Automated Robustness Verification of Concurrent Data Structure Libraries against Relaxed Memory Models

Extended Version

KARTIK NAGAR, IIT Madras, India
ANMOL SAHOO, Purdue University, USA
ROMIT ROY CHOWDHURY, Chennai Mathematical Institute, India
SURESH JAGANNATHAN, Purdue University, USA

Clients reason about the behavior of concurrent data structure libraries such as sets, queues, or stacks using specifications that capture well-understood correctness conditions, such as linearizability. The implementation of these libraries, however, focused as they are on performance, may additionally exploit relaxed memory behavior allowed by the language or underlying hardware that weaken the strong ordering and visibility constraints on shared-memory accesses that would otherwise be imposed by a sequentially consistent (SC) memory model. As an alternative to developing new specification and verification mechanisms for reasoning about libraries under relaxed memory model, we instead consider the orthogonal problem of *library robustness*, a property that holds when all possible behaviors of a library implementation under relaxed memory model are also possible under SC. In this paper, we develop a new *automated* technique for verifying robustness of library implementations in the context of a C11-style memory model. This task is challenging because a most-general client may invoke an unbounded number of concurrently executing library operations that can manipulate an unbounded number of shared locations. We establish a novel inductive technique for verifying library robustness that leverages prior work on the robustness problem for the C11 memory model based on the search for a non-robustness witness under SC executions. We crucially rely on the fact that this search is carried out over SC executions, and use high-level SC specifications (including linearizability) of the library to verify the absence of a non-robustness witness. Our technique is compositional - we show how we can safely preserve robustness of multiple interacting library implementations and clients using additional SC fences to guarantee robustness of entire executions. Experimental results on a number of complex realistic library implementations demonstrate the feasibility of our approach.

Additional Key Words and Phrases: Relaxed Memory Models, Concurrent Library implementations, Robustness

1 Introduction

Unlike a sequentially consistent (SC) [41] memory model where concurrently executing threads all witness the same view of memory, relaxed memory models such as those found in C11 [7, 8, 15] allow each thread to have its own logical view of memory, resulting in behaviors like store buffering that are not expressible under SC [37]. A program written to take advantage of the performance benefits afforded by executing under a relaxed memory model M, but whose behaviors can nonetheless be fully explained in terms of SC (interleaved) executions, is said to be *robust* against M. Intuitively, a robust program is one in which every relaxed memory execution is also an SC one.

In this paper, we consider the problem of *automatically* verifying the robustness of library implementations against RC20 [43], a useful variant of the C11 memory model. C11 allows the expression of high-performance concurrent C/C++ code through the use of annotations affixed to memory accesses. Release (`rel`) and acquire (`acq`) annotations enable a form of message-passing between "release" writes and "acquire" reads [38], yielding a causally-consistent ordering between

Authors' Contact Information: Kartik Nagar, IIT Madras, Chennai, India, nagark@cse.iitm.ac.in; Anmol Sahoo, Purdue University, West Lafayette, USA, sahoo9@purdue.edu; Romit Roy Chowdhury, Chennai Mathematical Institute, Chennai, India, romit@cmi.ac.in; Suresh Jagannathan, Purdue University, West Lafayette, USA, suresh@cs.purdue.edu.

2024. ACM 2475-1421/2024/10-ART362
https://doi.org/10.1145/3689802

the operations that precede the "release" and those that follow the "acquire"; and, relaxed (`rlx`) annotations are used to compile accesses to single hardware loads and stores with no additional synchronization, other than basic store coherence guarantees [33] provided by the hardware. RC20 provides an improved semantics for atomic accesses [40], prevents undesirable executions involving out-of-thin-air reads, and simplifies the semantics of synchronization for release-acquire pairs.

Robustness under the RC20 memory model is a well-studied problem [38, 43] for which algorithms to check robustness of finite-state programs operating over a bounded number of threads have been developed. By showing a program to be robust, one can then reason about its correctness w.r.t. a high-level specification under the SC memory model, which is a significantly simpler problem. Consequently, any correctness guarantees proven under SC also continue to hold under RC20. Thus, proving library robustness can also enable a similar pathway to reasoning about correctness of programs using library implementations under RC20. Prior efforts addressing the correctness of library implementations under relaxed memory models have mostly focused on developing newer forms of specification [18, 44, 49, 52] that allows reasonable non-SC behaviors and also exposes the internals of the underlying memory model. While these specifications can admit highly efficient library implementations, proving their correctness requires significant manual effort because of the need to closely correlate abstract state-based specifications with the event-based relaxed memory model semantics. Furthermore, using these relaxed specifications to prove correctness of client programs using the library implementation is also highly non-trivial. On the other hand, by proving a library implementation to be robust in the context of the most general client, we can extend this robustness guarantee to any client program using the library, thus allowing one to reason about its correctness under SC, which is a well-studied problem.

Following the classical modular verification paradigm, we would like to separately verify the robustness of library implementations and their clients and then compose these guarantees together to establish robustness of the overall execution. We consider the following strategy: first verify a library implementation to be robust in the presence of a *most general client*, which can call the library methods an arbitrary number of times across an arbitrary number of threads, and then use this robustness guarantee to prove the robustness of any client program potentially calling multiple libraries. Clearly, if a library is not robust on its own, any client program using the library (and which does not restrict the concurrent behaviors of the library) would also be non-robust.

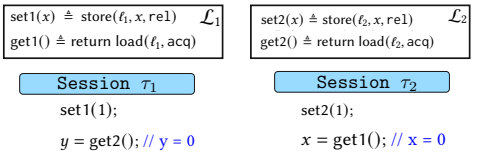

Fig. 1. Example showing non-compositionality of robustness

Unfortunately, we find that naïvely composing individually robust libraries may not lead to an overall robust execution. To illustrate this, consider the two simple register libraries $\mathcal{L}_1$ and $\mathcal{L}_2$ given in Fig. 1. It is well known that under the RC20 memory model, atomic accesses to a single location follow SC-per-location semantics, and thus both the libraries (which access single distinct locations ℓ_1 and ℓ_2) will be individually robust for the most general client. However, we can construct a standard store-buffering execution formed out of interaction with both libraries, as shown by the sessions τ_1 and τ_2 in the figure (assume that the initial value at locations ℓ_1 and ℓ_2 is 0). This execution is not possible under SC, because in any interleaving, at least one of set1 or set2 (and hence the underlying store) operations will happen first, so that at least one of the get1 or get2 operations would return 1. To recover overall robustness, we can add an SC-fence when crossing libraries within every session. In the RC20 memory model, an SC-fence is defined using a composition of three instructions: fence(`acq`); fadd(f, 0, `acqrel`); fence(`rel`) where f is

an identifier for a distinguished location not accessed by any other instruction [43]. The fadd instruction guarantees synchronization between any two SC fences.

In the execution above, suppose we place an SC fence between the two invocations in each session. Then, the total ordering among the fences would prevent both the loads from ignoring the stores under RC20, thus prohibiting the store buffering anomaly. On the other hand, if even one of the SC-fences is removed, store buffering is still possible. We call an execution *well-fenced* if there is an SC-fence between every adjacent pair of invocations to different libraries in every session.

Having well-fenced executions then gives us a way to decompose the overall robustness problem into robustness of individual libraries. In this work, we leverage this observation and make significant progress on the problem of verifying robustness of programs using library implementations: (i) we propose an automated, sound (but not complete) verification procedure for proving robustness of individual library implementations against the most general client, (ii) we formally prove that a well-fenced execution comprising of calls to individually robust libraries is guaranteed to be robust.

Verifying individual library robustness conceptually requires us to compare the set of all SC and RC20 executions of the library implementation. This is a non-trivial problem because the most general client can invoke library methods an arbitrary number of times concurrently across an arbitrary number of threads. Previous efforts on robustness verification of relaxed memory programs [11, 38, 43] are only applicable for finite state programs involving a bounded number of threads. Our primary contribution in this regard is a fully automated verification strategy that is not constrained by these restrictions.

Our verification strategy is based on searching SC executions of a library implementation for potential *non-robustness witnesses* [38, 43]. SC executions are constructed by simply treating all memory accesses as being sequentially consistent, ignoring the actual annotations affixed to these accesses in the program. A non-robustness witness is a prefix of an SC execution that contains a location whose latest write w would be witnessed by the next operation (say a read r) in that execution, but which may not necessarily be witnessed in an RC20 execution, i.e. informally, $\mathsf{hb}_{\mathsf{SC}}(w, r)$ holds but $\mathsf{hb}_{\mathsf{RC20}}(w, r)$ does not, where $\mathsf{hb}_{\mathsf{SC}}$ and $\mathsf{hb}_{\mathsf{RC20}}$ define a happens-before ordering among read and write actions in SC and RC20 executions, resp. Verifying the absence of such a witness is tantamount to showing robustness in a RC20 setting [38, 43]. We use an inductive strategy to cover the infinite set of executions, and crucially take advantage of axioms and constraints provided by the library's SC specification to help discharge the verification conditions.

We note that unlike prior approaches to robustness verification [38, 43] which are both sound and complete, our approach sacrifices completeness for the sake of automation by devising sufficient but not necessary conditions for proving absence of a non-robustness witness. However, this makes our verification strategy more amenable to automation, and also better suited to library implementations. We also crucially rely on expressive SC specifications to rule out infeasible non-robustness witnesses, with the caveat that spurious non-robustness witnesses can arise if the SC specification is not expressive enough. However, our extensive experimental evaluation suggests that the SC specification of the standard data structures is rich enough to verify a number of complex implementations.

This paper makes the following contributions:

(1) We provide a verification methodology for proving robustness of a concurrent data structure library executing under the RC20 memory model. To enable automated verification, our proof methodology exploits various constraints derived from the library's SC specification. We provide a common SMT-based framework to systematically relate these high-level constraints with the low-level event-based guarantees required for robustness proofs.

```
1   void set(int v) {
2     node* N = malloc(sizeof(node));
3     store(N→ val, v, rlx);
4     store(L, N, rel) ;
5   }
6
```

```
4   int get {
5     node* r = load(L, acq);
6     if (r == NULL)
7     return UNDEF;
8     else {
9       int v = load(r→ val,rlx);
10      return v;
11    }
12  }
13
```

Fig. 2. A stylized C11 implementation of a Register library. The contents of variables N and L can be accessed by multiple threads; variable r, on the other hand, is thread-local.

(2) We introduce a new notion of robustness, called *induced subgraph robustness*, tailored for reasoning about robustness of library implementations that may have benign non-robustness as part of their speculative computations. This weaker definition permits executions to have library-internal non-robust operations, as long as their effects do not manifest in a method's return value.
(3) We extend our methodology for proving robustness of an individual library to compose multiple robust libraries using SC fences to guarantee overall execution robustness. Our composition guarantees also extend to induced subgraph robustness. Our initial experiments indicate that the performance penalty of putting additional SC fences is not significant, and potentially masked by the synchronization already performed by the individual libraries to maintain robustness.
(4) We have applied our approach on a number of realistic concurrent C11 library implementations, including real-world benchmarks such as Meta's Folly lock-free queue implementation that make sophisticated use of relaxed atomics [26]. Our results establish, often for the first time, robustness against RC20 of well-known library implementations.

The remainder of the paper is structured as follows. The next section presents a motivating example to illustrate the core ideas underlying our approach for proving robustness of a library. §3 introduces a representative language, along with an axiomatic formalization of the memory model. The derivation of the inductive invariant used for our robustness proof is given in §4. We formalize the definition of induced subgraph robustness in §5. We provide our methodology to compose robust libraries in §6. Details about the implementation and SMT encoding are given in §7. Experimental results are presented in §8. Related work and conclusions are given in §9.

2 Motivating Example

Consider an implementation of a concurrent register data structure shown in Figure 2. Although not very efficient, it nonetheless exhibits access patterns commonly found in a number of real-world library implementations, and it allows us to provide a simple, concise demonstration of the core ideas of our verification strategy. The register data structure has two methods (`set(v)` and `get`), with the intended semantics that `get` returns the most recent value that was assigned to the register by a `set` operation, or else `UNDEF` if no `set` operations have yet executed. In Figure 2, the `set` method is implemented by allocating a new node containing the value (in the field `val`), storing a reference to the node in a shared variable `L`. The `get` method reads the variable `L` and then derefences the `val` field (if `L` is not `NULL`) to return the value. Notice that each memory access is annotated with an access mode, which has an effect on its behavior, as well as the behavior of subsequent accesses.

If we ignore the annotations defining relaxed access modes and assume all reads and writes to shared-memory are SC, then every concurrent execution can be thought of as an interleaving of

statements from different threads, all accessing a single shared memory. Under SC, it is straightforward to prove that the above implementation is correct, which means that all the invocations in an execution follow the intended register semantics.

2.1 RC20 Memory Model and Robustness

In order to describe executions under the RC20 relaxed memory model, we consider the read/write events generated during an execution, as well as dependency relations between these events. In the following, we give a brief informal description of these relations; they are formally defined in the next section. The relations include (i) *reads-from* (rf) that relates a write event to the read event which reads from it, (ii) *modification-order* (mo), a total ordering on all write events to the same location, and (iii) *from-read* (fr), a relation that relates a read event to write events that occur later than the write event it reads from according to the mo order. Events in the same thread are all totally ordered according to a session order (so) relation, a generalization of program order adapted to a concurrent library setting (a session is defined as a sequence of library method invocations performed by the same thread).

The access modes associated with read/write events are used to determine the *synchronizes-with* relation (sw): write and read events that are related by rf are also related by sw if the write is annotated with `rel` access mode and the read is annotated with `acq` mode (there are also other ways to establish sw, more details in §3.2). Finally, these primitive relations are used to define two derived relations: (i) $\mathsf{hb}_{\mathsf{SC}}$, the transitive closure of rf, fr, mo, so, and (ii) hb, the transitive closure of sw and so. Intuitively, $\mathsf{hb}_{\mathsf{SC}}$ will be used to define the behavior of SC executions, while hb will define the behavior of RC20 executions.

For the purpose of this section, we are mainly interested in how the RC20 memory model uses hb to constrain the behavior of read events by forcing them to not ignore write events that are in hb order to them. This means that a read event e_r cannot read from a write event e_w if there is another write event e'_w such that $e_w \xrightarrow{\mathsf{mo}} e'_w$ and $e'_w \xrightarrow{\mathsf{hb}} e_r$ (more precisely, e'_w must be in hb-order before some previous event in the session containing e_r). In this case, e_r must read from either e'_w or an event mo-after e'_w.

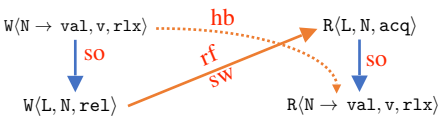

Fig. 3. An execution of the register library

To illustrate this, consider an execution of the register library consisting of two invocations, `set(v)` and `get`. The events generated during this execution are shown in Figure 3, with the two events on the left side of the figure generated by `set(v)` and the two events on the right generated by `get`. Assuming that the `malloc` statement in `set(v)` allocates a new node `N`, the rf relation from the store of `L` in `set` to the load of `L` in `get` induces a sw relation (because of the `rel` and `acq` annotations associated with the store operation in the `set` method (line 4) and the load operation in the `get` method (line 5)). Due to this, the write of `N`→`val` in `set` comes in hb-order before the read of `N`→`val` in `get`. As a result, this load must read the value `v`. However, if the access mode of either the load or store to `L` are changed to `rlx`, then the store to `N`→`val` would no longer be in hb order, allowing the load to `N`→`val` to ignore it. The execution in Figure 3 corresponds to classical message-passing behavior, and is possible under both SC and RC20.

The robustness problem then asks whether any arbitrary RC20 execution of the library implementation is also possible under SC. In order to solve the robustness problem, we consider an alternative event-based characterization of SC executions: SC constrains the behavior of read events by forcing them to not ignore write events which are in $\mathsf{hb}_{\mathsf{SC}}$ order before them. This characterization is almost

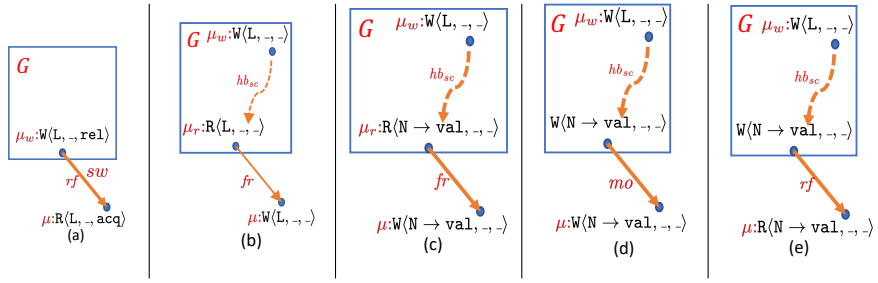

Fig. 4. Establishing the inductive invariant for the location L in the register library implementation

identical to the RC20 memory model, replacing the hb relation with $\mathsf{hb}_{\mathsf{SC}}$, and this observation leads to the following sufficient condition for robustness[43]: *At every step during an SC execution, if the next event is a read event e_r to location ℓ, and if the most recent write event e_w to ℓ (according to the* mo *ordering) is in* $\mathsf{hb}_{\mathsf{SC}}$ *order to the session τ containing e_r, then e_w is also in* hb *ordering to τ.* This means that, under RC20, the read event e_r cannot ignore the write event e_w, and hence it must have the same behavior as under SC. Notice that although we are trying to establish robustness of RC20 executions, the robustness condition itself needs to be checked over SC executions only. This is because we are essentially searching for the minimal or first robustness violation, and hence the execution leading up to the violation must be itself robust and hence possible under SC.

Coming back to the execution in Figure 3, just before the event $\mathtt{R}\langle \mathtt{N} \rightarrow \mathtt{val}, \mathtt{v}, \mathtt{rlx} \rangle$, the most recent write event to the location $\mathtt{N} \rightarrow \mathtt{val}$, $\mathtt{W}\langle \mathtt{N} \rightarrow \mathtt{val}, \mathtt{v}, \mathtt{rlx} \rangle$, is in both $\mathsf{hb}_{\mathsf{SC}}$ and hb order, thus ensuring that the read event has the same behavior under both SC and RC20. On the other hand, if the access modes of either the load or store to L were to be changed to rlx, robustness would be violated by the read event to N⟶val, resulting in a manifestation of a non-robustness witness.

2.2 Verifying Robustness

Our verification strategy aims to prove the absence of non-robustness witnesses, which are nothing but violations of the sufficient condition for robustness: an SC execution of the library implementation that contains a write event on a location ℓ that is in $\mathsf{hb}_{\mathsf{SC}}$ but not hb order to a session where the next event is a read event on ℓ. Since a library implementation can be invoked an arbitrary number of times by clients across arbitrary number of threads, we have to reason about robustness of an unbounded number of executions. To make this problem tractable, we define an inductive proof strategy that essentially uses the absence of non-robustness witnesses as an inductive invariant. We prove the correctness of the inductive invariant systematically on a per-location basis, and use information about the SC behavior of the library implementation (such as its high-level specification under SC) to facilitate automated verification.

To illustrate, we describe how to verify the robustness of the register implementation of Figure 2. Executions of this implementation access two classes of locations: (i) L and (ii) N⟶val for different values of N. We reason about the absence of non-robustness witnesses involving these location classes separately. Intuitively, the rel/acq annotations to the store/load of location L should ensure robust behavior of loads to location $\mathtt{N} \rightarrow \mathtt{val}$. However, it is not clear how the implementation ensures robustness for accesses to location L. We prove that it does so by establishing the following inductive invariant over all SC executions: *if the most recent write to the location* L *(according to* mo*) comes* $\mathsf{hb}_{\mathsf{SC}}$ *before another event, then it also comes* hb *before it.* This would then ensure that a subsequent load to L would have the same behavior under SC and RC20. We use induction on the

number of events in an SC execution, and consider adding events in an order that obeys the $\mathsf{hb}_{\mathsf{SC}}$ relation. Figure 4 depicts some of the more interesting inductive cases; for each case, we consider an arbitrary SC execution G of the library (depicted by the rectangle) and a new event μ to be added to the execution. The most recent write to location L in the execution G is labelled μ_w. Inductively, we assume that if μ_w comes $\mathsf{hb}_{\mathsf{SC}}$ before another event in G, it also comes hb before it.

The various cases exhaustively consider new additions to $\mathsf{hb}_{\mathsf{SC}}$ due to the inclusion of μ. We highlight the different ways in which the inductive invariant is maintained. Figure 4(a) considers the case where μ is a read event to L, which can cause a new $\mathsf{hb}_{\mathsf{SC}}$ edge to be established through the rf relation. Now, since all writes to L use the access mode `rel` while all reads to L use access mode `acq`, the rf relation induces a sw relation, and thus a hb relation so that the required condition continues to hold. Figure 4(b) depicts an execution in which μ is a write to L that results in an incoming fr edge from an existing read (μ_r) in G. Note that the dotted arrow labeled with $\mathsf{hb}_{\mathsf{SC}}$ indicates that the $\mathsf{hb}_{\mathsf{SC}}$ relation is established through some sequence of events. Now, since μ_w occurs $\mathsf{hb}_{\mathsf{SC}}$-before μ_r, it also occurs $\mathsf{hb}_{\mathsf{SC}}$-before μ. However, in the new execution after addition of the event μ, notice that μ now becomes the most recent write to L, and it cannot occur $\mathsf{hb}_{\mathsf{SC}}$ before any event in G (because we are adding events in $\mathsf{hb}_{\mathsf{SC}}$ order). Hence, the required invariant holds trivially, and we do not have to consider any $\mathsf{hb}_{\mathsf{SC}}$ paths from μ_w.

A more interesting case is shown in Figure 4(c). Here μ is a write event to `N→val` and its addition to the execution establishes a new $\mathsf{hb}_{\mathsf{SC}}$ path from μ_w through a fr relation on μ_r. On the surface, it would seem that this depicts a non-hb (but $\mathsf{hb}_{\mathsf{SC}}$) path from μ_w to μ. However, in an actual execution, the fr relation involving `N→val` can never arise. In the following, we show how we can infer the infeasibility of this execution by using the fact that we are only considering SC executions.

Notice that in a SC execution, there is a unique write to any location of the class `N→val`, because it only happens inside the `set` method, and `malloc` will always return a fresh location. Since μ_r does not read from this write event, and since there is no other write event to `N→val`, μ_r must read the initial value of the location (say 0), which will also be returned by `get` invocation containing μ_r. However, the register implementation is agnostic to the actual value passed as an argument to `set` (i.e., `set`'s control-flow is unaffected by v) and thus we can assume data independence of its arguments [1]. In particular, we prohibit executions where `set` is passed the initialization value 0. But the `get` invocation containing μ_r returns 0, and the register specification says that for every `get` invocation which return a non-`UNDEF` value, there must be a `set` invocation whose argument value is the same as the return value of the `get` invocation. We thus derive a contradiction, establishing infeasibility of the execution in Fig. 4(c). Notice how we are able to leverage program structure constraints (uniqueness of write to `N→val` enforced by `malloc`) and the register specification for this reasoning.

The cases depicted in Figures 4(d) and 4(e) can also be handled in a fashion similar to the handling of cases 4(a) - 4(c). In particular, the execution shown in Figure 4(d) is not possible due to uniqueness of writes to `N→val`. Figure 4(e) depicts a feasible execution. In this case, there will already exist a hb ordering between the event $\mathtt{W}\langle\mathtt{N{\rightarrow}val}, \_, \_\rangle$ and μ (as shown in Figure 3) via a path comprising sw and so edges. Using the inductive hypothesis that μ_w also occurs hb-before $\mathtt{W}\langle\mathtt{N{\rightarrow}val}, \_, \_\ \rangle$ and the transitivity of hb, we can infer that μ_w occurs hb-before μ. Collectively, our case analysis shows robustness for the location L, guaranteeing that there can be no non-robustness witnesses caused by accesses to this location. In each case, either we cannot establish the $\mathsf{hb}_{\mathsf{SC}}$ relation, or if we can, then hb relation is also established. A similar analysis can be carried out for the location `N→val`.

The following sections formalize these intuitions, generalizes the approach to deal with implementations that use loops and synchronization operations like `cas` and `fence`, and builds an automated procedure to perform the above reasoning.

$c \in$ Constant
$\ell \in$ Location
$x, y \in$ Var
$\tau \in$ SessionId
$m \in$ MethodName
$\mu \in$ MemEvt
$o_X \in \text{Mod}_X$ where $X \in \{\mathsf{R}, \mathsf{W}, \mathsf{U}, \mathsf{F}\}$

$$
\begin{aligned}
v &\in \text{Value} ::= c \mid \ell \\
e &\in \text{Expr} ::= v \mid x \mid e1 == e2 \mid \ldots \\
I &\in \text{Method} ::= \text{method } m(x) \ = \ s \\
s &\in \text{Stmt} ::= \text{skip} \mid x = e \mid x = \text{malloc}(c) \\
&\quad \mid \text{store}(x, e, o_{\mathsf{W}}) \mid y = \text{load}(x, o_{\mathsf{R}}) \mid y = \text{cas}(x, e_1, e_2, o_{\mathsf{U}}, o_{\mathsf{R}}) \mid \text{fence}(o_{\mathsf{F}}) \\
&\quad \mid y = \text{fadd}(x, e, o_{\mathsf{U}}) \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s \mid \text{return } e \\
\alpha &\in \text{Action} ::= \epsilon \mid \mathsf{R}\langle \ell, v, o_{\mathsf{R}} \rangle \mid \mathsf{W}\langle \ell, v, o_{\mathsf{W}} \rangle \mid \mathsf{U}\langle \ell, v_1, v_2, o_{\mathsf{U}} \rangle \mid \mathsf{F}\langle o_{\mathsf{F}} \rangle \\
\gamma &\in \text{InvkEvt} ::= \text{Invk}\langle i, m, v_a, v_r, \tau \rangle \\
\omega &\in \text{Evt} ::= \gamma \mid \text{Sil}\langle \gamma \rangle \mid \text{Mem}\langle i, \tau, \alpha \rangle
\end{aligned}
$$

Fig. 5. Domains and language syntax.

3 Preliminaries

3.1 Syntax and Program Semantics

To describe our approach, we consider a C-style imperative language, with standard control-flow operators (sequencing (;), while, if-then-else, etc.), and operations to access thread-local and shared state (Figure 5). A library implementation of a data structure is given by a set of methods written in this language, with each method corresponding to an operation of the data structure; we prohibit methods from invoking other methods.

Library implementations have access to a shared heap i.e., an unbounded collection of locations (Location), shared among all sessions; additionally, library methods can bind and read local variables whose values are accessible to the method only for the lifetime of the invocation. We assume that every instance of the malloc(c) command returns a unique location on the heap, allocating c units of memory. We also require that a library implementation owns its accessed locations, which means they cannot be accessed by clients (or other library implementations) [49]. A method body is a sequence of statements, at least one of which must be a return.

The most interesting part of the language are statements that access shared memory. Specifically, statements load(x,o_{R}) and store(x,e,o_{W}) are used to load values from, and store values to, locations in shared memory whose address is contained in variable x, resp. Following the RC20 memory model, each shared memory access in our language is also associated with an *access mode*, taken from the set $\text{Mod} \triangleq \{\texttt{rlx}, \texttt{acq}, \texttt{rel}, \texttt{acqrel}\}$. These modes define the consistency level of the memory access. Note that the language does not allow non-atomics, as every access is identified with a consistency level. We define specific subsets of access modes for the different types of memory accesses:

$$
\begin{aligned}
\text{Mod}_{\mathsf{R}} &= \{\texttt{rlx}, \texttt{acq}\} & \text{Mod}_{\mathsf{U}} &= \{\texttt{rlx}, \texttt{acq}, \texttt{rel}, \texttt{acqrel}\} \\
\text{Mod}_{\mathsf{W}} &= \{\texttt{rlx}, \texttt{rel}\} & \text{Mod}_{\mathsf{F}} &= \{\texttt{acq}, \texttt{rel}, \texttt{acqrel}\}
\end{aligned}
$$

A partial order $\sqsubseteq$ is also defined over these access modes, which intuitively orders them according to their consistency level:$\texttt{rlx} \sqsubseteq \texttt{acq} \sqsubseteq \texttt{acqrel}$ and $\texttt{rlx} \sqsubseteq \texttt{rel} \sqsubseteq \texttt{acqrel}$. Additionally, the cas($x$,$e_1$,$e_2$, o_{U}, o_{R}) operation atomically updates the location ℓ bound to x to the evaluation of e_2 if e_1 evaluates to the value stored in ℓ, returning true in this case, and returns false otherwise.

The semantics of library implementations are defined in terms of a most general client which can call arbitrary methods in any arbitrary session. Formally, for a library $\mathcal{L} \in \mathcal{P}(\text{Method})$, we define a labeled transition system (LTS), $\Omega_{\mathcal{L}} = (\Phi, \text{Evt}, \Rightarrow)$, where Φ denotes a set of states, Evt denotes a set of events (also used as labels on transitions) and $\Rightarrow \subseteq \Phi \times \text{Evt} \times \Phi$ defines a transition relation over states and events. We write $\overset{\omega}{\Longrightarrow}$ for the relation $\{\langle s_1, s_2 \rangle \| \langle s_1, \omega, s_2 \rangle \in \Rightarrow\}$.

$$\mathcal{I} \in \text{Invocations} = \text{SessionId} \to (\text{InvkEvt} \cup \{\bot\}) \times (\text{Stmt} \cup \{\bot\})$$
$$\varrho \in \text{Environments} = \text{SessionId} \to \text{EnvLocal}$$
$$\mathcal{L} \in \text{Library} \in \mathcal{P}(\text{Method})$$

$$\rho \in \text{EnvLocal} = \text{Var} \to \text{Value}$$

INVK

$$\frac{\mathcal{I}(\tau) = (\_, \bot) \quad m = \lambda x.s \in \mathcal{L} \quad \varrho' = \varrho[\tau \mapsto \varrho(\tau)[x \mapsto v_a]] \quad i \text{ unique} \quad \gamma = \mathsf{Invk}\langle i, m, v_a, v_r, \tau\rangle \quad \mathcal{I}' = \mathcal{I}[\tau \mapsto \langle \gamma, s\rangle]}{\langle \varrho, \mathcal{I}\rangle \overset{\gamma}{\Rightarrow} \langle \varrho', \mathcal{I}'\rangle}$$

RET

$$\frac{\mathcal{I}(\tau) = \langle \gamma, \mathsf{return}(e);\ s\rangle \quad \gamma = \mathsf{Invk}\langle \_, \_, \_, v_r, \tau\rangle \quad e \rightsquigarrow_{\varrho(\tau)} v_r \quad \mathcal{I}' = \mathcal{I}[\tau \mapsto \langle \gamma, \bot\rangle] \quad \varrho' = \varrho[\tau \mapsto \rho_\bot]}{\langle \varrho, \mathcal{I}\rangle \overset{\epsilon}{\Rightarrow} \langle \varrho', \mathcal{I}'\rangle}$$

STEPLOCAL

$$\frac{\mathcal{I}(\tau) = \langle \gamma, s\rangle \quad \langle \varrho(\tau), s\rangle \xrightarrow{\epsilon}_\tau \langle \rho', s'\rangle \quad \mathcal{I}' = \mathcal{I}[\tau \mapsto \langle \gamma, s'\rangle] \quad \varrho' = \varrho[\varrho(\tau) \mapsto \rho']}{\langle \varrho, \mathcal{I}\rangle \overset{\mathsf{Sil}\langle\gamma\rangle}{\Longrightarrow} \langle \varrho', \mathcal{I}'\rangle}$$

STEPMEM

$$\frac{\mathcal{I}(\tau) = \langle \gamma, s\rangle \quad \langle \varrho(\tau), s\rangle \xrightarrow{\alpha}_\tau \langle \rho', s'\rangle \quad \mathcal{I}' = \mathcal{I}[\tau \mapsto \langle \gamma, s'\rangle] \quad \varrho' = \varrho[\varrho(\tau) \mapsto \rho'] \quad i \text{ unique} \quad \mu = \mathsf{Mem}\langle i, \tau, \alpha\rangle}{\langle \varrho, \mathcal{I}\rangle \overset{\mu}{\Rightarrow} \langle \varrho', \mathcal{I}'\rangle}$$

Fig. 6. Global Reduction Rules

A *state* in Φ is a tuple, comprising a set of thread-local environments (ϱ) indexed by session ids and a set of current invocations ($\mathcal{I}$, also indexed by session ids) that records the state of each session along with the invocation event of the method currently executing in that session. In an invocation event, $\mathsf{Invk}\langle i, m, v_a, v_r, \tau\rangle$, i represents an invocation number (a unique identifier for a particular invocation), m is a method name, v_a and v_r represent the argument supplied to and value returned by the method, and τ is a session identifier. The prophesied return value v_r must match the value yielded by e in any `return` e statement executed during the execution of m with invocation number i. This follows the convention introduced by [49] to describe method invocations in a relaxed memory setting using a single event, instead of the traditional invocation and response events used in the SC setting.

We note that SessionId is an unbounded set of sessions. The *initial* state $s_\bot$ of Ω_L is $\langle \varrho_\bot, \mathcal{I}_\bot\rangle$. $\mathcal{I}_\bot$ maps each session in SessionId to $\langle \bot, \bot\rangle$, while $\varrho_\bot$ maps each variable to an initial value.

Figure 6 defines the semantics of clients. It models the behavior of a most-general client. A client can invoke a new method using the INVK rule, which also establishes an initial local environment. Notice that $\mathcal{I}(\tau) = (\_, \bot)$ ensures that τ is not an active session. The RET rule corresponds to the return of a method. We use the notation $e \rightsquigarrow_{\varrho(\tau)} v_r$ to indicate that the expression e evaluate to v_r under the local environment $\varrho(\tau)$. Notice that the RET rule sets the second component of $\mathcal{I}(\tau)$ to $\bot$, thus allowing future calls in τ through INVK. The STEP rules allow a method invocation active in a session to take either a silent step (STEPLOCAL) or a step that affects memory (STEPMEM) [1].

Steps taken by an invocation that do not access shared memory are recorded using a silent event ($\mathsf{Sil}\langle\gamma\rangle$). Steps that involve either reading or writing from shared memory are recorded using memory events that are of the form: $\mathsf{Mem}\langle i, \tau, \alpha\rangle$, where i is a unique event identifier, τ is the session containing the event, and α defines the particular operation performed on the memory (read (R), write (W), read-modify-update (U), or fence (F)). We assume the language modeled by the LTS is receptive [51] and does not constrain the specific values read from memory. This assumption is consistent with prior work [43] that also separates out the memory system from the program semantics, by allowing `load` operations to read arbitrary values from memory. These values are then constrained by the memory system as defined in §3.2.

[1] The local reduction rules defining $\xrightarrow{\alpha}_\tau$ are straightforward and provided in the appendix, §A.

3.2 Memory system

We use declarative specifications of the RC20 memory model to construct the memory system. These specifications constrain *execution graphs*, which consist of the memory events that we defined in the previous subsection, along with a number of binary relations among these events. Formally, an execution graph $G = \langle M, \mathsf{rf}, \mathsf{mo}, \mathsf{so}\rangle$ consists of a set of memory events M along with the binary relations *reads-from* (rf), *modification-order* (mo) and *session-order* (so). Our development closely follows prior work in this area [33, 38, 43].

Notation. For a relation R, we write $R^?$ for the reflexive and R^+ for the transitive closure of R, R^{-1} denotes its inverse, while $dom(R)$ denotes its domain. $R_1; R_2$ denotes the composition of the two relations R_1 and R_2. Given a set of memory events M, $[M]$ denotes the identity relation on M. Hence, $[M_1]; R; [M_2]$ denotes $R \cap (M_1 \times M_2)$. Given an execution graph $G = \langle M, \mathsf{rf}, \mathsf{mo}, \mathsf{so}\rangle$, we use the notation $G.X$ to denote the subcomponent X of G. Given a subset of events $M' \subseteq G.M$ and binary relation $G.R$ over $G.M$, we write $G.R_{|M'}$ to denote the projection of R over the events in M'. We also use the notation $G.M_\tau$ to denote the set of events in $G.M$ of session τ, and use $G.M_{\mathsf{T}}$ to denote set of events in $G.M$ of memory event type T ($\mathsf{T} \in \{\mathsf{R}, \mathsf{W}, \mathsf{U}, \mathsf{F}\}$). We use $G.M_{\mathsf{W+U}}$ to indicate the set of all write and update events. $G.M_{\sqsubseteq \mathsf{X}}$ is used to denote the set of all events in $G.M$ whose access mode is related by the partial order $\sqsubseteq$, with X. $G.M_\ell$ for location ℓ denotes the set of memory events accessing ℓ. We can also combine these notations to denote subsets of events satisfying multiple properties, for example, $G.M_{\mathsf{W},\ell}$ denotes write events to location ℓ. We use $G.w_\ell^{\max}$ to denote the most recent write (or update) event in $G.M$ to ℓ according to the $G.\mathsf{mo}$ order. Given a memory event $\mu = \mathsf{Mem}(i, \tau, \alpha)$, we use various projection functions $\mathsf{sess}(\mu), \mathsf{type}(\mu), \mathsf{loc}(\mu), \mathsf{rval}(\mu), \mathsf{wval}(\mu)$, $\mathsf{mod}(\mu)$ to denote the session, memory event type, memory location, read value, write value, access mode respectively. Similarly, for an invocation event γ, we use functions $\mathsf{method}(\gamma), \mathsf{arg}(\gamma), \mathsf{ret}(\gamma)$ to denote the method name, argument, and return value resp. Given a relation R between memory events, we use the notation $\mu_1 \xrightarrow{R} \mu_2$ to denote $R(\mu_1, \mu_2)$ for memory events μ_1, μ_2.

Execution Graphs. A valid execution graph $G = \langle M, \mathsf{rf}, \mathsf{mo}, \mathsf{so}\rangle$ obeys certain well-formedness constraints, irrespective of the memory model. These well-formedness constraints follow directly from the definition of the various relations and include the following: (i) $\mathsf{mo} \subseteq M_{\mathsf{W+U}} \times M_{\mathsf{W+U}}$ is a total order between write/update events to the same location, (ii) so is a total order between events in the same session, (iii) $\mathsf{rf} \subseteq M_{\mathsf{W+U}} \times M_{\mathsf{U+R}}$ is total on its co-domain and only relates events to the same location and (iv) if $\mu_1 \xrightarrow{\mathsf{rf}} \mu_2$, then $\mathsf{wval}(\mu_1) = \mathsf{rval}(\mu_2)$. We denote these constraints by χ_{base}. In order to understand how execution graphs are constrained by memory models, we first describe the specification for SC. Given an execution graph $G = \langle M, \mathsf{rf}, \mathsf{mo}, \mathsf{so}\rangle$, we define the following derived relations:

$$G.\mathsf{fr} = (G.\mathsf{rf}^{-1}; G.\mathsf{mo}) \setminus [G.M] \qquad G.\mathsf{sw} = [G.M_{\sqsupseteq \mathsf{rel}}]; ([G.M_{\mathsf{F}}]; G.\mathsf{so})^?; G.\mathsf{rf}^+; (G.\mathsf{so}; [G.M_{\mathsf{F}}])^?; [G.M_{\sqsupseteq \mathsf{acq}}]$$

$$G.\mathsf{hb}_{\mathsf{SC}} = (G.\mathsf{rf} \cup G.\mathsf{mo} \cup G.\mathsf{fr} \cup G.\mathsf{so})^+ \qquad G.\mathsf{hb} = (G.\mathsf{sw} \cup G.\mathsf{so})^+$$

The fr relation relates a read event with write events that it does not witness. Execution graph G is said to be *SC-consistent* if $\mathsf{hb}_{\mathsf{SC}}$ is irreflexive. The $\mathsf{hb}_{\mathsf{SC}}$ ordering is an interleaving of events across sessions; making it irreflexive ensures that each read event reads from the most recent write event, formalizing our intuitive understanding of SC executions. We denote this constraint by $\chi_{\mathsf{sc}}(G)$.

The sw relation relates `rel` writes with `acq` reads that read from them, as well as `rel` and `acq` fences that have rf-related write and read events between them; hb then transitively expands sw while also taking into account so. An execution graph G is said to be *RC20-consistent* if the following conditions hold: (1) $G.\mathsf{fr}; (G.\mathsf{rf})^?; G.\mathsf{hb}$ is irreflexive; (2) $G.\mathsf{mo}; (G.\mathsf{rf})^?; (G.\mathsf{hb})^?$ is irreflexive; (3) $G.\mathsf{fr}; G.\mathsf{mo}$ is irreflexive; and, (4) $G.\mathsf{rf} \cup G.\mathsf{so}$ is acyclic.

We denote this set of constraints by $\chi_{\mathsf{RC20}}(G)$. Intuitively, the first two irreflexivity constraints guarantee that read and write operations obey hb ordering, i.e. a read event cannot overlook a write

event that *happens-before* the read, and the modification order between two write events must agree with the *happens-before* order. The third irreflexivity constraint guarantees CAS semantics, i.e., two distinct update events cannot read from the same write event. The last constraint is required to prohibit out-of-thin-air reads [9]. Notice that $\chi_{\text{SC}}(G) \Rightarrow \chi_{\text{RC20}}(G)$, and hence, if an execution graph is SC-consistent, it is also RC20-consistent. The above specification of RC20-consistency matches with the development used in previous work [43].

Traces. We now present the memory systems for the SC and RC20 memory models. We define a common parameterized system that can be instantiated with either χ_{SC} or χ_{RC20}. The labeled transition system for memory model X is given by $\mathcal{MS}_X = \langle \mathcal{G}, \mathsf{MemEvt}, \rightarrow_X \rangle$. Here, $\mathcal{G}$ is the set of all execution graphs, MemEvt are memory events as defined earlier, and $\rightarrow_X \subseteq \mathcal{G} \times \mathsf{MemEvt} \times \mathcal{G}$ are labeled transitions. We define an initial execution graph $G_\perp = \langle M_\perp, \emptyset, \emptyset, \emptyset \rangle$, where $M_\perp$ contains an initial write event $\mathtt{W}\langle\, \ell, v_\ell^\perp, \mathtt{rlx} \,\rangle$ to every location ℓ.

Each transition adds a new memory event, corresponding to an arbitrary action α in arbitrary session τ, as long as the new execution graph obeys the well-formedness constraints and the consistency constraints of the memory model. We now take the product of the program semantics ($\Omega_{\mathcal{L}}$) and memory system ($\mathcal{MS}_X$) to describe executions of a library implementation $\mathcal{L}$ under memory model X. In the product, the transition of the program semantics $\stackrel{\mu}{\Longrightarrow}$ and the transition of the memory system $\xrightarrow{\mu}_X$ must agree on the memory event. For method invocations, returns, or silent transitions of the program semantics, there will be no transitions in the memory system. A *trace* of the combined transition system begins from the initial state $\langle s_\perp, G_\perp \rangle$ and contains a finite number of transitions. A *complete trace* must end in a final state $\langle s_\perp, G_\perp \rangle \stackrel{\omega_1}{\Longrightarrow} \ldots \stackrel{\omega_n}{\Longrightarrow} \langle s, G \rangle$, where in the state s, all invocations have completed their executions, i.e. there are no pending `return` statements in any session.

In the following, we consider an execution of a library implementation L under memory model X to be a tuple $E = \langle t, \Gamma, G \rangle$ where t is a trace of the combined transition system $\Omega_L \times \mathcal{MS}_X$, Γ is the set of all method invocation events and G is the execution graph in the final state of the trace. We also define an invocation session order so_{inv}, which relates invocation events belonging to the same session in the order in which they appear in the trace t. We use the notation $\mathcal{E}_X^L$ to denote the set of all such executions. Similarly, a complete execution corresponds to a complete trace, and we use $\mathcal{CE}_X^L$ to denote the set of all complete executions. We also use the notation $E.G$ to denote the execution graph of E, and $E.M, E.\mathsf{rf}, E.\mathsf{mo}, E.\mathsf{so}$ to denote the various components of the execution graph G.

3.3 Robustness and Library Correctness under SC

The robustness problem, in general, asks whether every RC20 execution of a program is also possible under SC. In the context of library implementations, we re-define the notion of execution-graph robustness, originally proposed in [38], as follows:

Definition 3.1. An execution $E \in \mathcal{E}_{\text{RC20}}^L$ of a library implementation L is *execution-graph robust*, if $E \in \mathcal{E}_{\text{SC}}^L$. A library implementation L is execution-graph robust if all of its complete executions $E \in \mathcal{CE}_{\text{RC20}}^L$ are execution-graph robust.

Our verification strategy (described in §4) uses the SC specification of a library implementation to discharge the verification conditions. The correctness of a library implementation under SC is typically specified using the notion of linearizability, which tries to establish a simulation between concurrent executions of the implementation and sequential executions of a reference implementation of the same data structure. However, previous works ([23, 25]) have also proposed

alternative declarative specifications which are equivalent to linearizability and which directly constrain the argument/return value of invocations using a collection of axioms. Such a declarative specification is more amenable to SMT encoding, and hence we have used such specifications in our work. The specifications also establish a *happens-before* ordering among invocations ($\mathsf{hb}_{\mathsf{inv}}$) which is required to be a total order.

For example, to provide a declarative specification of the register library of Fig. 2, we first define a binary predicate match over invocation events in an execution, which is used in the following three specification predicates. These take as input the invocation events Γ in an execution E of the library implementation:

$$
\begin{array}{lcl}
\mathsf{match}(\gamma_1, \gamma_2) & \triangleq & \mathsf{method}(\gamma_1) = \mathtt{set} \wedge \mathsf{method}(\gamma_2) = \mathtt{get} \wedge \mathsf{arg}(\gamma_1) = \mathsf{ret}(\gamma_2) \\
\chi_{\textsc{GetSet}}(\Gamma) & \triangleq & \forall \gamma \in \Gamma.\ \mathsf{method}(\gamma) = \mathtt{get} \wedge \mathsf{ret}(\gamma) \neq \mathtt{UNDEF} \Rightarrow \exists \gamma' \in \Gamma.\ \mathsf{match}(\gamma', \gamma) \wedge \\
 & & \gamma' \xrightarrow{\mathsf{hb}_{\mathsf{inv}}} \gamma \\
\chi_{\textsc{GetFrom}}(\Gamma) & \triangleq & \forall \gamma_1, \gamma_2, \gamma_3 \in \Gamma.\ \neg(\mathsf{match}(\gamma_1, \gamma_2) \wedge \mathsf{method}(\gamma_3) = \mathtt{set} \wedge \gamma_1 \xrightarrow{\mathsf{hb}_{\mathsf{inv}}} \gamma_3 \wedge \gamma_3 \xrightarrow{\mathsf{hb}_{\mathsf{inv}}} \gamma_2) \\
\chi_{\textsc{GetUNDEF}}(\Gamma) & \triangleq & \forall \gamma_1, \gamma_2 \in \Gamma.\ \mathsf{method}(\gamma_1) = \mathtt{set} \wedge \mathsf{method}(\gamma_2) = \mathtt{get} \wedge \gamma_1 \xrightarrow{\mathsf{hb}_{\mathsf{inv}}} \gamma_2 \\
 & & \Rightarrow \mathsf{ret}(\gamma_2) \neq \mathtt{UNDEF} \\
\chi_{\textsc{LIN}} & \triangleq & (\mathsf{so}_{\mathsf{inv}} \Rightarrow \mathsf{hb}_{\mathsf{inv}}) \wedge \mathsf{hb}_{\mathsf{inv}} \text{ is a total order}
\end{array}
$$

$\chi_{\textsc{GetSet}}$ constrains all get invocations that return non-UNDEF values to match their return values with the argument value of some set invocation (and also defines $\mathsf{hb}_{\mathsf{inv}}$ relation between them), $\chi_{\textsc{GetFrom}}$ disallows scenarios in which a get invocation returns the value of an older (not most-recent) set operation, while $\chi_{\textsc{GetUNDEF}}$ disallows a get operation to return UNDEF if there is set operation before it. $\chi_{\textsc{LIN}}$ ensures that $\mathsf{hb}_{\mathsf{inv}}$ is a total order and obeys the session order among invocations. The set $\chi_{\textsc{REG}} = \{\chi_{\textsc{GetSet}}, \chi_{\textsc{GetFrom}}, \chi_{\textsc{GetUNDEF}}, \chi_{\textsc{LIN}}\}$ defines a specification of the register library equivalent to linearizability[25]. Under SC, it is easy to see that any execution of the implementation of Fig. 2 satisfies $\chi_{\textsc{REG}}$.

In general, given a library implementation $\mathcal{L}$ of data structure $\mathcal{D}$ with specification $\chi_{\mathcal{D}}$, we assume library correctness under SC, which means that every complete execution $E = \langle t, \Gamma, G \rangle \in \mathcal{CE}^{L}_{\mathrm{SC}}$ satisfies the specification, i.e. $\mathcal{A}_i(\Gamma)$ holds for all $\mathcal{A}_i \in \chi_{\mathcal{D}}$. Declarative specifications equivalent to linearizability have been defined for all common data structures such as stack, queue, set, etc. in previous works [25].These specifications can be easily encoded as FOL formulae over the domain of invocation events, constraining the method names, arguments, return value and the session order relation. Note that specifications do not directly constrain any internal event of the library implementation, since these would not be directly observable to a client, and the specification should be agnostic of the implementation.

4 Induction for Robustness

In this section, we adapt an existing approach [43] to checking robustness of programs under the RC20 memory model to the library setting, and in particular derive an inductive strategy for establishing robustness. Our strategy is based on deriving sufficient conditions for robustness, such that if these conditions are maintained at every step in every SC execution of the implementation, then it is guaranteed to be execution-graph robust. In order to describe this robustness condition, we first define the notion of prefix of executions.

Definition 4.1. Given a complete execution $E = \langle t, \Gamma, G \rangle \in \mathcal{CE}^{L}_{\mathrm{SC}}$ of implementation L under SC, a prefix $E' = \langle t', \Gamma', G' \rangle \in \mathcal{E}^{L}_{\mathrm{SC}}$ is an execution obtained from a prefix t' of the trace t. That is, there exists t'' such that $t = t' \circ t''$ where $\circ$ composes two traces if the final state in t' is the same as the initial state of t''.

Note that for every relation R defined in the execution graph G, the relation R' in the execution graph G' will be $R_{|G'.M}$. That is because for events $e_1, e_2 \in G'.M$, if $e_1 \xrightarrow{R} e_2$, then since both events also occur in the trace t', we would also have $e_1 \xrightarrow{R'} e_2$. The prefix E' also obeys the following property:

$$\forall \mu' \in E'.M.\ \forall \mu \in E.M.\ \mu \xrightarrow{E.\mathsf{so}} \mu' \Rightarrow \mu \in E'.M$$

That is, if an event μ' is in the prefix, then every event μ before it in the same session must also be in the prefix. We now define the next event of a prefix:

Definition 4.2. Given a prefix $E' = \langle t', \Gamma', G' \rangle$ of E, μ is called the next event of E' if $t' \overset{\omega_1}{\Longrightarrow} C_1 \overset{\omega_2}{\Longrightarrow} C_2 \Rightarrow \ldots \overset{\omega_{n-1}}{\Longrightarrow} C_{n-1} \overset{\mu}{\Rightarrow} C_n$ is also a prefix of t, where $\omega_1, \ldots, \omega_{n-1}$ are either invocation events or local events and μ is a memory event.

The next event of prefix E' is denoted by $\mathsf{next}(E', E)$. For prefix E' and $\mu = \mathsf{next}(E', E)$, $E' + \mu$ is used to denote the execution corresponding to the prefix of E ending in event μ. The following definition characterizes a non-robustness witness[43], i.e. a SC execution which leads to a non-robust RC20 execution.

Definition 4.3. Given a complete SC execution $E \in \mathcal{CE}^L_{\mathsf{SC}}$ of implementation L, let E' be a prefix of E and event $\mu = \mathsf{next}(E', E)$ such that $\mathsf{loc}(\mu) = \ell$ and $\mathsf{sess}(\mu) = \tau$. Then E' is a non-robustness witness if the following conditions are true:[2]

(1) $E'.w^{\mathsf{max}}_\ell \in dom(E'.\mathsf{hb}^?_{\mathsf{SC}}; [E'.M_\tau])$
(2) There exists $\mu_w \in E'.M_{\mathsf{W},\ell} \cup E'.M_{\mathsf{U},\ell}$ such that $\mu_w \neq E'.w^{\mathsf{max}}_\ell$, $\mu_w \notin dom(E'.\mathsf{mo}; E'.\mathsf{rf}^?; E'.\mathsf{hb}^?; [E'.M_\tau])$ and if $\mathsf{act}(\mu) \neq \mathsf{R}$ then $\mu_w \notin dom(E'.\mathsf{rf}; [E'.M_{\mathsf{U}}]))$ and

The non-robustness witness definition formalizes our intuitive understanding of a non-robust execution: for a prefix E', the next event μ can be non-robust (i.e. can have a different behavior under RC20 and SC memory models) if the most recent write event w^{max}_ℓ to ℓ occurs $\mathsf{hb}_{\mathsf{SC}}$-before the session containing μ, so under SC, if μ is a read event, then it must read from w^{max}_ℓ. But w^{max}_ℓ does not occur hb-before the session containing μ (because of the presence of the earlier write event μ_w), so that in a RC20 execution, μ can read from μ_w. Note that if μ is not a read event, then the definition further constrains the earlier write event μ_w to not be related by rf to an update event, since the CAS semantics even under the C11 memory model would not allow another write (in this case μ) to come between the write (μ_w) from which an update events reads from. In essence, referring back to the memory system defined in Sec. 3.2, $E'.G \xrightarrow{\mu}_{\mathsf{RC20}} E'.G + \mu$ would be a valid transition in the RC20 memory model, but this would not be a valid transition in the SC system.

Further, the execution leading up to such a non-robustness witness must obey SC semantics, thus characterizing the first (or minimal) violation of robustness. If we can ensure such a non-robustness witness does not arise during any SC execution, the library implementation is guaranteed to be robust.

Theorem 4.4. *Given a library implementation L, if every prefix E' of every complete execution $E \in \mathcal{CE}^L_{\mathsf{SC}}$ is not a non-robustness witness, then L is execution-graph robust*[3].

The above theorem is a direct application of Theorem 4.6 in [43] but applied to executions of a library implementation in the presence of the most general client. In order to illustrate this idea,

[2] Adapted from Definition 4.5 in [43]
[3] All proofs can be found in appendix, §B

we revisit the execution in Figure 3. Notice that just before the event $\mathtt{R}\langle \mathtt{N} \rightarrow \mathtt{val}, \mathtt{v}, \mathtt{rlx} \rangle$, the most recent write event to the location $\mathtt{N}\rightarrow\mathtt{val}$, $\mathtt{W}\langle \mathtt{N} \rightarrow \mathtt{val}, \mathtt{v}, \mathtt{rlx} \rangle$, is in both $\mathsf{hb}_{\mathsf{SC}}$ and hb order, thus ensuring that the read event has the same behavior under both SC and RC20. On the other hand, suppose we modified the library implementation and changed the access modes of either the load or store of L to be rlx. Then, we would get a non-robustness witness E' consisting of the two events of the set invocation (on the left) and the first event of the get invocation ($\mathtt{R}\langle \mathtt{L}, \mathtt{N}, \mathtt{rlx} \rangle$), with the next event being the read of $\mathtt{N} \rightarrow \mathtt{val}$. This witness can be obtained through an SC execution, in the sense that it does not violate χ_{SC}, and it obeys both the conditions of Definition 4.3. In particular, with the next event to E' being a read to location $\mathtt{N} \rightarrow \mathtt{val}$, the most recent write event $w^{\mathsf{max}}_{\mathtt{N}\rightarrow\mathtt{val}}$ is $\mathtt{W}\langle \mathtt{N} \rightarrow \mathtt{val}, \mathtt{v}, \mathtt{rlx} \rangle$ and it is in $\mathsf{hb}_{\mathsf{SC}}$ order to the session on the right (thus obeying condition-1), but it is not in hb order to it. As a result, the initializing write event to $\mathtt{N} \rightarrow \mathtt{val}$ (not depicted in the figure) would take the role of the event μ_w in condition-2 of Definition 4.3. Hence, relaxing the access mode of either of the accesses to the location L renders the library implementation non-robust.

While Definition 4.3 simplifies our task of proving robustness of executions by reducing it to a search problem over SC executions, this is still a tall order, as it requires maintaining information about relations between low-level events across an unbounded number of arbitrarily long executions involving an unbounded number of heap locations. To address this issue, our verification strategy uses induction on the prefixes of SC executions, and then shows the absence of a non-robustness witness at every step of every SC execution. We can try to use the non-existence of a non-robustness witness itself as an inductive invariant. That is, for an SC execution E, we consider a prefix E' and assume inductively that for $\mu = \mathsf{next}(E', E)$, E' and μ do not form a non-robustness witness, and then prove that for $\mu' = \mathsf{next}(E' + \mu, E)$, $E' + \mu$ and μ' do not form a non-robustness witness. Notice from Def. 4.3 that E' and μ not forming a non-robustness witness would mean $\neg((1) \wedge (2)) \equiv (1) \Rightarrow \neg(2)$, which essentially means that if $E'.w^{\mathsf{max}}_{\ell}$ is in $\mathsf{hb}_{\mathsf{SC}}$ order before session τ, then it should also be in hb order before τ.

Unfortunately, we find that this does not constitute an appropriate inductive invariant, since it only constrains the events that access location ℓ, but the next event of the new prefix $E' + \mu$ may access a different location after this event is added. Hence, the inductive hypothesis involving ℓ is not useful. We could try to use as inductive invariant the required property for all locations: i.e. $\forall \ell$, if $E'.w^{\mathsf{max}}_{\ell}$ is in $\mathsf{hb}_{\mathsf{SC}}$ order before some session τ, it is in hb order before it. However, this condition is too strong and not necessary for establishing non-robustness. This is because even if this condition does not hold for some location (say ℓ'), which means that $E'.w'^{\mathsf{max}}_{\ell}$ is in $\mathsf{hb}_{\mathsf{SC}}$ order before the session τ but not in hb order before it, the next event in τ must access ℓ' to form the actual non-robustness witness. The presence of such a event $E'.w'^{\mathsf{max}}_{\ell}$ is a necessary condition for forming an actual non-robustness witness, which we formalize below as a potential witness:

Definition 4.5. Given an execution $E \in \mathcal{CE}^{L}_{\mathsf{SC}}$, a prefix E' of E, $\mu \in E'.M$ and a write event $\mu' \in E'.M^{\mathsf{max}}_{\mathtt{W}} \cup E'.M^{\mathsf{max}}_{\mathtt{U}}$, E', μ' and μ form a potential non-robustness witness if $\mu' \xrightarrow{E'.\mathsf{hb}_{\mathsf{SC}}} \mu$ and $\neg(\mu' \xrightarrow{E'.\mathsf{hb}} \mu)$.

$E.M^{\mathsf{max}}_{\mathtt{W}}$ indicates the set of maximal events in E according to the mo ordering. Note that to obtain an actual non-robustness witness, we would have to instantiate the above definition with $\mu' = w^{\mathsf{max}}_{\ell}$ for some location ℓ, and the next event in the session containing μ must also access ℓ. Figure 7 shows a potential non-robustness witness for the register library implementation. It shows an execution with two set invocations which happen in different sessions. Because of the mo relation between the two write events to location L, the write event $\mathtt{W}\langle \mathtt{N}_1 \rightarrow \mathtt{val}, \mathtt{v}_1, \mathtt{rlx} \rangle$ comes in $\mathsf{hb}_{\mathsf{SC}}$ order to the last event of session τ_2 ($\mathtt{W}\langle \mathtt{L}, \mathtt{N}_2, \mathtt{rel} \rangle$), but it is not in hb order before

it. Using the notation of the above definition, E' would be the execution depicted in Figure 7, μ' would be the write event to $\mathtt{N}_1 \rightarrow \mathtt{val}$, and μ would be the write event to $\mathtt{L}$ in session τ_2. A potential non-robustness witness, consisting of prefix E' and the write event μ' can become an actual non-robustness witness if E' can be expanded into another prefix E'' such that the next event μ'' of E'' accesses the location of μ', μ' continues to remain the most recent write to its location, and the non-hb $\mathsf{hb}_{\mathsf{SC}}$ relation continues to exist between μ_w and μ''. For the potential non-robustness witness of Fig. 7, a subsequent immediate access to location $\mathtt{N}_1 \rightarrow \mathtt{val}$ would result in an actual non-robustness witness. While every potential witness does not necessarily become an actual witness, an actual witness must have been at some stage a potential witness. For example, the potential non-robustness witness in our example will never result in an actual witness, since there will be no further access to the location $\mathtt{N}_1 \rightarrow \mathtt{val}$ in an SC execution. Intuitively, this is because of the register semantics, as the effect of a `set` invocation γ_2 "overwrites" the effects of a previous `set` invocation γ_1 through the write to location $\mathtt{L}$, and the $\chi_{\textsc{GetFrom}}$ specification prohibits a `get` invocation to read from an older `set` invocation.

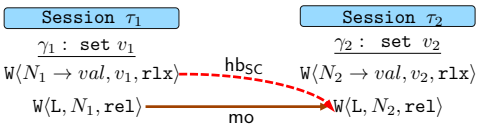

Fig. 7. Potential non-robustness witness in the register library implementation.

Table 1. Sufficient conditions for establishing robustness inductively.

$\Phi_{src}(E,\ell)$	$\forall \mu_m \in E.M_{\mathsf{W}}^{\max} \cup E.M_{\mathsf{U}}^{\max}.\ \forall \mu_1, \mu_2 \in E.M.\ \mathsf{loc}(\mu_m) = \ell \wedge$ $\wedge \mu_m \xrightarrow{E.\mathsf{hb}^?} \mu_1 \wedge \mu_1 \xrightarrow{E.\mathsf{fr} \vee E.\mathsf{mo} \vee E.\mathsf{rf}} \mu_2 \Rightarrow \exists \mu_3 \in E.M.\ \mu_m \xrightarrow{E.\mathsf{hb}} \mu_3 \wedge \mu_3 \xrightarrow{E.\mathsf{hb}^?} \mu_2$
$\Phi_{dst}(E,\ell)$	$\forall \mu_m \in E.M_{\mathsf{W}}^{\max} \cup E.M_{\mathsf{U}}^{\max}.\ \forall \mu_1, \mu_2 \in E.M.\ \mathsf{loc}(\mu_m) = \ell \wedge \mathsf{loc}(\mu_2) = \ell$ $\wedge \mu_m \xrightarrow{E.\mathsf{hb}_{\mathsf{SC}}} \mu_1 \wedge \mu_1 \xrightarrow{E.\mathsf{so}} \mu_2 \Rightarrow \exists \mu_3 \in E.M.\ \mu_m \xrightarrow{E.\mathsf{hb}} \mu_3 \wedge \mu_3 \xrightarrow{E.\mathsf{so}} \mu_2$

In order to verify robustness, we then have to ensure that either a potential non-robustness witness is not formed at all (thus ensuring robustness at the source), or if a potential robustness witness does form, it does not manifest into an actual non-robustness witness (thus ensuring robustness at the destination). As shown in Table 1, we present two conditions (Φ_{src} and Φ_{dst}) that ensure the absence of a non-robustness witness by disallowing it at the source and destination, resp. These conditions are parametric on executions and locations.

We first focus on Φ_{src}, which inductively tries to maintain the non-existence of a potential non-robustness witness. In order to understand $\Phi_{src}(E,\ell)$, we note that the only difference between the definitions of hb and $\mathsf{hb}_{\mathsf{SC}}$ is that the rf relation between relaxed accesses as well as fr and mo relations between any two accesses induce an $\mathsf{hb}_{\mathsf{SC}}$ relation, but they do not induce an hb relation. Hence, whenever $\mathsf{hb}_{\mathsf{SC}}$ is established due to these relations, we should also be able to establish hb. Φ_{src} considers a maximal write event μ_w to location ℓ in E. If there exists a $\mathsf{hb}_{\mathsf{SC}}$ (and hence inductively hb) relation from μ_w to some event μ_1, and if this $\mathsf{hb}_{\mathsf{SC}}$ relation were to be extended to μ_2 by a fr, mo or rf relation between μ_1 and μ_2, then there should also exist a hb relation from μ_w to μ_2, through some event μ_3. To illustrate, we refer back to Figure 4, which essentially shows the different cases to be handled in order to establish $\Phi_{src}(E,\mathtt{L})$ for every SC execution E of the register implementation. In particular, for the case depicted in Figure 4(a), using the notation of Φ_{src}, $\mu_w = \mu_1 = \mathtt{W}\langle \mathtt{L}, \_, \mathtt{rel} \rangle$ and $\mu_2 = \mu_3 = \mathtt{R}\langle \mathtt{L}, \_, \mathtt{acq} \rangle$. Cases depicted in Figure 4(b)-(d) do not satisfy the antecedent of Φ_{src} (because there does not exist events μ_1, μ_2 with the required relation between them), while for the case depicted in Figure 4(e), $\mu_1 = \mathtt{W}\langle \mathtt{N} \rightarrow \mathtt{val}, \_, \_ \rangle$, $\mu_2 = \mathtt{R}\langle \mathtt{N} \rightarrow \mathtt{val}, \_, \_ \rangle$,

and $\mu_3 = \mathsf{R}\langle \mathsf{L}, \mathsf{N}, \mathsf{acq} \rangle$; μ_3 is not depicted in the figure, but it would occur before μ_2 in the same `get` invocation. Note that the antecedent of Φ_{src} simply means that $\mu_w \xrightarrow{\mathsf{hb}} \mu_2$, but explicitly maintaining the event μ_3 which establishes this hb relation helps us in the SMT encoding of Φ_{src}, as we will explain in the next section.

Now, if Φ_{src} cannot be established, then a potential non-robustness witness may be formed, as illustrated in Fig. 7. Notice that this means that $\Phi_{src}(E, \mathsf{N}_1 \rightarrow \mathtt{val})$ does not hold for the execution E of Fig. 7. Then, we use the condition Φ_{dst} to prevent such a potential non-robustness witness from turning into an actual witness. In words, this condition considers a scenario when there is a maximal write event μ_w^{max} to location ℓ which occurs $\mathsf{hb}_{\mathsf{SC}}$-before event μ_1, with a later event μ_2 in the same session accessing the location ℓ. This could lead to an actual non-robustness witness, with the prefix being the execution corresponding to the trace leading upto the state just before μ_2. However, in such a scenario, Φ_{dst} enforces the existence of another event μ_3 before μ_2 (μ_3 could be the same as μ_1) in the same session, such that μ_w^{max} occurs hb-before μ_3. This would prevent the formation of an actual non-robustness witness involving μ_w^{max} and μ_2.

To illustrate, we have already established through Fig. 7 that Φ_{src} does not hold for the location $\mathsf{N}_1 \rightarrow \mathtt{val}$. To show Φ_{dst}, we let μ_w be $\mathsf{W}\langle \mathsf{N}_1 \rightarrow \mathtt{val}, \mathsf{v}_1, \mathtt{rlx} \rangle$, while μ_1 would be some event which occurs in so-order before μ_2, which has to be another access to $\mathsf{N}_1 \rightarrow \mathtt{val}$. As per the program structure of the register library, μ_2 can only be a read event, which must occur in a `get` invocation. The scenario exactly corresponds to Fig. 3, where, using the notation of Φ_{dst}, $\mu_w = \mathsf{W}\langle \mathsf{N} \rightarrow \mathtt{val}, \mathsf{v}, \mathtt{rlx} \rangle$, $\mu_1 = \mathsf{R}\langle \mathsf{L}, \mathsf{N}, \mathsf{acq} \rangle$ and $\mu_2 = \mathsf{R}\langle \mathsf{N} \rightarrow \mathtt{val}, \mathsf{v}, \mathtt{rlx} \rangle$. However, in this case, μ_1 must read from $\mathsf{W}\langle \mathsf{L}, \mathsf{N}, \mathsf{acq} \rangle$, because there is no another write to L which writes N (guaranteed by the `malloc` semantics). Hence, we can instantiate μ_3 with μ_1 itself in the consequent of Φ_{dst}.

Notice that if $\Phi_{src}(E, \ell)$ holds, then $\Phi_{dst}(E, \ell)$ also holds, because $\mathsf{hb}_{\mathsf{SC}}$ would imply hb from write events to ℓ, in which case we can take $\mu_3 = \mu_1$. However, in general, Φ_{dst} does not imply Φ_{src} (as we saw for the location $\mathsf{N}_1 \rightarrow \mathtt{val}$). The reason we separate out the two conditions is because for some locations, it is easier to establish $\Phi_{src}(E, \ell)$, since the condition just requires one access to the location ℓ, as opposed to $\Phi_{dst}(E, \ell)$ which requires two accesses. The distinction will become more clear when we go through our SMT encoding in §7.

To summarize, $\Phi_{src}(E, \ell)$ prevents the formation of a potential non-robustness witness involving location ℓ, while $\Phi_{dst}(E, \ell)$ prevents a potential non-robustness witness involving ℓ from turning to an actual non-robustness witness. If we can show that $\forall \ell \in \mathsf{Location}. \forall E \in \mathcal{CE}^L_{\mathsf{SC}}.\ \Phi_{dst}(E, \ell) \vee \Phi_{src}(E, \ell)$, then we can inductively show execution-graph robustness.

Theorem 4.6. *Given a library implementation L, if $\forall \ell \in \mathsf{Location}. \forall E \in \mathcal{CE}^L_{\mathsf{SC}}.\ \Phi_{dst}(E, \ell) \vee \Phi_{src}(E, \ell)$, then L is execution-graph robust.*

We note that Theorem 4.6 does not hold in the other direction, i.e. Φ_{dst} and Φ_{src} are not necessary for ensuring execution-graph robustness. In particular, there is a gap between the definition of a non-robustness witness (Def. 4.3) and the conditions Φ_{src} and Φ_{dst}. These conditions merely ensure that if the maximal write μ_m to a location appears $\mathsf{hb}_{\mathsf{SC}}$-before the next access to it, it also appear hb-before it. They do not consider the non-maximal write μ_w from Def. 4.3 at all, which is actually the root cause of the non-robust behavior. In particular, consider the scenario where a non-maximal write μ_w may be in rf relation to an update event, and hence cannot cause a robustness violation if the next event (say μ) to be added is a write/update event to the same location as μ_w, and the maximal write μ_m is in $\mathsf{hb}_{\mathsf{SC}}$ order to the session containing μ. In this case, $\mathsf{act}(\mu) \neq \mathsf{R}$, $\mu_w \in dom(E'.\mathsf{rf}; [E'.M]_{\mathsf{U}})$ (here E' is the set of events in the execution, see point(2) in Def. 4.3). Hence, we do not require μ_w and consequently μ_m to be hb-before μ.

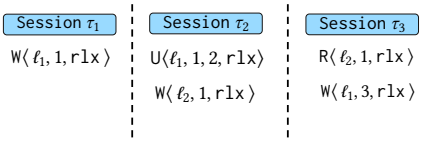

Fig. 8. Execution illustrating incompleteness of Φ_{src} and Φ_{dst}.

Fig. 8 concretely demonstrates this scenario. Assume that all the events in the execution come from some library invocations. The update operation to ℓ_1 in τ_2 reads from the write to ℓ_1 in τ_1. The read in τ_3 to location ℓ_2 reads from the write in τ_2, hence the maximal write to ℓ_1 (which is the update in τ_2) is in $\mathsf{hb}_{\mathsf{SC}}$ order, but not in hb order to τ_3, since the accesses to ℓ_2 are relaxed. Excluding the write to ℓ_1 in τ_3, the execution is a potential non-robustness witness. Let E' be this execution, with the next event $\mu = \mathtt{W}\langle \ell_1, 3, \mathtt{rlx} \rangle$. Following the notation used in Def. 4.3, we have the maximal write to ℓ_1, $\mu_m = \mathtt{U}\langle \ell_1, 1, 2, \mathtt{rlx}, \mathtt{rlx}\rangle$, and a non-maximal write $\mu_w = \mathtt{W}\langle \ell_1, 1, \mathtt{rlx} \rangle$. Now, $E' + \mu$ is not an actual non-robustness witness, because even though μ_m is $\mathsf{hb}_{\mathsf{SC}}$ order and μ_w is not in hb order to the session τ_3, μ_w is in rf order to an update event, and hence μ_w does not satisfy point(2) of Def. 4.3.

We note that except scenarios like the one given above, Φ_{src} and Φ_{dst} precisely model the absence of non-robustness witnesses, i.e. if these conditions are violated, it would imply the presence of an actual non-robustness witness. A distinguishing property of executions like Fig. 8 is that they involve at least one update event (generated through CAS operation) to a location. We have observed that in library implementations, either all writes to a location happen through CAS operations, or none do (i.e. it doesn't happen that there is both a CAS and a normal write operation to the same location). If all accesses to a location are through CASes, our automated verification strategy (given in §7) uses additional CAS constraints, which helps us avoid any false positives (like the execution in Fig. 8) arising out of the imprecision of Φ_{src} and Φ_{dst}.

5 Induced Subgraph Robustness

We now discuss the challenges that arise in using our verification strategy on real-world library implementations, and the mechanisms we employ to overcome them.

We observe that the notion of execution-graph robustness is often too strong for real-world library implementations that use low-level synchronization primitives such as compare-and-swap (CAS). A typical CAS-based synchronization pattern is as follows: a method invocation performs a number of speculative reads without any synchronization, followed by some computation based on these speculative reads, and then finally stores the result of this computation to the global state if no other concurrent method invocation has made conflicting changes. The pattern uses CAS operations to determine if a conflict exists. If another concurrent invocation has indeed made changes to the global state, the speculative computation is ignored (since the corresponding CAS would fail), and the pattern is retried. These speculative computations are often performed using relaxed accesses, and they may generate an arbitrary number of read/write events. Any non-robust behavior of such events would result in a failing CAS, in effect restarting the entire invocation. It is clearly ineffective to establish the robustness of events that are generated by such failed computation. A concrete example of a library implementation demonstrating this pattern is given in the supplemental material, §4.

In particular, an RC20 execution of a library implementation may be still be effectively robust against SC even if it exhibits potentially non-robust actions if all events that actually affect the return value of any method invocation obey SC semantics. To capture this distinction, we define a new notion of robustness called *induced subgraph robustness* that only focuses on the robustness of events that actually effect the client observable behavior of an invocation. Given an execution graph $G = \langle M, \mathsf{rf}, \mathsf{mo}, \mathsf{so}\rangle$, $G' = \langle M', \mathsf{rf}', \mathsf{mo}', \mathsf{so}'\rangle$ is called an induced subgraph of G if $M' \subseteq M$ and $\mathsf{rf}' = \mathsf{rf}_{|M'}$, $\mathsf{mo}' = \mathsf{mo}_{|M'}$, $\mathsf{so}' = \mathsf{so}_{|M'}$.

Definition 5.1. A complete execution $E = \langle t, \Gamma, G \rangle \in \mathcal{CE}^{L}_{\text{RC20}}$ of a library implementation L is *induced subgraph robust* if there exists a complete execution $E' = \langle t', \Gamma', G' \rangle \in \mathcal{CE}^{L}_{\text{SC}}$ such that (i) $\Gamma' = \Gamma$, (ii) For all invocation events $\gamma_1, \gamma_2 \in \Gamma$, γ_1 occurs before γ_2 in trace $t \Leftrightarrow \gamma_1$ occurs before γ_2 in trace t' and (iii) G' is an induced subgraph of G . L is *induced subgraph robust* if all of its complete executions $E \in \mathcal{CE}^{L}_{\text{RC20}}$ are induced subgraph robust.

Notice that induced subgraph robustness only considers complete executions of the library implementation. For every RC20 execution, there must exist another complete execution under SC such that the behavior of the method invocations (i.e. argument and return values) and the memory events which are responsible for this behavior remains the same as the original RC20 execution. The latter is captured by requiring the execution graph G' of the SC execution to be an induced subgraph of the RC20 execution. Referring back to the CAS based pattern used in libraries, the motivation behind requiring G' to be an induced subgraph of G is that G' would presumably only contain the events of G corresponding to the computation which resulted in a successful CAS, while the remaining events in G corresponding to failing speculative computation can be ignored. This is ensured by the fact that the behavior of the method invocation events remains the same. The client of the library need not care about the non-robustness of events which do not affect the observable behavior.

While induced subgraph robustness is a more useful correctness criterion for real-world library implementations, our verification strategy of the previous section aims to show execution-graph robustness. In order to connect these two notions, we consider a pre-processing step that transforms the implementation so that it does not generate those events which have no impact on an invocation's return value. Our goal is to ensure that all events remaining in the transformed program will have an effect on a method's return values. Induced subgraph robustness of the original implementation is then reduced to checking execution-graph robustness of the new implementation. Formally, we define a robustness-preserving transformation as follows:

Definition 5.2. A *robustness-preserving* program transformation is a function ρ that takes as input a library implementation L and outputs another library implementation $\rho(L)$ that obeys the following two conditions:

(1) For every complete RC20 execution $E = \langle t, \Gamma, G \rangle \in \mathcal{CE}^{L}_{\text{RC20}}$ of L, there exists a complete RC20 execution $E' = \langle t', \Gamma', G' \rangle \in \mathcal{CE}^{\rho(L)}_{\text{RC20}}$ of $\rho(L)$ such that (i) $\Gamma' = \Gamma$, (ii) the order of invocation events is the same in both the traces t and t' and (iii) G' is an induced subgraph of G.
(2) $\mathcal{CE}^{\rho(L)}_{\text{SC}} \subseteq \mathcal{CE}^{L}_{\text{SC}}$.

The first condition in the above definition allows executions of $\rho(L)$ to not contain all the events in executions of L, as long as the observable behavior of invocation events remains the same. The second condition ensures that the transformation does not add any new behaviors. If we can show that the transformed implementation $\rho(L)$ is execution graph robust, then this would imply that the original implementation is induced subgraph robust. We can directly apply our induction strategy of Section 4 on $\rho(L)$ to determine its execution-graph robustness.

Theorem 5.3. *Given a library implementation L and a robustness-preserving transformation ρ, if $\rho(L)$ is execution-graph robust, then L is induced subgraph robust.*

While checking whether a transformation is robustness preserving would be hard in general–since it requires comparing all RC20 executions of two implementations–applying the definition to the CAS based synchronization pattern used in libraries is quite straightforward. In particular, we consider the transformation ρ_{CAS}, which focuses on loops whose loop condition is based on the

success of a cas instruction, and which only generates read events in iterations where the CAS fails. In such cases, only the last iteration of the loop where the cas succeeds actually matters, and all previous iterations generate benign read events whose robustness can be ignored. The ρ_{CAS} transformation is implemented as a simple syntactic analysis in the following manner: For every while loop whose loop constraint is a cas operation, we first check that only read operations are performed in any iteration. Following that, we unroll the loop once in the original implementation, and check for any dependencies (through local variables) from read operations in the first iteration to write operations, return value of the method, or any operation in the second iteration. If there are no such dependencies, we conclude that the loop iterations are independent, and only the events in the last iteration need to be preserved. In this case, we remove the while loop, and replace the cas operation with a bcas. In such cases, it eliminates the loop, and replaces the cas with a blocking cas (bcas) with the same parameters [38]. The bcas operation blocks until its compare is successful, ensuring that only the events in the last iteration of the loop in the original implementation will be generated.

Lemma 5.4. *The transformation ρ_{CAS} is a robustness-preserving transformation.*

Intuitively, for any RC20 execution E of the original implementation L, we can construct a RC20 execution E' of $\rho_{CAS}(L)$ because all the write operations in E can be directly replicated in E', since the only events that will occur in E but not E' will be read events who do not have any dependences on the write events. Even though these read events can potentially induce more hb relations in E, this only restricts the behavior of other events which are preserved from E to E', and hence this behavior can be replicated in E' where $E'.\mathsf{hb} \subseteq E.\mathsf{hb}$.

Finally, we note that the notion of induced subgraph robustness is closely related to the previously proposed notion of observational robustness [43], which also allows benign non-robust events without any outgoing dependencies. The major difference is that we explicitly maintain the induced subgraph property, and the fact that invocation events retain the same behavior. Further, in heap manipulating programs, it often happens that the location to be read by a subsequent read operation depends on the value read from a previous read (as in the `get` implementation in the register library). In such a scenario, observational robustness would then enforce robustness of the earlier read (because there is an outgoing dependency), but in our case, induced subgraph robustness may allow both reads to be non-robust if there is no outgoing dependency to the return value/global state.

6 Compositionality

As discussed in the introduction, individual library robustness as defined in Def. 3.1 or Def. 5.1 is not sufficient for establishing robustness of executions involving multiple libraries. The issue is that robustness of a library itself does not provide enough synchronization guarantees that would be required to establish whole execution robustness. To compensate for this, we consider adding an SC-fence when crossing libraries within every thread. The store buffering example involving the two register libraries clearly demonstrates that we need SC-fences when crossing libraries within every thread, but the question is whether this is sufficient in general for any execution involving any robust library implementations? We answer this in the positive, and formally prove that for executions composed of calls to multiple robust libraries, if there is an SC-fence within each thread when crossing different libraries, then the overall execution is guaranteed to be robust.

In the following, we assume each pair of libraries $\mathcal{L}_1$ and $\mathcal{L}_2$ are disjoint, i.e. there is no common method that belongs to both libraries. We also require that the sets of memory locations that may be accessed by each library are also disjoint from each other, which is enforced by the ownership assumption mentioned at the beginning of §3. We denote an SC fence (i.e. the three instruction program $\mathsf{fence}(\mathtt{acq}); \mathsf{fadd}(f, 0, \mathtt{acqrel}); \mathsf{fence}(\mathtt{rel})$) by $\mathsf{fence}(\mathtt{sc})$. Note that the hb relation will

$$\textsc{Invk}\quad \frac{\begin{array}{c}\mathcal{I}(\tau) = \langle \gamma', \bot\rangle \quad m = \lambda\, x.s \in \mathcal{L} \\ (\gamma' = \mathsf{Invk}\langle \_, m', \_, \_, \tau\rangle \wedge m' \in \mathcal{L}) \vee \gamma' = \bot \\ \varrho' = \varrho[\tau \mapsto \varrho(\tau)[x \mapsto v_a]] \\ i \text{ unique} \quad \gamma = \mathsf{Invk}\langle i, m, v_a, v_r, \tau\rangle \\ \mathcal{I}' = \mathcal{I}[\tau \mapsto \langle \gamma, s\rangle]\end{array}}{\langle \varrho, \mathcal{I}\rangle \overset{\gamma}{\Longrightarrow} \langle \varrho', \mathcal{I}'\rangle}$$

$$\textsc{InvkF}\quad \frac{\begin{array}{c}\mathcal{I}(\tau) = \langle \gamma', \bot\rangle \quad \gamma' = \mathsf{Invk}\langle \_, m'', \_, \_, \tau\rangle \quad m'' \notin \mathcal{L} \\ m = \lambda\, x.s \in \mathcal{L} \quad m' = \lambda\, x.\mathsf{fence}(\mathsf{sc}); s \\ \varrho' = \varrho[\tau \mapsto \varrho(\tau)[x \mapsto v_a]] \\ i \text{ unique} \quad \gamma = \mathsf{Invk}\langle i, m, v_a, v_r, \tau\rangle \\ \mathcal{I}' = \mathcal{I}[\tau \mapsto \langle \gamma, s\rangle]\end{array}}{\langle \varrho, \mathcal{I}\rangle \overset{\gamma}{\Longrightarrow} \langle \varrho', \mathcal{I}'\rangle}$$

Fig. 9. New Global Reduction Rules of $\Omega_{\mathbb{L}}$

be total among all the update events that are generated by the fadd instructions, due to the `acqrel` annotation, while the `acq` and `rel` fences are required to synchronize with any corresponding fences (if present) in the library implementations.

Given a set of libraries $\mathbb{L} = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, we define a LTS $\Omega_{\mathbb{L}}$, which is almost exactly the same as the LTS $\Omega_{\mathcal{L}}$ that was defined in §3 for a single library $\mathcal{L}$. Similar to $\Omega_{\mathcal{L}}$, each state of $\Omega_{\mathbb{L}}$ also consists of a thread-local environment per active session, and the current invocations indexed by session id, except that now, these invocations can come from any of the libraries in $\mathbb{L}$. The transition rules also remain the same, except the Invk rule in Fig. 6, which is replaced by the two new rules given in Fig. 9.

The new Invk rule is used to invoke a method m of library $\mathcal{L} \in \mathbb{L}$ if the method preceding it in its session belonged to the same library; otherwise, in order to invoke a method following one from a different library, the InvkF rule must be used. In this case the method call is preceded with an SC-fence statement but the rest of the rule is the same as Invk.

Because of the altered Invk rules, executions of $\Omega_{\mathbb{L}} \times \mathcal{MS}_X$, for X = SC or RC20, will always have an SC-fence at the boundary when crossing from a method of one library to a method of a different library in session order. Notice that there is no SC-fence between invocations of methods from the same library. Intuitively, the executions generated by $\Omega_{\mathbb{L}} \times \mathcal{MS}_X$ maintain a *well-fencedness* property: in any (complete or partial) execution, if there is an event e_1 generated by a method belonging to $\mathcal{L}_1$ preceding an event e_2 generated by a method belonging to a different library $\mathcal{L}_2$ in session order, then there must be an SC-fence in between them. Let $\mathcal{CE}^{\mathbb{L}}_X$ denote the set all complete executions generated by $\Omega_{\mathbb{L}} \times \mathcal{MS}_X$ for the memory system X.

Theorem 6.1. *Given a set of libraries $\mathbb{L}$, if each library in $\mathbb{L}$ is execution-graph robust, then all executions in $\mathcal{CE}^{\mathbb{L}}_{\text{RC20}}$ are also execution graph robust.*

We prove the above theorem by contradiction, starting with a non-robust execution in $\mathcal{CE}^{\mathbb{L}}_{\text{RC20}}$ and inspecting the shape of a minimal hb_{SC} cycle that must be present in the execution graph. We construct an execution comprising solely of events within a single library $\mathcal{L} \in \mathbb{L}$ in which the hb_{SC} cycle is preserved, thus providing a contradiction to the guarantee of robustness of $\mathcal{L}$. In the case where the original cycle involves events from multiple libraries, we use the presence of SC fences between libraries to infer hb synchronization, and we construct an RC20-execution of $\mathcal{L}$ by converting these hb relations to so by changing the assignment of invocations to sessions.

We can also extend the above result for induced subgraph robustness, using the same definition as given in Def. 5.1 for complete executions in $\mathcal{CE}^{\mathbb{L}}_{\text{RC20}}$.

Theorem 6.2. *Given a set of libraries $\mathbb{L}$, if each library in $\mathbb{L}$ is induced subgraph robust, then all executions in $\mathcal{CE}^{\mathbb{L}}_{\text{RC20}}$ are also induced subgraph robust.*

Intuitively, there may be benign non-robustness in executions of individual libraries, which does not affect its observable behavior. In such a scenario, for multi-library executions, this non-robustness would still exist in the overall execution. Hence, we may not be able to show the overall execution is execution-graph robust, but instead, we can safely remove the non-robustness of each

individual library to show induced subgraph robustness of the entire execution. This preserves the observable behavior of each invocation.

Our formulation thus far still does not allow a client to also perform atomic RC20 accesses. However, it is possible to view the client program as a library itself, with every program segment in every session between two library calls encapsulated as a separate method in this artificially generated client library. Then, Theorems 6.1 and 6.2 allow us to decompose the overall robustness problem into proving robustness for the actual libraries and the (synthetic) client library. In particular, as a special case, we can consider a client program which does not access any shared variable (i.e., leaves all the shared memory operations to the libraries), or a client program which only accesses shared memory inside locks. Such client programs will be robust according to our definition (i.e., the artificially generated client library will be robust), and hence composing them with robust libraries guarantee overall robustness.

We note that the usage of the SC fences is as important as the robustness of the individual libraries to guarantee overall robustness of the composite execution. If a library $\mathcal{L}$ can generate a (non-benign) non-robust execution on its own, it is obvious that we can orchestrate a composite execution involving multiple libraries along with $\mathcal{L}$ which would also be non-robust. A reader may wonder whether we can perhaps place more SC fences to perhaps not require individual library robustness, e.g. before and after every library invocation, regardless of whether the surrounding invocations belonged to the same library or not. However, this would also not work, because we can consider a library which has two methods corresponding to the two sessions involved in the store buffering anomaly, i.e. each method has two memory events, a store followed by load to different location, with opposite order of locations in the two methods. Essentially, each method performs the memory operations of one of the sessions involved in the store buffering anomaly. Then, in an execution where each of the methods is called in a separate session, even with the presence of SC fences before and after the invocations, store buffering is still possible.

7 Automated Verification

Our automated verification strategy relies on discovering violations of Φ_{src} and Φ_{dst}, which essentially corresponds to discovering potential and actual non-robustness witnesses. Towards this end, we model memory events using FOL domains, executions as relations between events and then we generate FOL queries that instantiate memory events involved in establishing Φ_{src} and Φ_{dst}. For example, referring back to Table 1, violations of Φ_{src} would require us to instantiate events μ_w, μ_1, μ_2 satisfying the antecedent of Φ_{src}, while the negation of the consequent can be simplified as $\neg(\mu_w \xrightarrow{\mathsf{hb}} \mu_2)$.

In addition to memory events which are directly involved in Φ_{src} and Φ_{dst}, for the library method containing these events, the FOL query also instantiates an event for every program statement present in the implementation of the method. We call the entire set of instantiated events as a *partial* execution, which is essentially the non-robust core of an actual execution. If such a non-robust core cannot be instantiated, then no potential or actual non-robustness witnesses can exist. Thus, if the generated FOL formulae are not satisfiable, we can conclude that Φ_{src} or Φ_{dst} hold, implying robustness of the library implementation.

Next, we require that the partial execution must be a part of a valid SC execution. To ensure this, we encode the constraints on the executions in terms of FOL formula obtained from the χ_{base} and χ_{SC} constraints of §3.2. For each relation $R \in \{\mathsf{rf}, \mathsf{mo}, \mathsf{fr}, \mathsf{sw}, \mathsf{so}, \mathsf{hb}, \mathsf{hb}_{\mathsf{SC}}\}$, we encode constraints that ensure or prohibit their presence between pairs of events. For instance, all events belonging to the same invocation must be related by so, two events writing to the same location must be related

by mo, there must be a rf between a unique write event to a location and a read event returning the same value, etc. We also encode how derived relations depend on the base relations.

Note that Φ_{src} and Φ_{dst} need to be checked individually for every location used in any execution of the implementation. While an implementation can allocate and access an unbounded number of locations on the heap during executions, to generate our encoding, we use a fixed, finite number of *location classes*. We define a location class for every shared global variable and every field (for record types allocated on the heap). For example, the location classes for the register implementation of Fig. 2 are the global variable `L` and the field `val`.

Instantiating partial executions with a fixed, finite number of events may result in a number of false positives, i.e. partial executions that would not be a part of any complete execution. To prune these false positives, we introduce an analysis phase before checking Φ_{src} and Φ_{dst} that derives useful constraints obeyed by any SC execution of the library implementation. These constraints are then expressed as universally quantified FOL formulae and added to the encoding. The derived constraints can be broadly classified into two classes: (1) *program structure* constraints that are derived by a static analysis of the implementation and (2) *specification* constraints that are directly obtained from the specification of the implemented data structure (as described in §3.3). Since we assume that the library implementation is correct under SC, we can directly incorporate these specification constraints.[4] We find these derived constraints to be effective in helping to establish robustness conditions over low-level memory events.

Input: Library Implementation L, Specification axioms $\Psi_{\mathcal{D}}$
$L' \leftarrow \rho_{CAS}(L)$
$\Psi_{Analysis} \leftarrow ConstraintAnalysis(L', \Psi_{\mathcal{D}})$
$\Psi \leftarrow \Psi_{base} \wedge \Psi_{SC} \wedge \Psi_{Analysis}$
foreach $\ell \in$ LocClass(L') **do**
  if $\neg$ *Check*-$\Phi_{src}(\Psi, L', \ell)$ **then**
    if $\neg$ *Check*-$\Phi_{dst}(\Psi, L', \ell)$ **then**
      return L may be non-robust
    end
  end
end
return L is robust

Fig. 10. Main Algorithm

For each library implementation, we generate (i) a set of access constraints from the provided implementation code and the library specification that e.g., identify locations written to at most once in an execution, are always written to within the same invocation, etc., or (ii) CAS constraints that identify locations exclusively modified using `acqrel` CAS operations and thus are totally ordered under the hb relation, etc. We additionally extract (iii) specification constraints from the axiomatic, declarative specification of the data structure to generate additional hb edges. For example, for the register implementation shown in Figure 1 we add the specifications χ_{GetSet} and χ_{GetUNDEF}, which state that a get should return the value from another set and if a get returns `UNDEF`, then no set method should have been executed before it.

Figure 10 depicts the algorithm for checking robustness, combining the various components discussed above. The algorithm takes as input a library implementation L, along with the specification axioms ($\Psi_{\mathcal{D}}$) of the implemented data structure $\mathcal{D}$. We first perform a robustness-preserving CAS-to-BCAS transformation ρ_{CAS} (shown in 5.4) to obtain the modified implementation L'. We then populate the constraint set $\Psi_{Analysis}$ as described above. We then iterate through all the location classes, and check if either Φ_{src} or Φ_{dst} hold for each location class. Each check for Φ_{src} or Φ_{dst} will check the feasibility of instantiating a partial execution involving a violation to the corresponding condition, in the presence of the constraints Ψ. This feasibility check is reduced to checking the satisfiability of a FOL formula.

[4] Complete details about the SMT encoding and the derived constraints are provided in the supplemental material, §D.

Table 2. Results of applying RoboCop to RC20 concurrent data structure libraries. All benchmarks were verified to be robust.

Benchmark	Time (s)	Locs	RLX	RA	Tot
Atomic Reference Counter [20]	10.21	3	2	4	6
Singleton [5]	8.43	4	0	6	6
Read Copy Update [39]	9.20	4	0	8	8
Spinlock [47]	12.54	2	2	4	6
Seqlock [10]	10.82	4	4	4	8
Ticketlock [47]	4.54	2	1	3	4
Lamport Mutex [42]	9.78	5	0	12	12
Peterson Lock [43]	6.54	3	3	3	6
Dekkers Mutex [54]	8.22	3	8	2	10
Treiber Stack [53]	13.67	3	5	3	8
Herlihy-Wing Queue [30]	12.39	2	0	4	4
TwoLock Queue [45]	18.12	6	7	7	14
Lockfree Queue [45]	24.55	4	5	7	12
SPSC Queue [50]	11.05	3	4	4	8
MPMC Bounded Queue [44]	18.45	4	2	8	10
MPMC Unbounded Queue [26]	25.74	6	4	12	16
Non-blocking Set [29]	19.42	3	6	6	12
Work-stealing Queue [17]	28.40	3	10	4	14

8 Evaluation

We have implemented a tool called RoboCop to test our methodology on real-world benchmarks. RoboCop takes a library of methods written in C11 and produces a result that indicates if the library is robust under RC20. RoboCop directly implements the algorithm of Figure 10. RoboCop parses the provided inputs, performs the ρ_{CAS} robustness transformation (5.4), and generates the necessary SMTLIB encoding needed to specify constraints (§7), which is then discharged by Z3.

Our evaluation considers a number of real-world benchmarks adapted from the literature and open-source repositories. All these implementations have SC specifications which have been proven to be correct, but their robustness under RC20 has not been established automatically prior to this work. All experiments were executed on an Intel® Core™ i5-7200U CPU @ 2.50 GHz, Ubuntu 18.04 machine using Z3 4.8.10. Table 2 summarizes key results. Column Loc denotes the number of location classes considered for verification. Columns RLX, RA, and Tot denote the number of relaxed accesses, release-acquire operations (accesses and fences), and total operations performed, which includes accesses to location classes and fence operations in the library source code. As these numbers indicate, most of the benchmarks make meaningful use of relaxed and release-acquire operations. Our benchmarks cover the following commonly occuring access patterns in real-world libraries - static locations, dynamically allocated locations and locations accessed by an offset into an array.

Next, we describe situations where the derived constraints aid the automated verification procedure. We note that removing any of these constraints either produces a robustness violation that is a false-positive or the solver loops until timeout, trying to unroll and instantiate memory events.

Access Constraints. The Treiber stack, Lockfree queue, MPMC Unbounded queue and Non-blocking set follow the pattern of creating a new node on a push/add operation and then linking it

to the main data structure. Thus, the updates to the fields of these new nodes can only be performed inside these methods and removing this constraint generates false-positive memory events where two writes may update this field and lead to a non-robustness witness, if observed by a read.

In the case of the SPSC bounded queue, MPMC bounded and unbounded queues and Chase-Lev deque, the threads access an underlying array, using integer indices. The access constraint-generated states that writes to different indices are made by different threads, ensuring that a read cannot potentially read from two different writes, leading to a non-robustness witness.

CAS Constraints: Since all the data structures are written to be lock-free, they make use of a CAS update to a designated location to update the data structure and retrying if the CAS fails. In the Treiber stack, this is the head of the stack and in the Lockfree queue, these are the head and tail of the queues. Similarly, the indices signalling where to write into the array for the SPSC bounded queue, MPMC bounded queue and MPMC unbounded queue are updated using fetch-and-add instructions, ensuring that a unique thread gets access to the index element. The CAS constraints ensure that two events are not instantiated that read from different orderings to these locations, since the CAS's impose a total order.

Specification constraints: For each benchmark, we add the specification constraint and map it to the internal program statements of the benchmarks. For example, the `AddRem` constraint for stacks, queues and sets ensures that in the respective implementations, there is an hb edge between a push and a pop that have matching argument and return values. This is translated to an hb edge between the program points in the method invocations where the method executes, in the SC order.

In the case of the libraries that create and link nodes, such as Treiber stack and Lockfree queue, this ensures that a pop operation always reads from a unique push operation (thus ruling out a non-robust witness, where a pop may read from two different push operations). Similarly, for implementations which use indexing, this also ensures that there is a single push operation at a certain index, that the pop operation reads from.

To demonstrate that our approach can also detect robustness violations, we systematically relaxed memory access statements in these benchmarks to create a non-robust implementation. RoboCop was successfully able to provide counter-examples in the form of partial executions that serve as a witness to the violation for all modified benchmarks.[5]

Finally, we performed an experiment to understand the impact of the SC-fence insertion overhead in the presence of multiple instances of a stack or queue library. We create a benchmark scenario where there are N producers, N consumers and N intermediaries. The producers add messages to a queue and the intermediaries remove messages from this queue. Then, the intermediaries push these messages to another queue and finally the consumers remove messages from the second queue. This benchmark models a message bus or pipeline, observed in real-world software. To ensure compositionality, the intermediaries need to insert a fence after the pop from the first queue and before the push to the second queue. We measure how long it takes to complete a fixed number of operations, from producers to consumers. We run the experiments on a `c7g.metal` AWS instance, that has 64 ARMv8.4 cores. We run the benchmarks for 1M operations by each thread, over $N = 16$ consumers ($N = 1$ for SPSC queue), producers and intermediaries. We run the benchmark for 3 cases, where in each benchmark the intermediary data structures are set to be the `Boost 1.74` libraries for SPSC queue, MPMC queue and MPMC stack and observe 4.79%, 7.39% and 3.69% increase in runtime when SC fences (`DMB ISH`) are inserted between pop and push operations.

[5] Additional details are provided in the supplemental material.

9 Related Work and Conclusion

There has been substantial prior work on determining robustness against hardware models [3, 4, 12, 19], with the x86-TSO memory model being particularly well-studied [11, 48]. Guaranteeing data-race freedom (DRF-SC) [2, 21] is a well-known instantiation of robustness applicable to a language's concurrency semantics. The notion of execution graph robustness closely resembles the DRF-SC property, and indeed prior work [43] has formally proved a correspondence between the two. However, in this work, our main contribution is a fully automated approach for verifying this property in the context of library implementations in the presence of the most general client. As discussed earlier, [38] presents a method to verify execution-graph robustness against the C11 release-acquire concurrency semantics, implementing their procedure using a model-checker that operates over programs with a finite data domain. [43] extends this result to additionally support relaxed accesses and release/acquire fences. Their approach also takes into account speculative actions as part of their robustness formulation, similar to our definition of effect robustness, but as described earlier, our notion of robustness is weaker and better suited for library implementations. Other works [34–36] have also proposed model-checking based techniques for verifying programs under weak memory that cannot, however, be soundly used for verifying library implementations in the presence of a most-general client.

A number of prior works have also considered the specification and verification problem of libraries in a relaxed memory setting [6, 13, 22, 24, 31, 49, 52]; these efforts, however, do not consider automated verification tooling or robustness arguments in their proof methodology. [14, 16, 27, 28, 32] also propose correctness notions that are weaker than linearizability - we consider the incorporation of these ideas as a topic for future research.

Recent work [18, 44, 49] has developed new proof techniques to modularly reason about clients that interact with libraries which often have weaker specifications that expose relaxed memory behavior. We focus on an orthogonal problem in this paper - establishing the robustness of library implementations that internally use relaxed memory primitives. As we have shown here, libraries that internally use relaxed accesses may have enough synchronization to make them robust, enabling a pathway to automated verification. Having said that, we also believe that synchronization specifications of libraries as proposed by these other efforts would be useful in addition to robustness guarantees, since they would allow an optimal SC fence placement strategy for guaranteeing robustness of executions involving multiple libraries.

This paper presents a modular verification strategy for verifying robustness of programs using library implementations. Our verification strategy adapts the notion of execution-graph robustness to the library setting, and exploits specification axioms of the library under SC, to generate constraints sufficient to imply a suitable inductive robustness invariant. We also show how to effectively compose robustness guarantees of multiple libraries. We have successfully demonstrated our technique on a number of challenging real-world concurrent data structure implementations that meaningfully exploit sophisticated weak memory behavior. Our results suggest that automated robustness proofs can be effectively applied to ascertain whether concurrent libraries, specified assuming sequential consistency, can be safely refined to exploit relaxed atomics.

Data-Availability Statement

The software that supports Section. 8 is available on Zenodo [46].

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under grant CCF-FMiTF 2019263. The first author is also supported by a grant from the Tezos Foundation.

References

[1] 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*.

[2] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering—a New Definition. *SIGARCH Comput. Archit. News* 18, 2SI (may 1990), 2–14. https://doi.org/10.1145/325096.325100

[3] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion. *ACM Trans. Program. Lang. Syst.* 39, 2 (2017), 6:1–6:38. https://doi.org/10.1145/2994593

[4] Jade Alglave and Luc Maranget. 2011. Stability in Weak Memory Models. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 50–66. https://doi.org/10.1007/978-3-642-22110-1_6

[5] Helge Bahmann and Tim Blechmann. 2012. https://www.boost.org/doc/libs/1_85_0/libs/atomic/doc/html/atomic/usage_examples.html. Accessed 31-07-2024.

[6] Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, 235–248.

[7] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2012), Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 509–520. https://doi.org/10.1145/2103656.2103717

[8] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66. https://doi.org/10.1145/1926385.1926394

[9] Hans-Juergen Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-Of-Thin-Air Results. In *Proceedings of the workshop on Memory Systems Performance and Correctness, MSPC '14, Edinburgh, United Kingdom, June 13, 2014*, Jeremy Singer, Milind Kulkarni, and Tim Harris (Eds.). ACM, 7:1–7:6. https://doi.org/10.1145/2618128.2618134

[10] Hans-J. Boehm. 2012. Can seqlocks get along with programming language memory models?. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (Beijing, China) *(MSPC '12)*. Association for Computing Machinery, New York, NY, USA, 12–20. https://doi.org/10.1145/2247684.2247688

[11] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 533–553. https://doi.org/10.1007/978-3-642-37036-6_29

[12] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 12–21. https://doi.org/10.1145/1250734.1250737

[13] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Concurrent Library Correctness on the TSO Memory Model. In *European Symposium on Programming*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 87–107.

[14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 271–284. https://doi.org/10.1145/2535838.2535848

[15] C11 Last Accessed Nov 9 2022. C11 Memory Model (std::memory_order). https://en.cppreference.com/w/cpp/atomic/memory_order.

[16] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. 2015. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks. In *Distributed Computing*, Yoram Moses (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 420–435.

[17] David Chase and Yossi Lev. 2005. Dynamic Circular Work-Stealing Deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Las Vegas, Nevada, USA) *(SPAA '05)*. Association for Computing Machinery, New York, NY, USA, 21–28. https://doi.org/10.1145/1073970.1073974

[18] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA,*

USA, June 13 - 17, 2022, Ranjit Jhala and Isil Dillig (Eds.). ACM, 792–808. https://doi.org/10.1145/3519939.3523451

[19] Egor Derevenetc and Roland Meyer. 2014. Robustness Against POWER is PSpace-Complete. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8573)*, Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.). Springer, 158–170. https://doi.org/10.1007/978-3-662-43951-7_14

[20] Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 448–475.

[21] Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 242–255. https://doi.org/10.1145/3192366.3192421

[22] Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. 2018. On Abstraction and Compositionality for Weak-Memory Linearisability. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer International Publishing, 183–204.

[23] Michael Emmi and Constantin Enea. 2018. Sound, Complete, and Tractable Linearizability Monitoring for Concurrent Collections. *Proc. ACM Program. Lang.* 2, POPL (2018), 25:1–25:27. https://doi.org/10.1145/3158113

[24] Michael Emmi and Constantin Enea. 2019. Weak-Consistency Specification via Visibility Relaxation. *Proc. ACM Program. Lang.* 3, POPL, Article 60 (Jan 2019), 28 pages. https://doi.org/10.1145/3290373

[25] Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Monitoring Refinement via Symbolic Reasoning. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 260–269. https://doi.org/10.1145/2737924.2737983

[26] Folly Last Accessed Nov 9 2022. Meta Folly MPMC Queue. https://github.com/facebook/folly/blob/main/folly/MPMCQueue.h.

[27] Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. 2016. Local Linearizability for Concurrent Container-Type Data Structures. In *27th International Conference on Concurrency Theory (CONCUR 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 59)*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.6

[28] Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular Verification of Concurrency-Aware Linearizability. In *Distributed Computing*, Yoram Moses (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 371–387.

[29] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[30] M Herlihy and J Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. In *ACM TOPLAS, 12(3)*.

[31] Radha Jagadeesan, Gustavo Petri, Corin Pitcher, and James Riely. 2013. Quarantining Weakness. In *European Symposium on Programming*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 492–511.

[32] Radha Jagadeesan and James Riely. 2014. Between Linearizability and Quiescent Consistency. In *Automata, Languages, and Programming*, Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–231.

[33] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

[34] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL (2018), 17:1–17:32. https://doi.org/10.1145/3158105

[35] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. 2022. Truly Stateless, Optimal Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498711

[36] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 96–110. https://doi.org/10.1145/3314221.3314609

[37] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 649–662. https://doi.org/10.1145/2837614.2837643

[38] Ori Lahav and Roy Margalit. 2019. Robustness Against Release/Acquire Semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 126–141. https://doi.org/10.1145/3314221.3314604

[39] Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Automata, Languages, and Programming*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323.

[40] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632. https://doi.org/10.1145/3062341.3062352

[41] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[42] Leslie Lamport. 1987. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.* 5, 1 (jan 1987), 1–11. https://doi.org/10.1145/7351.7352

[43] Roy Margalit and Ori Lahav. 2021. Verifying Observational Robustness Against a C11-style Memory Model. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–33. https://doi.org/10.1145/3434285

[44] Glen Mével and Jacques-Henri Jourdan. 2021. Formal Verification of a Concurrent Bounded Queue in a Weak Memory Model. *Proc. ACM Program. Lang.* 5, ICFP, Article 66 (aug 2021), 29 pages. https://doi.org/10.1145/3473571

[45] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, James E. Burns and Yoram Moses (Eds.). ACM, 267–275. https://doi.org/10.1145/248052.248106

[46] Kartik Nagar, Anmol Sahoo, Romit Roy Chowdhury, and Suresh Jagannathan. 2024. *Artifact - Automated Robustness Verification of Concurrent Data Structure Libraries Against Relaxed Memory Models.* https://doi.org/10.5281/zenodo.13626195

[47] Scott Owens. 2010. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *European Conference on Object-Oriented Programming*. https://api.semanticscholar.org/CorpusID:28735505

[48] Scott Owens. 2010. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 478–503. https://doi.org/10.1007/978-3-642-14107-2_23

[49] Azalea Raad, Marko Doko, Lovro Rozic, Ori Lahav, and Viktor Vafeiadis. 2019. On Library Correctness Under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries Under Declarative Consistency Models. *PACMPL* 3, POPL (2019), 68:1–68:31. https://doi.org/10.1145/3290381

[50] Boris Schling. 2011. *The Boost C++ Libraries.* XML Press.

[51] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22:1–22:50. https://doi.org/10.1145/2487241.2487248

[52] Abhishek Kr Singh and Ori Lahav. 2023. An Operational Approach to Library Abstraction under Relaxed Memory Concurrency. *Proc. ACM Program. Lang.* 7, POPL (2023), 1542–1572. https://doi.org/10.1145/3571246

[53] R.K. Treiber. 1986. *Systems programming: Coping with Parallelism. Technical Report RJ-5118.* Technical Report. IBM Almaden Research Center.

[54] Anthony Williams. Last Accessed Nov 9, 2022. Implementing Dekker's algorithm with fences. https://www.justsoftwaresolutions.co.uk/threading/implementing_dekkers_algorithm_with_fences.html.

A Program Semantics

The two step rules (StepLocal and StepMem) encapsulate the behavior of the library and differ only with respect to the event label recorded in its transition. They both rely on an auxiliary relation ($\rightarrow$) that specifies the behavior of the library. The salient rules comprising its definition, which loosely follow prior operational formulations of similar systems (e.g., [38, 43]) are shown in Figure 12. The StepLocal rule applies when an invocation takes a step that does not involve access or updates to memory ($\xrightarrow{\epsilon}_\tau$); this action is recorded as a silent step in the global trace. The StepMem rule applies when an invocation performs a memory action step ($\xrightarrow{\alpha}_\tau$).

The local reduction rules are mostly standard, as shown in Fig. 12; the rules use a local environment ρ to hold bindings for local variables, and local evaluation relation ($\rightsquigarrow$) to evaluate expressions. Its most notable aspect is that load operations on shared locations are unconstrained, and can return any arbitrary value. Later, we will define a memory system to generate valid traces of memory events following the C11 relaxed memory model. We will then construct a combined

$$M' = M \cup \{\mu\} \qquad \mathsf{rf}' = \begin{cases} \mathsf{rf} \cup \{(w,\mu)\} & if\ \mathsf{type}(\mu) = \mathtt{R}\ \text{or}\ \mathtt{U} \\ \mathsf{rf} & otherwise \end{cases}$$

$$\mathsf{so}' = \mathsf{so} \cup \{(\mu',\mu) \mid \mu' \in G.M \wedge \mathsf{sess}(\mu') = \tau\} \qquad \mathsf{mo}' = \begin{cases} \mathsf{mo} \cup \{(w',\mu) \mid \mathsf{mo}(w',w) \vee (w' = w)\} \\ \cup\{(\mu,w'') \mid \mathsf{mo}(w,w'')\} & if\ \mathsf{type}(\mu) = \mathtt{W}\ \text{or}\ \mathtt{U} \\ \mathsf{mo} & otherwise \end{cases}$$

Fig. 11. The definition of $\mathsf{add}(G, i, \tau, \alpha, w)$.

transition system by taking the product of the program semantics and the memory system, such that the transitions in the two systems agree on every memory event. We will also define a memory system for Sequential Consistency (SC), which when combined with the program semantics will describe the behavior of the library implementation under SC.

We now present the memory systems for the SC and RC20 memory models. We define a common parameterized system that can be instantiated with either χ_{SC} or χ_{RC20}. The labeled transition system for memory model X is given by $\mathcal{MS}_X = \langle \mathcal{G}, \mathsf{MemEvt}, \rightarrow_X \rangle$. Here, $\mathcal{G}$ is the set of all execution graphs, MemEvt are memory events as defined earlier, and $\rightarrow_X \subseteq \mathcal{G} \times \mathsf{MemEvt} \times \mathcal{G}$ are labeled transitions. We define an initial execution graph $G_\perp = \langle M_\perp, \emptyset, \emptyset, \emptyset \rangle$, where $M_\perp$ contains an initial write event $\mathtt{W}\langle\, \ell, v_\ell^\perp, \mathtt{rlx} \,\rangle$ to every location ℓ.

Each transition of the memory system adds a new memory event to the execution graph. Given $G = \langle M, \mathsf{rf}, \mathsf{mo}, \mathsf{so} \rangle$, we define a function $\mathsf{add}(G, i, \tau, \alpha, w)$, shown in Figure 11, to compute the new execution graph $G' = \langle M', \mathsf{rf}', \mathsf{mo}', \mathsf{so}' \rangle$, which adds the new memory event $\mu = \mathsf{Mem}\langle\, i, \tau, \alpha \,\rangle$ in session τ performing the operation α. Note that $w \in M_{\mathtt{W},\mathsf{loc}(\mu)} \cup M_{\mathtt{U},\mathsf{loc}(\mu)}$ is a write/update event to the location of μ.

The transition of the memory system can now be defined using the add function:

$$\frac{\begin{array}{c} \tau \in \mathsf{SessionId} \quad \alpha \in \mathsf{Action} \quad i\ \text{unique} \\ \mu = \mathsf{Mem}\langle\, i, \tau, \alpha \,\rangle \quad w \in G.M_{\mathtt{W},\mathsf{loc}(\mu)} \cup G.M_{\mathtt{U},\mathsf{loc}(\mu)} \\ G' = \mathsf{add}(G, i, \tau, \alpha, w) \quad \chi_{\mathsf{base}}(G') \quad \chi_X(G') \end{array}}{G \xrightarrow{\mu}_X G'}$$

BIND

$$\frac{\rho, e \rightsquigarrow^* v \quad \rho' = \rho[x \mapsto v]}{\langle \rho, x \;=\; e\rangle \xrightarrow{\epsilon}_{\tau} \langle \rho', \mathsf{skip}\rangle}$$

LOAD

$$\frac{\rho(x) = \ell}{\langle \rho, \mathsf{load}(x, o_{\mathsf{R}})\rangle \xrightarrow{\mathsf{R}\langle \ell, v, o_{\mathsf{R}}\rangle} \langle \rho, \mathsf{skip}\rangle}$$

STORE

$$\frac{\rho,\ e \rightsquigarrow^* v \quad \rho(x) = \ell}{\langle \rho, \mathsf{store}(x, e, o_{\mathsf{W}})\rangle \xrightarrow{\mathsf{W}\langle \ell, v, o_{\mathsf{W}}\rangle} \langle \rho, \mathsf{skip}\rangle}$$

CASTRUE

$$\frac{\rho, e_1 \rightsquigarrow^* v_1 \quad \rho, e_2 \rightsquigarrow^* v_2 \quad \rho(x) = \ell}{\langle \rho, \mathsf{cas}(x, e_1, e_2, o_{\mathsf{RMW}})\rangle \xrightarrow{\mathsf{U}\langle \ell, v_1, v_2, o_{\mathsf{RMW}}\rangle} \langle \rho, \mathsf{true}\rangle}$$

CASFALSE

$$\frac{\rho, e_1 \rightsquigarrow^* v' \quad v' \neq v \quad \rho(x) = \ell}{\langle \rho, \mathsf{cas}(x, e_1, e_2, o_{\mathsf{RMW}})\rangle \xrightarrow{\mathsf{R}\langle \ell, v, o_{\mathsf{RMW}}\rangle} \langle \rho, \mathsf{false}\rangle}$$

FENCE

$$\frac{}{\langle \rho, \mathsf{fence}(o_{\mathsf{F}})\rangle \xrightarrow{\mathsf{F}\langle o_{\mathsf{F}}\rangle} \langle \rho, \mathsf{skip}\rangle}$$

SEQ

$$\frac{\langle \rho, s_1\rangle \xrightarrow{\omega} \langle \rho', s_1'\rangle}{\langle \rho, s_1; s_2\rangle \xrightarrow{\omega} \langle \rho', s_1'; s_2\rangle}$$

SEQQ

$$\frac{\langle \rho, s_1\rangle \xrightarrow{\omega} \langle \rho', \mathsf{skip}\rangle}{\langle \rho, s_1; s_2\rangle \xrightarrow{\omega} \langle \rho', s_2\rangle}$$

IFTRUE

$$\frac{e \rightsquigarrow^* \mathsf{true}}{\langle \rho, \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\rangle \xrightarrow{\epsilon} \langle \rho, s_1\rangle}$$

IFTRUE

$$\frac{e \rightsquigarrow^* \mathsf{false}}{\langle \rho, \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\rangle \xrightarrow{\epsilon} \langle \rho, s_2\rangle}$$

WHILETRUE

$$\frac{e \rightsquigarrow^* \mathsf{true}}{\langle \rho, \mathsf{while}\ e\ \mathsf{do}\ s\rangle \xrightarrow{\epsilon} \langle \rho, s;\ \mathsf{while}\ e\ \mathsf{do}\ s\rangle}$$

WHILEFALSE

$$\frac{e \rightsquigarrow^* \mathsf{false}}{\langle \rho, \mathsf{while}\ e\ \mathsf{do}\ s\rangle \xrightarrow{\epsilon} \langle \rho, \mathsf{skip}\rangle}$$

Fig. 12. Selected local reduction rules.

B Proofs

Theorem 4.4 Given a library implementation L, if every prefix E' of every complete execution $E \in \mathcal{CE}^L_{\mathsf{SC}}$ is not a non-robustness witness, then L is execution-graph robust.

Proof. We will show that if L is not execution-graph robust, then there exists a complete execution E which has a prefix E' which is non-robustness witness satisfying Definition 4.3.

If L is not execution-graph robust, then by definition, there exists a complete execution which is in $\mathcal{CE}^L_{\mathsf{RC20}}$ but not in $\mathcal{CE}^L_{\mathsf{SC}}$. We consider such an execution $E \in \mathcal{CE}^L_{\mathsf{RC20}}$ which has the smallest prefix (in terms of number of events) E' such that $E' \notin \mathcal{E}^L_{\mathsf{SC}}$. We will show that this prefix E' and its next event $\mu = \mathsf{next}(E', E)$ form a non-robustness witness.

Suppose the event μ accesses the location ℓ and is present in session τ. Then, $E'.w^{\max}_\ell \in dom(E'.\mathsf{hb}^?_{\mathsf{SC}}; [E.M_\tau])$. This is because otherwise, there exists a shorter execution E'' which does not contain $E.w^{\max}_\ell$ and all other events in E' which occurs $\mathsf{hb}_{\mathsf{SC}}$-after it, but contains all the other events in E'. All the other events remain the same, and hence the next event in E'' will also be μ. Now, $E'' + \mu$ becomes a shorter prefix that is not in $\mathcal{E}^L_{\mathsf{SC}}$, thus contradicting our assumption that E' is the shortest such prefix.

Since $E.w^{\max}_\ell \in dom(E'.\mathsf{hb}^?_{\mathsf{SC}}; [E'.M_\tau])$, this satisfies the first criterion of Definition 4.3. Now, we consider different cases based on the type of the event μ:

Case $\mathrm{act}(\mu) = \mathtt{R}$: Let $E'' = E' + \mu$. Consider the event μ_w such that $\mu_w \xrightarrow{E''.\mathsf{rf}} \mu$. Then, μ_w cannot be $E'.w^{\max}_\ell$, because otherwise, $E' + \mu \in \mathcal{E}^E_{\mathsf{SC}}$. We will now show that $\mu_w \notin dom(E'.\mathsf{mo}; E'.\mathsf{rf}^?; E'.\mathsf{hb}^?; [E'.M]_\tau)$ by contradiction.

We will show that E'' cannot be in $\mathcal{E}^L_{\mathsf{RC20}}$. Suppose there exists μ' such that $\mu_w \xrightarrow{E'.\mathsf{mo}} \mu'$ and $\mu' \xrightarrow{E'.\mathsf{hb}} \mu''$ where $\mu'' \in E'.M_\tau$. Then, $\mu' \xrightarrow{E''.\mathsf{hb}} \mu$. We also have $\mu \xrightarrow{E''.\mathsf{fr}} \mu'$. This implies that $E''.\mathsf{fr}; E''.\mathsf{hb}$ is not irreflexive, which contradicts the first condition in the definition of RC20 consistency.

Suppose $\mu_w \xrightarrow{E'.\mathsf{mo}} \mu'$ and $\mu' \xrightarrow{E'.\mathsf{rf}} \mu''$, where $\mu'' \in E'.M_\tau$. Now, $\mu'' \xrightarrow{E''.\mathsf{hb}} \mu$, and $\mu \xrightarrow{E''.\mathsf{fr}} \mu'$. This implies that $E''.\mathsf{fr}; E''.\mathsf{rf}; E''.\mathsf{hb}$ is not irreflexive, which again contradicts the first condition in the definition of RC20 consistency.

Suppose $\mu_w \xrightarrow{E'.\mathsf{mo}} \mu'$ and $\mu' \in E'.M_\tau$. In this case. $\mu' \xrightarrow{E''.\mathsf{hb}} \mu$ and $\mu \xrightarrow{E''.\mathsf{fr}} \mu'$, which implies that $E''.\mathsf{fr}; E''.\mathsf{hb}$ is not irreflexive. This concludes the proof that $\mu_w \notin dom(E'.\mathsf{mo}; E'.\mathsf{rf}^?; E'.\mathsf{hb}^?; [E'.M]_\tau)$.

Case $\mathrm{act}(\mu) = \mathtt{W}$: Consider the event $\mu_w \in E'.M$ such that $\mu_w \xrightarrow{E''.\mathsf{mo}} \mu$ and μ is the immediate successor of μ_w according to the $E''.\mathsf{mo}$ ordering. First, suppose that there exists $\mu' \in E'.M_{\mathsf{U}}$ such that $\mu_w \xrightarrow{E'.\mathsf{rf}} \mu'$. Then, we must have $\mu_w \xrightarrow{E'.\mathsf{mo}} \mu'$, because otherwise, $E'.\mathsf{mo}; E'.\mathsf{rf}$ would become reflexive.

Now, from $\mu_w \xrightarrow{E'.\mathsf{mo}} \mu'$ and μ being the immediate successor of μ_w in the mo ordering, we get that $\mu \xrightarrow{E''.\mathsf{mo}} \mu'$. We also have $\mu' \xrightarrow{E''.\mathsf{fr}} \mu$. This implies that $E''.\mathsf{fr}; E''.\mathsf{mo}$ is not irreflexive, which contradicts the third condition in the definition of RC20 consistency. Hence, $\mu_w \notin dom(E'.\mathsf{rf}; [E'.M]_{\mathsf{U}})$

We will now show that $\mu_w \notin dom(E'.\mathsf{mo}; E'.\mathsf{rf}^?; E'.\mathsf{hb}^?; [E'.M]_\tau)$ by contradiction. Suppose there exists μ' such that $\mu_w \xrightarrow{E'.\mathsf{mo}} \mu'$ and $\mu' \xrightarrow{E'.\mathsf{hb}} \mu''$ where $\mu'' \in E'.M_\tau$. This implies that $\mu' \xrightarrow{E''.\mathsf{hb}} \mu$, and since μ is the immediate mo-successor of μ_w, we have $\mu \xrightarrow{E''.\mathsf{mo}} \mu'$. Now, $E''.\mathsf{mo}; E''.\mathsf{hb}$ is not irreflexive, which contradicts the second condition in the definition of RC20 consistency.

Suppose there exists μ' such that $\mu_w \xrightarrow{E'.\mathsf{mo}} \mu'$ and $\mu' \xrightarrow{E'.\mathsf{rf}} \mu''$ where $\mu'' \in E'.M_\tau$. Then, that $\mu'' \xrightarrow{E''.\mathsf{hb}} \mu$, and since μ is the immediate mo-successor of μ_w, we have $\mu \xrightarrow{E''.\mathsf{mo}} \mu'$. Now,

$E''.\mathsf{mo}; E''.\mathsf{rf}; E''.\mathsf{hb}$ is not irreflexive, which contradicts the second condition in the definition of RC20 consistency. This concludes the proof that $\mu_w \notin dom(E'.\mathsf{mo}; E'.\mathsf{rf}^?; E'.\mathsf{hb}^?; [E'.M]_\tau)$.

Case $\mathsf{act}(\mu) = \mathsf{U}$: Similar reasoning to the case $\mathsf{act}(\mu) = \mathsf{R}$. □

Theorem 4.6. Given a library implementation L, if $\forall \ell \in \mathsf{Location}. \forall E \in \mathcal{CE}^L_{\mathrm{SC}}.\ \Phi_{dst}(E, \ell) \vee \Phi_{src}(E, \ell)$, then L is execution-graph robust.

Proof. For any execution $E \in \mathcal{CE}^L_{\mathrm{SC}}$, we will show that the following is an inductive invariant over the trace of the execution:

$$\text{For execution } E \in \mathcal{CE}^L_{\mathrm{SC}}, \text{for prefix } E' \text{ of } E,\ \forall \mu_w \in E'.M^{\mathsf{max}}_{\mathsf{W}} \cup E'.M^{\mathsf{max}}_{\mathsf{U}}. \forall \mu \in E'.M.$$

$$\Phi_{dst}(E, \mathsf{loc}(\mu_w)) \vee (\mu_w \xrightarrow{E'.\mathsf{hb}_{\mathrm{SC}}} \mu \wedge \Phi_{src}(E, \mathsf{loc}(\mu_w)) \Rightarrow \mu_w \xrightarrow{E.\mathsf{hb}} \mu)$$

This in turn would imply that no prefix of an execution can be a non-robustness witness, which by Theorem 4.4, would imply that L is execution-graph robust.

The base case is trivial as the set of memory events is empty.

For the inductive case, consider a prefix E' of E and its next event μ_2. Let $E'' = E' + \mu_2$. Consider some write event $\mu_w \in E''.M^{\mathsf{max}}_{\mathsf{W}} \cup E''.M^{\mathsf{max}}_{\mathsf{U}}$. If $\Phi_{dst}(E, \mathsf{loc}(\mu_w))$ holds, then the required condition is established for all $\mu \in E''.M$. Hence, suppose $\neg\Phi_{dst}(E, \mathsf{loc}(\mu_w))$. Then, we must have $\Phi_{src}(E, \mathsf{loc}(\mu_w))$.

Now, by the inductive hypothesis, the required condition holds for all $\mu \in E'.M$. Hence, we only need to consider the case where $\mu_w \xrightarrow{E''.\mathsf{hb}_{\mathrm{SC}}} \mu_2$. For this to happen, either $\mu_w \xrightarrow{E''.\mathsf{rf} \vee E''.\mathsf{fr} \vee E''.\mathsf{mo} \vee E''.\mathsf{so}} \mu_2$ or there exists another event $\mu_1 \in E'.M$ such that $\mu_w \xrightarrow{E'.\mathsf{hb}_{\mathrm{SC}}} \mu_1$ and $\mu_1 \xrightarrow{E''.\mathsf{rf} \vee E''.\mathsf{fr} \vee E''.\mathsf{mo} \vee E''.\mathsf{so}} \mu_2$.

In the first case, from the definition of Φ_{src}, there exists another event μ_3 which ensures that $\mu_w \xrightarrow{E''.\mathsf{hb}} \mu$. In the second case, since $\mu_w \xrightarrow{E'.\mathsf{hb}_{\mathrm{SC}}} \mu_1$, from the inductive hypothesis, $\mu_w \xrightarrow{E'.\mathsf{hb}} \mu_1$. From Φ_{src}, we can then deduce that there must exist μ_3 which ensures that $\mu_w \xrightarrow{E''.\mathsf{hb}} \mu$. □

Theorem 5.3. Given a library implementation L and a robustness-preserving transformation ρ, if $\rho(L)$ is execution-graph robust, then L is induced subgraph robust.

Proof. Consider an execution $E \in \mathcal{CE}^L_{\mathrm{RC20}}$. By the definition of a robustness-preserving transformation, there exists $E' \in \mathcal{CE}^{\rho(L)}_{\mathrm{RC20}}$ such that the invocations, their ordering and the final values of all locations are the same in both executions. Further the execution graph G' in E' would also be an induced graph of G. Since $\rho(L)$ is execution-graph robust, $E' \in \mathcal{CE}^{\rho(L)}_{\mathrm{SC}}$. Finally, by the definition of a robustness-preserving transformation, an SC execution of $\rho(L)$ is also an SC execution of L, hence, $E' \in \mathcal{CE}^L_{\mathrm{SC}}$. Thus, E' is the SC execution of L which has the same invocations, ordering and final values of all locations as E. Hence, L is induced subgraph-robust. □

Lemma 5.5. The transformation ρ_{CAS} is a robustness-preserving transformation.

Proof. Let L be a library implementation, and let $L' = \rho_{CAS}(L)$. We first prove condition-1. Consider an execution $E_1 = \langle t, \Gamma, G \rangle \in \mathcal{CE}^L_{\mathrm{RC20}}$. We will use induction on the length of the trace t to construct an execution $E_2 \in \mathcal{CE}^{L'}_{\mathrm{RC20}}$ which obeys the constraints of condition-1. For a prefix $E_1^{'}$ of E_1, the inductive invariant we will use is:

$$\exists E_2^{'} \in \mathcal{E}^{L'}_{\mathrm{RC20}}.((\forall \mu_w \in E_1^{'}.M.\ \mathsf{act}(\mu_w) \in \{\mathsf{W}, \mathsf{U}\} \Rightarrow \mu_w \in E_2^{'}.M)$$
$$\wedge \forall \gamma \in E_1^{'}.\Gamma.\ \gamma \in E_2^{'}.\Gamma \wedge E_2^{'}.G \text{ is an induced subgraph of } E_1^{'}.G$$

The base case corresponding to trace of length 0 is straightforward as both $E_1^{'}.M$ and $E_1^{'}.\Gamma$ are empty, while both $E_1^{'}.G$ and $E_2^{'}.G$ consists of the initialization events. Consider a prefix $E_1^{'}$ and the

next event μ. If μ is a local event or an invocation event, then the condition holds trivially. Since ρ_{CAS} only removes read events from `while` loops, it is clear that if μ is a write/update event, from the inductive invariant, the condition can be directly established by simply performing the same event μ from the trace corresponding to $E_2^{'}$. Note that since read events which lead to a write event to a global location/return value are preserved by the transformation ρ_{CAS}, the same write event μ can be performed in execution $E_2^{'}$, since the values of the local variables which contribute to the write event would be the same.

If μ is a read event which is removed by the transformation (i.e. if μ corresponds to an event in any iteration-but-the-last), then μ is not performed in $E_2^{'}$. If μ is a read event which is preserved by the transformation, suppose $\mu_w \xrightarrow{E_1^{'}.\mathsf{rf}} \mu$. Let $\mathsf{sess}(\mu) = s$. We know by the inductive invariant that $\mu_w \in E_2^{'}.M$. For μ to occur in $E_2^{'}$, we have to show that $\neg(\mu_w \xrightarrow{E_2^{'}.\mathsf{mo};E_2^{'}.\mathsf{hb}^?} [E_2^{'}.M]_s)$. Since $E_2^{'}$ may not contain read events in session s which would be present in $E_1^{'}$, $E_2^{'}.\mathsf{hb} \subseteq E_1^{'}.\mathsf{hb}_{|E_2^{'}.M}$. Since the mo relation would be the same in both the executions, and since $\neg(\mu_w \xrightarrow{E_1^{'}.\mathsf{mo};E_1^{'}.\mathsf{hb}^?} [E_1^{'}.M]_s)$, we have the required condition. Hence, μ can read from the same write event in $E_2^{'}$. This concludes the inductive step.

Condition-2 is straightforward to show, as given an execution $E \in \mathcal{CE}_{\mathrm{SC}}^{L'}$, the same execution E also belongs to $\mathcal{CE}_{\mathrm{SC}}^{L}$. □

Theorem 6.1. Given a set of libraries $\mathbb{L}$, if each library in $\mathbb{L}$ is execution graph robust, then all executions in $\mathcal{CE}_{\mathrm{RC20}}^{\mathbb{L}}$ are also execution graph robust.

Proof. First, consider the case with $\mathbb{L} = \{\mathcal{L}_1, \mathcal{L}_2\}$. We work with a single arbitrary execution belonging to $\mathcal{CE}_{\mathrm{RC20}}^{\mathbb{L}}$. It is labelled as $E = \langle t, \Gamma, G\rangle$ where t is a trace of $\Omega_{\mathbb{L}} \times \mathcal{MS}_{\mathrm{RC20}}$, Γ is the set of all method invocation events generated in the trace, and G is the execution graph in the final state of the trace. To prove that G is execution-graph robust, we assume the contrary i.e. G is not SC-consistent, and $G.\mathsf{hb}_{\mathrm{SC}}$ is not irreflexive. Thus, take a cycle C made up of $(\mathsf{so}, \mathsf{rf}, \mathsf{mo}, \mathsf{fr})$ edges ($\mathsf{hb}_{\mathrm{SC}} = (\mathsf{so}, \mathsf{rf}, \mathsf{mo}, \mathsf{fr})^+$) which has a minimal number of memory events (we call this an $\mathsf{hb}_{\mathrm{SC}}$ cycle for convenience). Now consider cases on the nature of this cycle (viewing relations on the events in G as labelled edges in a directed graph).

Case All the memory events in the cycle were generated by methods of $\mathcal{L}_1$ (this can be checked by looking at the trace: for any relevant memory event $\mu = \mathsf{Mem}\langle i, \tau, \alpha\rangle$ in a session τ, we locate it in the trace using the unique label i and note the method in the most recent invocation event γ). Clearly, this denotes an instance of non-robustness in the library $\mathcal{L}_1$, and we prove a contradiction by creating a new execution in $\Omega_{\mathcal{L}_1} \times \mathcal{MS}_{\mathrm{RC20}}$ which would generate a non-robust execution graph (i.e. having an $\mathsf{hb}_{\mathrm{SC}}$ cycle).

In particular, we construct $E' = \langle t', \Gamma', g'\rangle$, an execution belonging to $\mathcal{CE}_{\mathrm{RC20}}^{\mathcal{L}_1}$. The transitions in t' are taken to be, in order, exactly the transitions in t which correspond to invocations of a method of $\mathcal{L}_1$ or events generated by reductions due to statements corresponding to an invocation γ of af method of $\mathcal{L}_1$. This specifically excludes any fences added by the InvkF reduction, allowing the trace to be generated by $\Omega_{\mathcal{L}_1} \times \mathcal{MS}_{\mathrm{RC20}}$. For the state $\langle s, G\rangle$ at each step, the LTS state s is determined straightforwardly by the transition system where the environment now only contains variables local to methods of $\mathcal{L}_1$ and thus follows the states in t, except that EnvLocal of invocations within $\mathcal{L}_2$ are replaced by the relevant environment of the current invocation in each session. The partial execution graph at every step is a subgraph of the corresponding execution graph of the state in t, generated by taking only the memory events that were created by $\mathcal{L}_1$ methods. This graph is obviously valid and RC20-consistent since, at every step, an RC20 consistency violation

would have implied an RC20 inconsistency in the original graph in the trace t. Finally Γ' is the corresponding set of $\mathcal{L}_1$ method invocations and G' is the final execution graph obtained in the trace.

Now, G' must have the same $\mathsf{hb}_{\mathsf{SC}}$ cycle that was present in G since every edge was so, rf, mo or fr and since each of the memory events in the cycle are present, the mo and rf relations are between the same events and fr is derived from rf and mo.

Case All the memory events were generated by methods of $\mathcal{L}_2$. This case is symmetric.

Case The cycle consists of events of $\mathcal{L}_1$ as well as $\mathcal{L}_2$; this is the general case where fences are made use of. Pick any $\mathcal{L}_1$ event and and consider the next contiguous set of $\mathcal{L}_2$ events in the cycle followed by another $\mathcal{L}_1$ event (which may be the same). Suppose these are $e_1, \ldots, e_n$ between d_1 and d_2. Now since d_1 and e_1 are events of different libraries, they cannot access the same memory location and can thus only be related by a so edge; and by the well-fenced criterion, must include an SC-fence in order i.e. $d_1 \xrightarrow{\mathsf{so}} f_1 \xrightarrow{\mathsf{so}} e_1$. Note that f_1 is a representation for three memory events in so, namely those generated by the acquire fence, CAS and release fence. Similarly we must have $d_2 \xrightarrow{\mathsf{so}} f_2 \xrightarrow{\mathsf{so}} e_2$ where f_1 and f_2 are both SC-fences.

The U events in the fences f_1 and f_2 are totally ordered in hb, and we hereafter refer to this as an hb ordering on the fences as a whole. We prove a lemma 6.11 separately that thehb-ordering between the two must be $f_1 \xrightarrow{\mathsf{hb}} f_2$ (in particular, obeying the direction of the $\mathsf{hb}_{\mathsf{SC}}$ chain in $f_1 \to e_1 \to \cdots \to e_n \to f_2$) using the robustness of $\mathcal{L}_2$. This implies that the original cycle C can be modified to remove the $\mathcal{L}_2$ events, having $d_1 \xrightarrow{\mathsf{so}} f_1 \xrightarrow{\mathsf{hb}} f_2 \xrightarrow{\mathsf{so}} d_2$ i.e. $d_1 \xrightarrow{\mathsf{hb}} d_2$. We can similarly continue this process until all $\mathcal{L}_2$ events are removed from the cycle, replacing them with hb-synchronization.

The cycle C is now left with so, rf, mo, fr edges as well as hb edges which appeared through indirect synchronization; thus it is not always possible to retain the cycle after removing $\mathcal{L}_2$ events from G to obtain an induced subgraph. We instead construct a new execution of $\mathcal{L}_1$ where the relevant memory events appear in so instead of hb, recreating the $\mathsf{hb}_{\mathsf{SC}}$ cycle in an RC20-consistent execution and contradicting the robustness of $\mathcal{L}_1$.

Consider the order of the memory events d_i in C appearing in the trace t. In the new execution, the trace will again have all the transitions corresponding to events in methods of the library $\mathcal{L}_1$, in the same order; however we will change the session IDs of these events in order to move invocations to different threads from the original while still maintaining validity. In particular, we will move every single invocation event into its own thread, in order to remove any of the unnecessary synchronization through so, except in the instances we actually require to maintain so, which is exactly only between events in so and hb in the modified cycle C. This is what allows us to convert the hb edges back to so to gain a legitimate cycle without causing branching or other such conflicts.

As a preliminary, consider some invocation γ which has memory events in the cycle. Since all such possible events are ordered by so, it is easy to see that any γ can have atmost two memory events in the cycle, in immediate succession to each other through so.

The new trace t' is constructed from t as follows. Firstly, for every memory event transition in the trace, we can locate the invocation it was a part of, since it is uniquely identified by i (so we look at the Invk event γ current active in the same session τ). Suppose the infinite set of possible sessions is $\tau_1, \ldots$. We start by adding the first event in t which must be some method invocation, and thus eventually all the events of that particular method invocation, into the first session τ_1 in t' (by taking the same transition but replacing the session ID with τ_1). This changes the state from

the initial state to that in which the first invocation has occurred in one session which is equivalent to the first state reached after the event in t.

For each of the next events, we consider the invocation γ associated with it. If the event is not the invocation itself, then the invocation must already be present in some session and the event must have been generated in the same session. For adding a new invocation event γ, we check if there is some memory event associated with γ which is in the modified cycle C, and then whether the edge immediately preceding this event was so from a memory event of a different invocation say γ'. Therefore in the original execution, γ must have been called in the same session after γ', and similarly for the new execution E' we place the invocation event for γ in the same session as that of γ' (which must already have been placed, following the trace ordering of t). Again if a memory event associated with γ in the modified cycle was immediately preceded by an hb from an event of a different invocation γ', we place the invocation event of γ in the same session as that of γ'. γ' must already be present since the hb in the cycle was propagated through a pair of SC-fences: one placed after γ' and one before γ and thus appearing in that order in t. Crucially, since γ can have atmost one memory event so or hb after a different method's memory event in the cycle, there is never a conflict about which session to place γ into.

We need to determine the relations between the memory events in the execution graphs being created in the trace also. For every partial execution graph g' in t', the set of memory events is a subset of the set of memory events of some graph in t (and in fact, of the final execution graph in E, namely G). We set the relations in g' as follows: for every pair of memory events in g', uniquely identified as say i_1 and i_2 (since each event has an identifier), if $(i_1, i_2) \in G.\mathsf{rf}$ then $(i_1, i_2) \in g'.\mathsf{rf}$ and if $(i_1, i_2) \in G.\mathsf{mo}$ then $(i_1, i_2) \in g'.\mathsf{mo}$. As each memory event reads from the same access as in the original trace or writes the same values, within each invocation the control flow is unaffected and the local environment is simply derived from the corresponding environment in the original trace. We now need to show that the intermediate graphs g' are in fact well-formed execution graphs (i.e. rf, mo follow some basic properties) and are all consistent under RC20; all this will give us a valid trace of $\mathcal{L}_1$-only events. The well-formedness follows directly from the well-formedness of G for every criterion (so we do not go into detail). For consistency, we show that $g'.\mathsf{hb} \subseteq G.\mathsf{hb}$.

$g'.\mathsf{hb} \triangleq (g'.\mathsf{so} \cup g'.\mathsf{sw})^+$ and if $(a, b) \in \mathsf{so}$ in g' then in G we must have had $(a, b) \in (\mathsf{so} \cup \mathsf{hb})^+$ in G (since the only invocations in the same session are those which had events in the cycle related by so or hb). Thus $g'.\mathsf{so} \subseteq G.\mathsf{hb}$.

Any sw edge in g' is of the form $[g'.M_{\sqsupseteq \mathtt{rel}}]; ([g'.M_{\mathsf{F},\sqsupseteq \mathtt{rel}}]; g'.\mathsf{so})^?; g'.\mathsf{rf}^+; (g'.\mathsf{so}; [g'.M_{\mathsf{F},\sqsupseteq \mathtt{acq}}])^?; [g'.M_{\sqsupseteq \mathtt{acq}}]$. In the case where the form simplifies to $[g'.M_{\sqsupseteq \mathtt{rel}}]; g'.\mathsf{rf}^+; [g'.M_{\sqsupseteq \mathtt{acq}}]$ since the same rf edges were present in G, the sw edge in g' was also present in $G.\mathsf{sw}$. Similarly for fence synchronization through so edges, if the sw edge is of the form $f_{\sqsupseteq \mathtt{rel}} \xrightarrow{\mathsf{so}} a \xrightarrow{\mathsf{rf}^+} b \xrightarrow{\mathsf{so}} g_{\sqsupseteq \mathtt{acq}}$ where f and g are fences (which are events in invocations of $\mathcal{L}_1$, along with a and b) and the so edges in g' were between events in G which were also in so; then again we have the sw edge was also present in G between the same library events. However, the possibility remains that the so edge was added by converting hb edges in G (through synchronization outside $\mathcal{L}_1$) into so edges; we would have in G the $f \to a$ edge being some combination of $f \xrightarrow{\mathsf{so}} a_1 \xrightarrow{\mathsf{hb}} a_2 \to \ldots a_n \xrightarrow{\mathsf{hb/so}} a$ edges. We consider the last such hb edge introduced through external SC-fence synchronization. If $a_n \xrightarrow{\mathsf{hb}} a$ for memory events in $\mathcal{L}_1$, where the hb was added through the introduced SC-fences, then clearly in G the synchronization was in fact of the form $a_n \xrightarrow{\mathsf{so}} f_1 \xrightarrow{\mathsf{rf/hb}} f_2 \xrightarrow{\mathsf{so}} a$ where f_1 and f_2 were the fences added at the interface. Similarly, if $b \xrightarrow{\mathsf{so}} g$ in g' then we have $b \xrightarrow{\mathsf{so}} g_1 \xrightarrow{\mathsf{rf/hb}} g_2 \xrightarrow{\mathsf{so}} b_1 \xrightarrow{\mathsf{so}} b_2 \xrightarrow{\mathsf{hb}} b_3 \to \cdots \to b_n \to g$ where g_1 and g_2 were SC-fences in the interface (and b_1 as well as g being events of $\mathcal{L}_1$). In

particular, we have $f_2 \xrightarrow{\text{so}} a \xrightarrow{\text{rf}^+} b \xrightarrow{\text{so}} g_1$ and therefore $f_2 \xrightarrow{\text{sw}} g_1$ by the same pattern in G. (Here, SC-fences are of the form fence(acq); f.cas(0, 0, acqrel, rlx); fence(rel) and specifically the $\mathsf{F}\langle\mathtt{rel}\rangle_f \xrightarrow{\text{so}} a \xrightarrow{\text{rf}^+} b \xrightarrow{\text{so}} \mathsf{F}\langle\mathtt{acq}\rangle$ forms the relevant pattern.) Now for all the other hb and so edges, they were all included in G.hb so the sw edge in g' also corresponds to an hb edge in G.

Thus:

$$\begin{aligned} g'.\text{so} &\subseteq G.\text{hb} \\ g'.\text{sw} &\subseteq G.\text{hb} \\ (g'.\text{so} \cup g'.\text{sw})^+ &\subseteq G.\text{hb}^+ \\ g'.\text{hb} &\subseteq G.\text{hb} \end{aligned}$$

This relation allows us to declare RC20-consistency of g' in terms of RC20-consistency of G. The conditions are:

- $g'.\text{mo}; g'.\text{rf}^?; g'.\text{hb}^?$ is irreflexive: Since these are all subsets (as relations) of the original relation over G, any event in g' which showed this reflexivity would do the same in G, thus making G RC20-inconsistent (a contradiction).
- $g'.\text{fr}; g'.\text{rf}^?; g'.\text{hb}$ is irreflexive; $g'.\text{fr}; g'.\text{mo}$ is irreflexive: The same argument as above.
- $g'.\text{so} \cup g'.\text{rf}$ is acyclic: Again, $g'.\text{rf} \subseteq G.\text{rf}$. Also, $g'.\text{so}$ is a subset of $(\text{so} \cup \text{hb})^+$ in G but $a_1 \xrightarrow{\text{hb}} a_2$ is a sequence of either so edges or sw edges, where sw edges are again $\left(\xrightarrow{\text{so}}\right)^? \left(\xrightarrow{\text{rf}}\right)^+ \left(\xrightarrow{\text{so}}\right)^?$ so $g'.\text{so} \subseteq G.\text{so} \cup G.\text{rf}$ and thus cyclicity in g' would imply cyclicity in G.

Intuitively, this construction maintains consistency since it only allows more behaviours without restricting any; and it makes the validity of g' easier to construct by removing the necessity to maintain so edges between unimportant library events (which may interfere with the movement into new sessions). Thus we can construct all the intermediate execution graphs of the new trace t' and it is easy to see that under $\Omega_{\mathcal{L}_1} \times \mathcal{MS}_{\text{RC20}}$ the relevant transitions do in fact allow the chosen sequence of states (which include the local environments moved to the appropriate sessions).

The final execution $E' = \langle t', \Gamma', G' \rangle$ where G' is the final execution graph and Γ' is just the set of $\mathcal{L}_1$ invocation events in t'. Since all of the events of the modified cycle C are in G' (they are $\mathcal{L}_1$ events only); rf, mo and fr events are maintained while hb edges are converted to so; the hb_{SC} cycle is recreated in the $\mathcal{L}_1$-only system which is a contradiction to the robustness of $\mathcal{L}_1$.

Lemma 6.11 For a minimal hb_{SC} cycle which involves a contiguous sequence of events of $\mathcal{L}_2$ sandwiched between events of $\mathcal{L}_1$, the hb between the relevant fences at the outermost events must follow the direction of the chain.

Proof. That is, given the execution E with final graph G in the new semantics, whenever $d_1 \xrightarrow{\text{so}} f_1 \xrightarrow{\text{so}} e_1 \to \cdots \to e_n \xrightarrow{\text{so}} f_2 \xrightarrow{\text{so}} d_2$ for d_1, d_2 events of $\mathcal{L}_1$, $e_1, \ldots, e_n$ events of $\mathcal{L}_2$ and f_1, f_2 SC-fences placed by the modification in semantics; if $\mathcal{L}_2$ is robust we must have $f_1 \xrightarrow{\text{hb}} f_2$. To the contrary, assume $f_2 \xrightarrow{\text{hb}} f_1$ in order to obtain an hb_{SC} cycle $f_1 \xrightarrow{\text{so}} e_1 \to \cdots \to e_n \xrightarrow{\text{so}} f_2 \xrightarrow{\text{hb}} f_1$ i.e. $e_1 \to \cdots \to e_n \xrightarrow{\text{hb}} e_1$. This is a minimal modified cycle in $\mathcal{L}_2$ events only, and we go through exactly the same process as previously with $\mathcal{L}_2$ instead in order to obtain a contradiction to the robustness of $\mathcal{L}_2$. □

Finally, we extend this straightforwardly to the case where $\mathbb{L} = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$. We observe that we can consider any library, say $\mathcal{L}_1$, as the primary library and attempt to reduce any *hbsc* cycle in G to an execution in $\mathcal{L}_1$ events only by considering contiguous sequences of events in every other

library separately, unless one of the hb edges between the SC-fences is in a contradictory direction. In this case we get a modified cycle using events of that library only, and continue as normal. □

Theorem 6.2. Given a set of libraries $\mathbb{L}$, if each library in $\mathbb{L}$ is induced subgraph robust, then all executions in $C\mathcal{E}^{\mathbb{L}}_{\text{RC20}}$ are also induced subgraph robust.

Proof. Again suppose $\mathcal{L} = \{\mathcal{L}_1, \mathcal{L}_2\}$ and take a single arbitrary execution generated by $\Omega_{\mathbb{L}} \times \mathcal{MS}_{\text{RC20}}$, involving $\mathcal{L}_1$ and $\mathcal{L}_2$ which are induced subgraph robust, labelled $E = \langle t, \Gamma, G\rangle$. We need to construct a new execution $\hat{E} = \langle \hat{t}, \hat{\Gamma}, \hat{G}\rangle$ which has the same invocation events as E whose order is respected in the trace, and for which $\hat{G}$ is an induced subgraph of G.

We employ a similar strategy to the previous. Taking a minimal hb_{SC} cycle in E with respect to number of events in the cycle, we try to reduce it to a cycle in only $\mathcal{L}_1$ or $\mathcal{L}_2$ events in an execution of the corresponding transition system. Suppose we worked with $\mathcal{L}_1$. Now with this non-robust execution in $C\mathcal{E}^{\mathcal{L}_1}_{\text{RC20}}$, we can use induced subgraph robustness of $\mathcal{L}_1$ to get a new execution which maintains the ordering of invocation events in the trace and in which the events form an induced subgraph, but which is now robust. What we would like to do is mimic the pattern of calling of the events within the new execution's trace, back in the original trace t. Being able to retrofit the robust $C\mathcal{E}^{\mathcal{L}_1}_{\text{RC20}}$ in this way would allow us to generate an execution in $C\mathcal{E}^{\mathbb{L}}_{\text{RC20}}$ which is an induced subgraph of the original execution, but missing the hb_{SC} cycle we were working with. We can then repeat this process to remove hb_{SC} cycles one by one since new cycles are not introduced.

Consider such a minimal hb_{SC} cycle C in E. The first thing we do is locate sequentially in C, every successive pair of events a and b such that a was generated by a method of $\mathcal{L}_1$, and b by a method of $\mathcal{L}_2$. As previously, the edge between a and b must be so and by well-fencedness, they must be separated by an SC-fence f i.e. $a \xrightarrow{\text{so}} f \xrightarrow{\text{so}} b$. We insert this fence f into the cycle between a and b. We continue to do this between every pair of events in the cycle.

At the end, we obtain the modified cycle C in which every contiguous sequence of $\mathcal{L}_1$ events, bounded by $\mathcal{L}_2$ events, is also bounded by fences (and vice versa). Now in the execution E, the U events within the fences are totally ordered in hb so any pair of the SC-fences has some hb ordering between them.

Consider, in the total ordering in hb over the fences, the first fence present in the cycle say f_1. We follow the outgoing edge of f_1 to the next event say of library $\mathcal{L}_2$, and continue along successive events say $e_1 \to \cdots \to e_n$ until we reach a second fence f_2 (clearly if there is a fence, there at at least 2 SC-fences in C, with the second placed after the contiguous sequence of $\mathcal{L}_2$ events). Consider the ordering between f_1 and f_2: If it is in the direction $f_2 \xrightarrow{\text{hb}} f_1$, then we have a cycle in $\mathcal{L}_2$ events only i.e. $f_1 \to e_1 \to \cdots \to e_n \to f_2 \xrightarrow{\text{hb}} f_1$ i.e. $e_1 \to \cdots \to e_n \xrightarrow{\text{hb}} e_1$ which involves hb as well. Therefore we can now perform the same procedure as previously, in order to create an execution using $\mathcal{L}_2$ events only, in which the hb_{SC} cycle is recreated and the order of the events of $\mathcal{L}_2$ in t are maintained using so, rf, mo, fr, only; call it $E' = \langle t', \Gamma', G'\rangle$. The case in which the hb_{SC} cycle was involving events of one library only is now subsumed by this.

Now if the hb between the fences was in the direction of the hb_{SC} chain i.e. $f_1 \xrightarrow{\text{hb}} f_2$, we would locate the next fence successively until we found a pair of fences which were in the opposite direction. This would allow us to create again an execution using events of one library only in which the hb_{SC} cycle is recreated. The remaining case in which such a pair is never found. Suppose there are m fences $f_1, \ldots, f_m$, and we have $f_1 \xrightarrow{\text{hb}} f_2 \xrightarrow{\text{hb}} \cdots \xrightarrow{\text{hb}} f_m$. There is a contiguous set of events say $e_1, \ldots, e_n$ of a single library ($\mathcal{L}_1$, if the library between f_1 and f_2 was $\mathcal{L}_2$) in the cycle C following f_m and ending with f_1, then we have $f_1 \xrightarrow{\text{hb}} f_m \xrightarrow{\text{so}} e_1 \to \cdots \to e_n \xrightarrow{\text{so}} f_1$ i.e.

$e_1 \rightarrow \cdots \rightarrow e_n \xrightarrow{\mathsf{hb}} e_1$ (where as usual, the hb is between the update events generated by the fences). We can thus again make an execution E' using events of that library only, which preserves the $\mathsf{hb}_{\mathsf{SC}}$ cycle and the order of $\mathcal{L}_1$ events in t.

Without loss of generality, suppose we are working with a cycle in library $\mathcal{L}_1$. From the property of induced subgraph robustness of $\mathcal{L}_1$ we get a different execution of $\mathcal{L}_1$, $\hat{E}' = \langle \hat{t}', \hat{\Gamma}', \hat{G}' \rangle$ which is robust and in which $\hat{G}'$ is an induced subgraph of G'. We need to be able to integrate this execution back into E, in order to obtain a new execution $\hat{E}$ which does not contain the $\mathsf{hb}_{\mathsf{SC}}$ cycle C.

We follow the original trace t to construct $\hat{t}$. Note that the order of invocations of $\mathcal{L}_1$ events in both the traces t and $\hat{t}'$ is exactly the same. Intuitively, whenever we obtain an $\mathcal{L}_1$ event in a session, we will follow the transitions in $\hat{t}'$ with respect to adding non-memory events belonging to that session, advancing the local state until some other memory event in $\hat{t}'$ is reached (which represents potential conflicts with t, and thus has to be added according to the order in t).

If the first transition of t involves an invocation of $\mathcal{L}_2$ in some session τ, we follow the same transition in the new trace $\hat{t}$, invoking the corresponding $\mathcal{L}_2$ event and following the states in the trace t. For successive $\mathcal{L}_2$ events and SC-fences, we continue in the same manner. At this stage, the local environments and partial execution graphs $\langle \varrho, G \rangle$ making up the state in $\hat{t}$ follows the state in t. On encountering an $\mathcal{L}_1$ invocation in t say γ, we place it in the appropriate session and consider $\hat{t}'$. The same invocation appeared at some position in $\hat{t}'$ in some session τ_1, and we follow $\hat{t}'$ adding transitions in $\hat{t}'$ which correspond to non-memory events i.e. change only the local state of the environment in τ_1 and ignoring events which did not take place in τ_1. We continue to do this until we reach a memory event in τ_1 (or the invocation of γ ends in τ_1). At this point we return to t where we left. This is possible since the validity of all of these transitions only depends on the local environment of τ_1 and is thus synchronized with the local environment of τ_1 in $\hat{t}'$, and does not make any changes to the partial execution graph of the states in $\hat{t}$.

For local transitions and memory events generated by invocations of $\mathcal{L}_2$, we continue to follow t until we again encounter an invocation or memory event of a method of $\mathcal{L}_1$. Now if this memory event say e does not appear in the $\mathcal{L}_1$-only trace $\hat{t}'$ (which we can check using the unique identifiers of the memory events) then we ignore it and continue, unless it is an SC fence introduced artificially due to the InvkF rule. In this case we follow the trace t add the relevant event. Otherwise the memory event appears in $\hat{t}'$, and we look at the relevant session in $\hat{E}'$ say τ_2 again. In this session, we have inductively conducted local transitions so that the next memory event in τ_2 is available, and by the property of $\hat{G}'$ being a subgraph of G, $\hat{E}'$ inherits the total order of memory events in so within τ_2 from the corresponding session in E. Therefore it is possible to take the transition corresponding to adding the memory event encountered, and we need to make sure that the rf, mo edges to be added in the partial execution graph synchronize with the same memory events as in $\hat{t}'$. This is possible since we are following the order of events in t, and the event e must synchronize with the same events it did in E using the induced subgraph property. Thus the transition in $\hat{t}'$ can be taken in $\hat{t}$, and we again perform local transitions in the same session τ_2 until the next memory event before returning to t.

Once all the transitions in t have been taken, we have an execution $\hat{E}$ which is a complete execution since all the events corresponding to $\mathcal{L}_2$ events were performed in order while for $\mathcal{L}_1$ events, all the memory events in $\hat{t}'$ were present in t and thus taken while local events were taken afterwards. At each step, the local state corresponds to an environment of a state in either t or $\hat{t}'$ depending on the active invocation while the partial execution graph is a subgraph of the original execution graph G. The final execution graph $\hat{G}$ is obtained from the final state of the trace and the set of invocation events clearly follows that in Γ. However in $\hat{E}$, the $\mathsf{hb}_{\mathsf{SC}}$ cycle that we originally considered can no longer be present, since the chain that was in $\mathcal{L}_1$ events has been removed by

the robustness of $\hat{E}'$ (explicitly, consider the two events e_1 and e_2 generated by $\mathcal{L}_1$ methods in hb through SC-fence synchronization. These were in so in the execution $\hat{E}'$, so any $\mathsf{hb}_{\mathsf{SC}}$ chain from e_2 to e_1 cannot have been present in $\hat{E}'$ since this would contradict its robustness. Now any $\mathsf{hb}_{\mathsf{SC}}$ chain in $\hat{E}$ from e_2 to e_1 must be reflected in $\hat{E}'$ and therefore cannot exist. Thus the original minimal $\mathsf{hb}_{\mathsf{SC}}$ cycle, which contained the $e_2 \rightarrow \cdots \rightarrow e_1$ events, must not exist in $\hat{E}$).

The process above allows us to take an execution in $C\mathcal{E}^{\mathbb{L}}_{\mathsf{RC20}}$ and generate an execution in which at least one of the $\mathsf{hb}_{\mathsf{SC}}$ cycles has been removed and in which the final execution graph is an induced subgraph of the original. It also does not add any, since any cycle present in $\hat{G}$ must have been present in G also; thus we can repeat this process a finite amount of times until the resulting execution $\hat{E}$ has a graph which has no $\mathsf{hb}_{\mathsf{SC}}$ cycles at all, and is in fact robust. Thus we obtain for any complete execution E of $\Omega_{\mathbb{L}} \times \mathcal{MS}_{\mathsf{RC20}}$, a corresponding robust execution $\hat{E}$ which proves that all executions in $C\mathcal{E}^{\mathbb{L}}_{\mathsf{RC20}}$ are induced subgraph robust. □

C Example demonstrating the need for induced subgraph robustness

```
method m1(z1) = store(ℓ2, z1, rlx)
method m2(z2) = while true do
                  x1 = load(ℓ1,rlx)
                  y1 = load(ℓ2,rlx)
                  if cas(ℓ1, x1, y1,acqrel))
                  then return x1 else skip
```

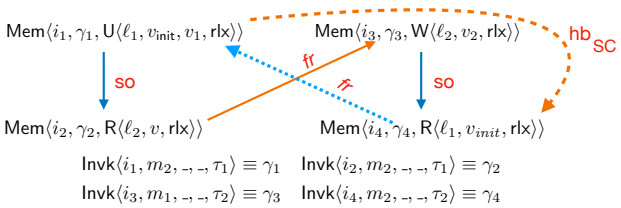

Fig. 13. A non-robust but benign execution of a library implementation L. For simplicity, we directly record the shared memory locations ℓ_1 and ℓ_2 accessed by the methods in the program text. Only relevant edges are shown in the execution. The dashed blue edge captures a benign dependency that does not contribute to the final return value of its corresponding source invocation.

To illustrate the need of induced subgraph-robustness for library implementations, consider the implementation L shown in Figure 13, which is synthetically constructed but exhibits an execution pattern common in existing implementations. The implementation consists of two methods m_1 and m_2, and involves accesses to two global locations ℓ_1 and ℓ_2. The method m_1 simply stores its input argument to the location ℓ_1, while the method m_2 speculatively reads ℓ_1 and ℓ_2, and updates ℓ_1 through a CAS operation. Next to the implementation, we show a snippet of an execution of this library involving 4 different invocations $(\gamma_1, \gamma_2, \gamma_3, \gamma_4)$ across two sessions (τ_1, τ_2). For clarity, we have also indicated the invocation of each memory event in its description. This execution is an instance of the classical store-buffering anomaly, manifesting here through interactions between events in multiple invocations. In this execution, invocation γ_1, executed by session τ_1, successfully performs the CAS operation on ℓ_1 in method m_2. Subsequently, invocation γ_2 initiates another call to m_2 in session τ_1 that performs a `load` on ℓ_2. This load does *not* read from the store performed by invocation γ_3 in session τ_2 , resulting in an fr edge between these two events. Combined with obvious session ordering among operations in different invocations executing within the same session, these edges establish a path that induces an $\mathsf{hb}_{\mathsf{SC}}$ ordering between the store to ℓ_1 by the CAS operation in γ_1 and the store to ℓ_2 by γ_3. However, there is no hb ordering between these two events, and hence using the terminology from the previous section, this is a potential non-robustness witness involving the location ℓ_1. This potential witness will turn into an actual witness by the subsequent read to ℓ_1 by invocation γ_4, which reads initializing value v_{init}, ignoring the value v_1.

Neither Φ_{src} nor Φ_{dst} can be established for this execution, and hence we may conclude that this implementation is not robust. However, even though this execution is non-robust, its behavior is actually not problematic. This is because the CAS operation that subsequently follows the load of ℓ_1 in γ_4 would fail since ℓ_1 has been updated by the earlier CAS in invocation γ_1 (two update operations cannot read the same value). Consequently, the non-robust read in invocation γ_4 would be ignored since the loop, when re-executed, would initiate a reload of location ℓ_1. In other words, the non-robust read event does not effect the shared state or the return value of invocations in any fashion. In fact, only the read event in the last iteration affects the return value of the invocation, and this event is guaranteed to be robust, due to the fact that its behavior has to match the behavior of the final succeeding CAS. *Induced subgraph robustness* allows us to ignore all such non-robust reads, and in this particular case, consider only the robustness of events in the last iteration of the loop, allowing us to conclude that the implementation is effectively robust. This illustrates a pattern common in many implementations: *synchronizing CAS operations ensure robustness of events that actually affect observable library behavior.*

D SMT Encoding Details

Our encoding uses finite uninterpreted domains $\mathbb{E}$, $\mathbb{V}$ and $\mathbb{I}$ to represent memory events, values and invocations, resp. Uninterpreted functions TYP, LOC, WVAL, RVAL : $\mathbb{E} \to \mathbb{V}$ are used to encode various memory event properties such as access type, access location, read and write values respectively. Binary predicates so, mo, rf, fr, sw : $\mathbb{E} \times \mathbb{E} \to \mathbb{B}$ relate various memory events, as discussed earlier. We also define predicates hbsc and hb, corresponding to the derived relations $\mathsf{hb}_{\mathsf{SC}}$ and hb.

Let Ψ_{base} denote the encoding of various well-formedness constraints that must be obeyed by the binary relations between events across all executions. Ψ_{base} is a direct encoding of the χ_{base} constraints mentioned in Section 3.2. Ψ_{SC} encodes the constraint that $\mathsf{hb}_{\mathsf{SC}}$ is irreflexive.

Along with memory events, the encoding also instantiates invocations from the domain $\mathbb{I}$. Functions ARG, RET : $\mathbb{I} \to \mathbb{V}$ are used to denote the arguments and return values corresponding to an invocation and METH : $\mathbb{I} \to \mathbb{M}$ assigns a method type to each invocation. Further, the predicates $\mathsf{so}_I, \mathsf{hb}_I : \mathbb{E} \times \mathbb{E} \to \mathbb{B}$ are used to encode session order and happens-before ($\mathsf{hb}_{\mathsf{inv}}$) relations between invocations.

To encode the relation between invocations and memory events, we first introduce some notation. Assuming the library implementation L consists of methods $\mathbb{M}$, $L(m, l)$ yields the program statement that has *label* l in method $m \in \mathbb{M}$. Let $Labels^L(m)$ denote the set of labels present in the implementation of method m in L. Additionally, while an implementation can allocate and access an unbounded number of locations on the heap during executions, to generate our encoding, we

use a fixed, finite number of location classes. We have a location class for every shared global variable and every field type (for nodes allocated on the heap). Given a location class ℓ, we use the notation $\mathtt{Acc}^L(\ell)$ to denote all possible set of pairs (m, l) such that the expression with label l in method m accesses ℓ. Similarly, $\mathtt{Acc}^L_{\mathtt{W}}(\ell)$ denotes the set of labels that perform a store operation on ℓ. Let $\mathsf{LocClass}_L$ denote the set of all location classes in L. For example, the location classes for the register implementation of §2 in the main paper are `L`, `val`, while $\mathtt{Acc}^L(\mathtt{val}) = \{(\mathtt{set}, 3), (\mathtt{get}, 9)\}$, if we interpret line numbers to serve as labels. Note that even though there is only one location class per field type, accesses to the same field in different node instances are encoded as accesses to different locations. Given an invocation $i \in \mathbb{I}$ of method type $m = \textsc{meth}(i)$, the encoding $\Psi^L_m(i, l)$ instantiates the memory event $\eta(i, l)$ generated by the label l in method m. This encoding essentially assigns the location, access type, read/write value, etc. for the event $\eta(i, l)$. For example,

Function Check-$\Phi_{src}(\Psi, L, \ell)$:
  foreach $(m, l) \in \mathsf{Acc}^L_{\mathsf{W}}(\ell)$ **do**
    foreach $\ell' \in \mathsf{LocClass}(L) \wedge (m_1, l_1), (m_2, l_2) \in \mathsf{Acc}^L(\ell')$ **do**
      $\varphi \leftarrow \exists i_w, i_1, i_2 \in \mathbb{I}.\ \Psi^L_m(i_w) \wedge \Psi^L_{m_1}(i_1) \wedge \Psi^L_{m_2}(i_2) \wedge \mathsf{hb}(\eta(i_w, l), \eta(i_1, l_1)) \wedge (\mathsf{rf} \vee \mathsf{fr} \vee \mathsf{mo})(\eta(i_1, l_1), \eta(i_2, l_2)) \wedge \neg\mathsf{hb}(\eta(i_w, l), \eta(i_2, l_2))$
      if CheckSAT$(\Psi \wedge \varphi)$ **then**
        return FALSE
      end
    end
  end
  return TRUE
End Function

Fig. 14. Algorithm to check Φ_{src}

the following rule describes the encoding of a `store` operation:

$$\frac{L(m, l) = \mathtt{store}(x, e, o_{\mathsf{W}})}{\Psi^L_m(i, l) \triangleq \exists \mu \in \mathbb{E}.\ \textsc{typ}(\mu) = \mathsf{W} \wedge \textsc{loc}(\mu) = \mathsf{LocClass}(x) \wedge \textsc{wval}(\mu) = [\![e]\!] \wedge \eta(i, l) = \mu}$$

Here, $\mathit{LocClass}(x)$ gives the location class for x, while $[\![e]\!]$ denotes the encoding of the expression e. For operations inside the body of if-then-else statements or while-statements, we also encode the corresponding conditionals in Ψ^L_m.

The encoding of an invocation consists of encoding of all the statements in the method: $\Psi^L_m(i) = \bigwedge_{l \in Labels^L(m)} \Psi^L_m(i, l)$. Figure 14 presents the algorithm to check Φ_{src} for a given location class ℓ in a library implementation L. Ψ contains constraints from Ψ_{base} and Ψ_{SC} along with the derived constraints. Informally, any execution of the library L must obey the constraints in Ψ.

Check-Φ_{src} then considers every program statement writing to location class ℓ, and constructs the FOL encoding of $\neg\Phi_{src}$ in the formula φ. Recall that Φ_{src} involves a write event to ℓ, along with two events (both related by either rf, mo or fr) to the same location. φ instantiates these events, along with every other event in the enclosing invocations (using the Ψ^L_m formulae). Referring back to the notation used in Table 1 in the main paper, $\eta(i_w, l)$ takes the place of μ_w, while $\eta(i_1, l_1), \eta(i_2, l_2)$ take the place of μ_1, μ_2 resp. φ asserts the antecedent of Φ_{src} and $\neg\mu_w \xrightarrow{\mathsf{hb}} \mu_2$. An SMT solver is then called with the conjunction of φ and Ψ to check its satisfiability. φ is constructed for all possible writes to ℓ and every combination of events μ_1 and μ_2, and if all CheckSAT calls are unsatisfiable, we conclude Φ_{src} for ℓ. Referring back to Figure 3 in the main paper, the various cases considered in that figure correspond to the various SMT queries generated by applying Check-Φ_{src} on the register library implementation.

In a similar manner, we can construct the encodings for checking Φ_{dst} as shown by the algorithm in Fig 15. We can now check Φ_{src} and Φ_{dst} for every location class, and if they hold, conclude that the implementation is robust.

Figure 15 describes our encoding for checking violations of Φ_{dst} involving the location class ℓ in library implementation L. Following the notation used in the definition of Φ_{dst} in Table 1 in the main paper, the algorithm consider every write access to location ℓ ($\mu_w = \eta(i_w, l_w)$) and every other access to ℓ ($\mu_2 = \eta(i_2, l_2)$). The antecedent of Φ_{dst} is simplified and encoded as $\mathsf{hb}_{\mathsf{SC}}(\eta(i_w, l_w), \eta(i_2, l_2))$. The negation of the consequent is encoded by considering every event before $\eta(i_2, l_2)$ in the invocation i_2, and asserting that $\eta(i_w, l_w)$ is not in hb order to it.

D.1 Details of derived constraints

As discussed in §6, our analysis also generates derived constraints to aid in verification. These can be broadly classified into two classes: (1) *program structure* constraints that are derived by a static analysis of the implementation and (2) *specification* constraints that are directly obtained from

Function Check-$\Phi_{dst}(\Psi, L, \ell)$**:**
  foreach $(m_w, l_w) \in \mathsf{Acc}^L_{\mathtt{W}}(\ell)$ **do**
    foreach $(m_2, l_2) \in \mathsf{Acc}^L(\ell)$ **do**
      $\varphi \leftarrow \exists i_w, i_2 \in \mathbb{I}.\ \Psi^L_m(i_w) \wedge \Psi^L_{m_2}(i_2) \wedge \mathsf{hb}_{\mathsf{SC}}(\eta(i_w, l_w), \eta(i_2, l_2)) \wedge$
       $\bigwedge_{l \in Labels^L(m_2)} \mathsf{so}(\eta(i_w, l_w), \eta(i_2, l)) \Rightarrow \neg\mathsf{hb}(\eta(i_w, l_w), \eta(i_2, l))$
      if CheckSAT$(\Psi \wedge \varphi)$ **then**
        return FALSE
      end
    end
  end
  return TRUE
End Function

Fig. 15. Algorithm to check Φ_{dst}

the specification of the implemented data structure. Assuming that the library implementation is correct under SC, we can directly assume the specification constraints. We reap the benefits here of basing our verification procedure on searching non-robustness witnesses across SC executions and perhaps surprisingly, we find that these constraints are very effective in pruning false positives. We now provide more details about the constraints:

Access Constraints. These are program structure constraints that deal with locations allocated on the heap, which may be written to at most once in any execution, or are always written within the same invocation, thus ensuring a so relation between all such write events. For example, a *unique write* access constraint to a location class ℓ is encoded as follows:

$$\forall \mu_1, \mu_2.\ \mathsf{loc}(\mu_1) = \mathsf{loc}(\mu_2) = \ell \wedge \mathsf{type}(\mu_1) = \mathsf{type}(\mu_2) = \mathtt{W} \Rightarrow \mu_1 = \mu_2$$

We find that such constraints are common in library implementations that dynamically allocate memory on the heap, and we derive these constraints using a simple static analysis that looks for store operations to locations dynamically allocated using `new()`. As an example, in the register implementation, the *unique write* access constraint holds for the field `val`.

CAS Constraints: These are program structure constraints that deal with locations exclusively modified using `acqrel` CAS operations. Due to CAS semantics, we can then guarantee that the hb relation among all the update events to the location is total. Intuitively, this happens because every update to the location must then read from another update, and because two updates cannot read from the same CAS (U) event, this results in a total order among all updates. This constraint is encoded as follows:

$$\forall \mu_1, \mu_2.\ \mathsf{loc}(\mu_1) = \mathsf{loc}(\mu_2) = \ell \wedge \mathsf{type}(\mu_1) = \mathsf{type}(\mu_2) = \mathtt{U} \Rightarrow \mu_1 = \mu_2 \vee \mathsf{hb}(\mu_1, \mu_2) \vee \mathsf{hb}(\mu_2, \mu_1)$$

As an example, the CAS constraint holds for the location ℓ_1 in Figure 13.

Specification constraints: Since we assume the library implementation is correct under SC, these constraints are directly obtained from the axiomatic, declarative specification of the implemented data structure. These specifications only deal with the method type, argument or return values of invocations as well as their happens-before ordering. We use the specifications for stack, queue, set and register data structures as defined in previous works ([23, 25]).

Linearization point constraints. Optionally, if the correctness specifiction is equivalent to linearizability, we can also derive useful constraints from the linearization points involved in the proof of linearizability. In linearizability proofs, every method implementation is statically associated with a linearization point (LP), which is nothing but a program statement where an invocation of the method takes actual effect. It is then natural that there would be some correlation between the

$\mathsf{hb}_{\mathsf{inv}}$ ordering between invocations and the hb ordering between the events corresponding to their LPs. We leverage this observation to determine hb relations that must exist between the LPs of different invocations. The constraints are derived from specification constraints by replacing the $\mathsf{hb}_{\mathsf{inv}}$ order between invocations with the hb order between LPs of invocations.

Received 2024-04-06; accepted 2024-08-18