowed us to handle complex lock-free code efficiently even under explicit memory management. Of course, our heuristically computed stateless summaries can miss some reachable shared heaps, but, as a major part of our contribution, we proved that one can check whether this is the case on the generated state space. Hence, we can perform sound verification using a potentially unsound abstraction.

In the future, we would like to investigate CEGAR to include missing effects into our summaries. The main question here is how to refine the program code of a summary using an abstract representation of the missing effects. Further, it may be necessary to introduce stateful summaries in order to include certain effects, as revealed by the DGLM queue under explicit memory management. Moreover, in theory, our approach could increase not only efficiency but also precision compared with other approaches. This is due to the atomicity of the CAS blocks that could rule out interleavings that other approaches would explore. We have not found this confirmed in our experiments. Nevertheless, we find it worth investigating the theoretical and practical aspects of this matter in the future.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. In: LICS. pp. 165–175. IEEE (1988)
2. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: TACAS. LNCS, vol. 7795, pp. 324–338. Springer (2013)
3. Abdulla, P.A., Jonsson, B., Trinh, C.Q.: Automated verification of linearization policies. In: SAS. LNCS, vol. 9837, pp. 61–83. Springer (2016)
4. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, S.: Thread quantification for concurrent shape analysis. In: CAV. LNCS, vol. 5123, pp. 399–413. Springer (2008)
5. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: a technique to pass information between verifiers. In: SIGSOFT FSE. p. 57. ACM (2012)
6. Christakis, M., Wüstholz, V.: Bounded abstract interpretation. In: SAS. LNCS, vol. 9837, pp. 105–125. Springer (2016)
7. Cook, B., Haase, C., Ouaknine, J., Parkinson, M.J., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: CONCUR. LNCS, vol. 6901, pp. 235–249. Springer (2011)
8. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE. LNCS, vol. 3235, pp. 97–114. Springer (2004)
9. Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying linearizability proofs with reduction and abstraction. In: TACAS. LNCS, vol. 6015, pp. 296–311. Springer (2010)
10. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL. pp. 2–15. ACM (2009)
11. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: SPIN. LNCS, vol. 2648, pp. 213–224. Springer (2003)
12. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI. pp. 266–277. ACM (2007)

13. Gotsman, A., Rinetzky, N., Yang, H.: Verifying concurrent memory reclamation algorithms with grace. In: ESOP. LNCS, vol. 7792, pp. 249–269. Springer (2013)
14. Haziza, F., Holík, L., Meyer, R., Wolff, S.: Pointer race freedom. In: VMCAI. LNCS, vol. 9583, pp. 393–412. Springer (2016)
15. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS 12(3), 463–492 (1990)
16. Holík, L., Meyer, R., Vojnar, T., Wolff, S.: Effect summaries for thread-modular analysis. CoRR abs/1705.03701 (2017), http://arxiv.org/abs/1705.03701
17. Jagannathan, S., Petri, G., Vitek, J., Pichardie, D., Laporte, V.: Atomicity refinement for verified compilation. In: PLDI. p. 27. ACM (2014)
18. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP. pp. 321–332 (1983)
19. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM TOPLAS 5(4), 596–619 (1983)
20. Jonsson, B.: Using refinement calculus techniques to prove linearizability. Formal Asp. Comput. 24(4-6), 537–554 (2012)
21. Leroy, X.: A formally verified compiler back-end. JAR 43(4), 363–446 (2009)
22. Malkis, A., Podelski, A., Rybalchenko, A.: Thread-modular verification is cartesian abstract interpretation. In: ICTAC. LNCS, vol. 4281, pp. 183–197. Springer (2006)
23. Michael, M.M., Scott, M.L.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. JPDC 51(1), 1–26 (1998)
24. Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: ESOP. LNCS, vol. 6602, pp. 398–418. Springer (2011)
25. Miné, A.: Relational thread-modular static value analysis by abstract interpretation. In: VMCAI. LNCS, vol. 8318, pp. 39–58. Springer (2014)
26. Monat, R., Miné, A.: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In: VMCAI. LNCS, vol. 10145, pp. 386–404. Springer (2017)
27. Popeea, C., Rybalchenko, A., Wilhelm, A.: Reduction for compositional verification of multi-threaded programs. In: FMCAD. pp. 187–194. IEEE (2014)
28. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: Tada: A logic for time and data abstraction. In: ECOOP. LNCS, vol. 8586, pp. 207–231. Springer (2014)
29. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM TOCL 15(4), 31:1–31:37 (2014)
30. Segalov, M., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Abstract transformers for thread correlation analysis. In: APLAS. LNCS, vol. 5904, pp. 30–46. Springer (2009)
31. Treiber, R.: Systems programming: coping with parallelism. Tech. Rep. RJ 5118, IBM (1986)
32. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: VMCAI. LNCS, vol. 5403, pp. 335–348. Springer (2009)
33. Vafeiadis, V.: Automatically proving linearizability. In: CAV. LNCS, vol. 6174, pp. 450–464. Springer (2010)
34. Vafeiadis, V.: RGSep action inference. In: VMCAI. LNCS, vol. 5944, pp. 345–361. Springer (2010)
35. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. LNCS, vol. 4703, pp. 256–271. Springer (2007)
36. Zhang, S.J., Liu, Y.: Model checking a lazy concurrent list-based set algorithm. In: SSIRI. pp. 43–52. IEEE (2010)