Library Abstraction for C/C++ Concurrency

Mark Batty
University of Cambridge

Mike Dodds
University of York

Alexey Gotsman
IMDEA Software Institute

Abstract

When constructing complex concurrent systems, abstraction is vital: programmers should be able to reason about concurrent libraries in terms of abstract specifications that hide the implementation details. Relaxed memory models present substantial challenges in this respect, as libraries need not provide sequentially consistent abstractions: to avoid unnecessary synchronisation, they may allow clients to observe relaxed memory effects, and library specifications must capture these.

In this paper, we propose a criterion for sound library abstraction in the new C11 and C++11 memory model, generalising the standard sequentially consistent notion of linearizability. We prove that our criterion soundly captures all client-library interactions, both through call and return values, and through the subtle synchronisation effects arising from the memory model. To illustrate our approach, we verify implementations against specifications for the lock-free Treiber stack and a producer-consumer queue. Ours is the first approach to compositional reasoning for concurrent C11/C++11 programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Verification, Concurrency, Modularity, C, C++

1. Introduction

Software developers often encapsulate functionality in libraries, and construct complex libraries from simpler ones. The advantage of this is information hiding: the developer need not understand each library's implementation, but only its more abstract *specification*. On a sequential system, a library's internal actions cannot be observed by its client, so its specification can simply be a relation from initial to final states of every library invocation. This does not suffice on a concurrent system, where the invocations can overlap and interact with each other. Hence, a concurrent library's specification is often given as just another library, but with a simpler (e.g., atomic) implementation; the two libraries are called *concrete* and *abstract*, respectively. Validating a specification means showing that the simpler implementation *abstracts* the more complex one, i.e., reproduces all its client-observable behaviours. Library abstraction has to take into account a variety of ways in which a client and library can interact, including values passed at library calls and returns, the contents of shared data structures and, in this paper, the *memory model*.

The memory model of a concurrent system governs what values can be returned when the system reads from shared memory. In a traditional *sequentially consistent (SC)* system, the memory model is straightforward: there is a total order over reads and writes, and each read returns the value of the most recent write to the location being accessed [15]. However, modern processors and programming languages provide *relaxed memory models*, where there is no total order of memory actions, and the order of actions observed by a thread may not agree with program order, or with that observed by other threads.

In this paper, we propose a criterion for library abstraction on the relaxed memory model defined by the new ISO C11 [12] and C++11 [13] standards (henceforth, the 'C11 model'). We handle the core of the C11 memory model, leaving more esoteric features, such release-consume atomics and fences, as future work (see §9). The C11 model is designed to support common compiler optimisations and efficient compilation to architectures such as x86, Power, ARM and Itanium, which themselves do not guarantee SC. It gives the programmer fine-grained control of relaxed behaviour for individual reads and writes, and is defined by a set of axiomatic constraints, rather than operationally. Both of these properties produce subtle interactions between the client and the library that must be accounted for in abstraction.

Our criterion is an evolution of *linearizability* [5, 7, 10, 11], a widely-used abstraction criterion for non-relaxed systems. Like linearizability, our approach satisfies the *Abstraction Theorem*: if one library (a specification) abstracts another (an implementation), then the behaviours of any client using the implementation are contained in the behaviours of the client using the specification. This result allows complex library code to be replaced by simpler specifications, for verification or informal reasoning. Hence, it can be viewed as giving a proof technique for contextual refinement that avoids considering all possible clients. Our criterion is compositional, meaning that a library consisting of several smaller non-interacting libraries can be abstracted by considering each sub-library separately. When restricted to the SC fragment of C11, our criterion implies classical linearizability (but not vice versa).

The proposed criterion for library abstraction gives the first sound technique for specifying C11 and C++11 concurrent libraries. To justify its practicality, we have applied it two typical concurrent algorithms: a non-blocking stack and an array-based queue. To do this, we have adapted the standard linearization point technique to the axiomatic structure of the C11 model. These case studies represent the first step towards verified concurrent libraries for C11 and C++11.

Technical challenges. Apart from managing the mere complexity of the C11 model, defining a criterion for library abstraction

POPL'13, January 23–25, 2012, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1832-7/13/01. . . $10.00

requires us to deal with several challenges that have not been considered in prior work.

First, the C11 memory model is defined axiomatically, whereas existing techniques for library abstraction, such as linearizability, have focused on operational trace-based models. To deal with this, we propose a novel notion of a *history*, which records all interactions between a client and a library. Histories in our work consist of several partial orders on call and return actions. This is in contrast to variants of linearizability, where histories are linear sequences (for this reason, in the following we avoid the term 'linearizability'). We define an abstraction relation on histories as inclusion over partial orders, and lift this relation to give our abstraction criterion for libraries: one library abstracts another if any history of the former can be reproduced in abstracted form by the latter.

Second, C11 offers the programmer a range of options for concurrently accessing memory, each with different trade-offs between consistency and performance. These choices can subtly affect other memory accesses across the client-library boundary—a particular choice of consistency level inside the library might force or forbid reading certain values in the client, and vice versa. This is an intended feature: it allows C11 libraries to define synchronisation constructs that offer different levels of consistency to clients. We propose a method for constructing histories that captures such client-library interactions uniformly. The Abstraction Theorem certifies that our histories indeed soundly represent *all* possible interactions.

Finally, some aspects of the C11 model conflict with abstraction. Most problematically, the model permits *satisfaction cycles*. In satisfaction cycles, the effect of actions executed down a conditional branch is what causes the branch to be taken in the first place. This breaks the straightforward assumption that faults are confined to either client or library code: a misbehaving client can cause misbehaviour in a library, which can in turn cause the original client misbehaviour! For these reasons, we actually define *two* distinct library abstraction criteria: one for general C11, and one for a language without the feature leading to satisfaction cycles. The former requires an a priori check that the client and the library do not access each others' internal memory locations, which hinders compositionality. The latter lifts this restriction (albeit for a C11 model modified to admit incomplete program runs) and thus provides evidence that satisfaction cycles are to blame for non-compositional behaviour. Our results thus illuminate corner cases in C11 that undermine abstraction, and may inform future revisions of the model.

As we argue in §9, many of the techniques we developed to address the above challenges should be applicable to other models similar to C11.

Structure. In the first part of the paper, we describe informally how algorithms can be expressed and specified in the C11 memory model (§2), and our abstraction criteria (§3). We then present the model formally (§4 and §5), followed by the criteria (§6) and a method for establishing their requirements (§7). Proofs are given in an extended version of the paper [1, §C].

2. C11 Concurrency and Library Specification

In this section we explain the form of our specifications for C11 concurrent libraries, together with a brief introduction to programming in the C11 concurrency model itself. As a running example, we use a version of the non-blocking Treiber stack algorithm [22] implemented using the concurrency primitives in the subset of C11 that we consider. Figure 1a shows its specification, and Figure 1b its implementation, which we have proved to correspond (§7 and [1, §E]). For readability, we present examples in a pseudocode instead of the actual C/C++ syntax. Several important features are highlighted in red—these are explained below.

SPECIFICATION:

```
atomic Seq S;




void init() {
  store_REL(&S,empty);
}

void push(int v) {
  Seq s, s2;
  if (nondet()) while(1);
  atom_sec {
    s = load_RLX(&S);
    s2 = append(s,v);
    CAS_RLX,REL(&S,s,s2);
  }
}


int pop() {
  Seq s;
  if (nondet()) while(1);
  atom_sec {
    s = load_ACQ(&S);
    if (s == empty)
      return EMPTY;
    CAS_RLX,RLX(&S,s,tail(s));
    return head(s);
  }
}
```

(a)

IMPLEMENTATION:

```
struct Node {
  int data;
  Node *next;
};
atomic Node *T;

void init() {
  store_REL(&T,NULL);
}

void push(int v) {
  Node *x, *t;
  x = new Node();
  x->data = v;
  do {
    t = load_RLX(&T);
    x->next = t;
  } while
    (!CAS_RLX,REL(&T,t,x));
}

int pop() {
  Node *t, *x;
  do {
    t = load_ACQ(&T);
    if (t == NULL)
      return EMPTY;
    x = t->next;
  } while
    (!CAS_RLX,RLX(&T,t,x));
  return t->data;
}
```

(b)

Figure 1. The Treiber stack. For simplicity, we let pop leak memory. The CASes in the specification always succeed.

Stack specification. As noted in §1, specifications are just alternative library implementations that have the advantage of simplicity, in exchange for inefficiency or nondeterminism. The specification in Figure 1a represents the stack as a sequence abstract data type and provides the three expected methods: `init`, `push` and `pop`. A correct stack implementation should provide the illusion of atomicity of operations to concurrent threads. We specify this by wrapping the bodies of `push` and `pop` in ***atomic sections***, denoted by `atom_sec`. Atomic sections are not part of the standard C11 model—for specification purposes, we have extended the language with a prototype semantics for atomic section (§5). Both `push` and `pop` may non-deterministically diverge, as common stack implementations allow some operations to starve (in concurrency parlance, they are lock-free, but not wait-free). All these are the expected features of a specification on an SC memory model. We now explain the features specific to C11.

The sequence `S` holding the abstract state is declared `atomic`. In C11, programs must not have data races on normal variables; any location where races can occur must be explicitly identified as `atomic` and accessed using the special commands `load`, `store`, and `CAS` (***compare-and-swap***). The latter combines a load and a store into a single operation executed atomically. A CAS takes three arguments: a memory address, an expected value and a new value. The command atomically reads the memory address and, if it contains the expected value, updates it with the new one. Due to our use of atomic sections, the CASes in the specification always succeed. We use CASes here instead of just stores, because, for subtle technical reasons, the latter have a stronger semantics in C11 than our atomic sections (see release sequences in §A).

The `load` and `store` commands are annotated with a ***memory order*** that determines the trade-off between consistency and performance for the memory access; CASes are annotated with two memory orders, as they perform both a load and a store. The choice of memory orders inside a library method can indirectly affect its clients, and thus, a library specification must include them. In the stack specification, several memory operations have the ***release-acquire*** memory orders, denoted by the subscripts REL (for stores) and ACQ (for loads). To explain its effect, consider the following client using the stack according to a typical message-passing idiom:

```
                int a, b, x=0;
                     ||  do {
x=1;                 ||    a = pop();
push(&x);            ||  } while (a==EMPTY);
                     ||  b=*a;
```

The first thread writes `1` into `x` and calls `push(&x)`; the second thread pops the address of `x` from the stack and then reads its contents. In general, a relaxed memory model may allow the second thread to read `0` instead of `1`, e.g., because the compiler reorders `x=1` and `push(&x)`. The release-acquire annotations guarantee that this is not the case: when the ACQ load of `S` in `pop` reads the value written by the REL store to `S` in `push`, the two commands ***synchronise***. We define this notion more precisely later, but informally, it means that the ordering between the REL store and ACQ load constrains the values fetched by reads from other locations, such as the read `*a` in the client.

To enable this message-passing idiom, the specification only needs to synchronise from pushes to pops; it need not synchronise from pops to pushes, or from pops to pops. To avoid unnecessary synchronisation, the specification uses the ***relaxed*** memory order (RLX). This order is weaker than release-acquire, meaning that the set of values a relaxed load can read from memory is less constrained; additionally, relaxed loads and stores do not synchronise with each other. However, relaxed operations are very cheap, since they compile to basic loads and stores without any additional hardware barrier instructions. Hence, the specification allows implementations that are efficient, yet support the intended use of the stack for message passing. On the other hand, as we show below, it intentionally allows non-SC stack behaviours.

Stack implementation. Figure 1b gives our implementation of the Treiber stack. The stack is represented by a heap-allocated linked list of nodes, accessed through a top-of-stack pointer `T`. Only the latter needs to be `atomic`, as it is the only point of contention among threads. The `push` function repeatedly reads from the top pointer `T`, initialises a newly created node `x` to point to the value read, and tries to swing `T` to point to `x` using a CAS; `pop` is implemented similarly. For simplicity, we let `pop` leak memory.

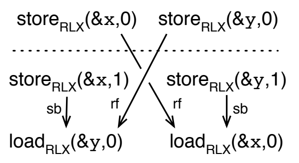
Like the specification, the implementation avoids unnecessary hardware synchronisation by using the relaxed memory order RLX. However, the load of `T` in `pop` is annotated ACQ, since the command `x = t->next` accesses memory based on the value read, and hence, requires it to be up to date.

What does it mean for the implementation in Figure 1b to meet the specification in Figure 1a? As well as returning the right values, it must also faithfully implement the correct synchronisation. To understand how this can be formalised, we must therefore explain how synchronisation works in C11's semantics.

C11 model structure. The C11 memory model is defined axiomatically. An ***execution*** of a program consists of a set of ***actions*** and several partial orders on it. An action describes a memory operation, including the information about the thread that performed it, the address accessed and the values written and/or read. The semantics of a program is given by the set of executions consistent with the program code and satisfying the ***axioms*** of the memory model (see Figure 4 for a flavour of these). Here is a program with one of its executions, whose outcome we explain below:

STORE BUFFERING (SB):

```
store_RLX(&x,0); store_RLX(&y,0)
store_RLX(&x,1)  || store_RLX(&y,1)
load_RLX(&y)     || load_RLX(&x)
```



Note that, in diagrams representing executions, we omit thread identifiers from actions. Several of the most important relations in an execution are:

- ***sequenced-before*** (sb), a transitive and irreflexive relation ordering actions by the same thread according to their program order.
- ***initialised-before*** (ib), ordering initial writes to memory locations before all other actions in the execution[1]. Above we have shown ib by a dotted line dividing the two kinds of actions.
- ***reads-from*** (rf), relating reads r to the writes w from which they take their values: $w \xrightarrow{\text{rf}} r$.
- ***happens-before*** (hb), showing the precedence of actions in the execution. In the fragment of C11 that we consider, it is transitive and irreflexive.

Happens-before is the key relation, and is the closest the C11 model has to the notion of a global time in an SC model: a read must not read any write to the same location related to it in hb other than its immediate predecessor. Thus, for writes w_1 and w_2 and a read r accessing the same location, the following shapes are forbidden:

$$\begin{tikzcd} w_1 \arrow[r, "\text{hb}"] \arrow[rr, bend right, "\text{rf}"'] & w_2 \arrow[r, "\text{hb}"] & r \end{tikzcd} \qquad \begin{tikzcd} r \arrow[r, bend left, "\text{hb}"] & w_1 \arrow[l, bend left, "\text{rf}"] \end{tikzcd} \qquad \text{(RD)}$$

However, in contrast to an SC model, hb is partial in C11, and some reads can read from hb-unrelated writes: we might have $w \xrightarrow{\text{rf}} r$, but not $w \xrightarrow{\text{hb}} r$.

Memory orders. By default, memory reads and writes in C11 are ***non-atomic*** (NA). The memory model guarantees that data-race free programs with only non-atomic memory accesses have SC behaviour. A data race occurs when two actions on the same memory location, at least one of which is a write, and at least one of which is a non-atomic access, are unrelated in happens-before, and thus, intuitively, can take place ‘at the same time’. Hence, non-expert programmers who write code that is free from both data races and atomic accesses need not understand the details of the relaxed memory model. Data races are considered faults, resulting in undefined behaviour for the whole program.

The three main atomic memory orders, from least to most restrictive, are relaxed, release-acquire and sequentially consistent[2]. We have already seen the first two in the stack example above. The third, ***sequentially consistent*** (SC), does not allow relaxed behaviour: if all actions in a race-free program are either non-atomic or SC, the program exhibits only sequentially-consistent behaviour [2, 4]. However, the SC memory order is more expensive.

The weakest memory order, ***relaxed***, exhibits a number of relaxations, as the C11 model places very few restrictions on which write a relaxed read might read from. For example, consider the (SB)

[1] This is a specialisation of the *additional-synchronises-with* relation from the C11 model [2] to programs without dynamic thread creation, to which we restrict ourselves in this paper (see §4).

[2] Release-consume atomics and fences [2] are left for future work (see §9). We also omit some C11 subtleties that are orthogonal to abstraction (see §4).

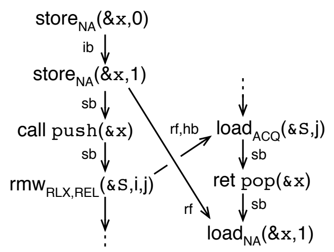
example above. The outcome shown there is allowed by C11, but cannot be produced by any interleaving of the threads' actions. C11 disallows it if all memory accesses are annotated as SC.

The ***release-acquire*** memory orders allow more relaxed behaviour than SC, while still providing some guarantees. Consider the following execution of the client of the stack in Figure 1a or 1b we have seen above:

MESSAGE PASSING (MP):

```
  int a, b, x=0;

x=1;      || do {a=pop();}
push(&x); || while
          ||    (a==EMPTY);
          || b=*a;
```



$\text{store}_{NA}(\&x,0)$
ib
$\text{store}_{NA}(\&x,1)$
sb
call push(&x)
sb
$\text{rmw}_{RLX,REL}(\&S,i,j)$
rf,hb
$\text{load}_{ACQ}(\&S,j)$
sb
ret pop(&x)
sb
rf
$\text{load}_{NA}(\&x,1)$

Here rmw (read-modify-write) is a combined load-store action produced by a CAS. In this case, the ACQ load in pop synchronises with the REL store part of the CAS in push that it reads from. This informal notion of synchronisation we mentioned above is formalised in the memory model by including the corresponding rf edge into hb. Then, since $\mathsf{sb} \cup \mathsf{ib} \subseteq \mathsf{hb}$ and hb is transitive, both writes to x happen before the read. Hence, by (RD), the read from x by the second thread is forced to read from the most recent write, i.e., 1.

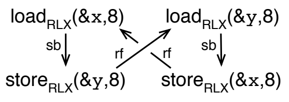
If all the memory order annotations in the Treiber stack were relaxed, the second thread could read 0 from x instead. Furthermore, without release-acquire synchronisation, there would be a data race between the non-atomic write of x in the first thread and the non-atomic read of x in the second.

The release-acquire memory orders only synchronise between pairs of reads and writes, but do not impose a total order over memory accesses, and therefore allow non-SC behaviour. For example, if we annotate writes in (SB) with REL, and reads with ACQ, then the outcome shown there will still be allowed: each load can read from the initialisation without generating a cycle in hb or violating (RD). We can also get this outcome if we use push and pop operations on two instances of the stack from Figure 1a or 1b instead of load and store. Thus, both the implementation and the specification of the stack allow it to have non-SC behaviour.

To summarise, very roughly, release-acquire allows writes to be delayed but not reordered, while relaxed allows both. Relaxed actions produce even stranger behaviour, including what we call ***satisfaction cycles***:

SATISFACTION CYCLE (SCL):

```
a=load_RLX(&x)       || b=load_RLX(&y)
if (a==8)            || if (b==8)
  store_RLX(&y,a)    ||   store_RLX(&x,8)
```



$\text{load}_{RLX}(\&x,8)$
$\text{load}_{RLX}(\&y,8)$
sb
rf
rf
sb
$\text{store}_{RLX}(\&y,8)$
$\text{store}_{RLX}(\&x,8)$

Here, each conditional satisfies its guard from a later write in the other thread. This is possible because relaxed reads and writes do not create any happens-before ordering, and thus neither read is constrained by (RD). Unlike relaxed, release-acquire does not allow satisfaction cycles. If the loads and stores in the example were annotated release-acquire, then both rf edges would also be hb edges. This would produce an hb cycle, which is prohibited by the memory model. Satisfaction cycles are known to be a problematic aspect of the C11 model; as we show in this paper, they also create difficulties for library abstraction.

3. Library Abstraction Informally

Histories. Our approach to abstraction is based on the notion of a ***history***, which concisely records all interactions between the client and the library in a given execution. Clients and libraries can affect one another in several ways in C11. Most straightforwardly, the library can observe the parameters passed by the client at calls, and the client can observe the library's return values. Therefore, a history includes the set of all call and return actions in the execution. However, clients can also observe synchronisation and other memory-model effects inside a method. These more subtle interactions are recorded by two kinds of partial order: *guarantees* and *denies*.

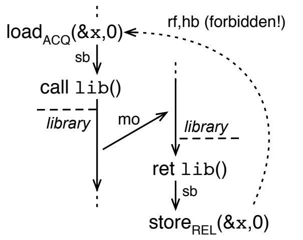
Synchronisation internal to the library can affect the client by forcing reads to read from particular writes. For example, in (MP) from §2, the client is forced to read 1 from x because the push and pop methods synchronise internally in a way that generates an hb ordering between the call to push and the return from pop. If the methods did not hb-synchronise, the client could read from either of the writes to x. The client can thus observe the difference between two library implementations with different internal synchronisation, even if all call and return values are identical. To account for this, the ***guarantee*** relation in a history of an execution records hb edges between library call and return actions.

Even non-synchronising behaviour inside the library can sometimes be observed by the client. For example, the C11 model requires the existence of a total order mo over all atomic writes to a given location. This order cannot go against hb, but is not included into it, as this would make the model much stronger, and would hinder efficient compilation onto very weak architectures, such as Power and ARM [21]. Now, consider the following:

DENY (DN):

```
           store_REL(&x,1);

load_ACQ(&x); || lib();
lib();        || store_REL(&x,0);
```



In this execution, a write internal to the invocation of lib in the second thread is mo-ordered after a write internal to the invocation of lib in the first thread. This forbids the client from reading 0 from x. To see this, suppose the contrary holds. Then the ACQ load synchronises with the REL store of 0, yielding an hb edge. By transitivity with the client sb edges, which are included in hb, we get an hb edge from ret lib in the second thread to call lib in the first. Together with the library's sb edges, this yields an hb edge going against the library-internal mo one, which is prohibited by the memory model. To account for such interactions, the ***deny*** relations in a history of an execution record hb or other kinds of edges between return and call actions that the client *cannot* enforce due to the structure of the library, e.g., the hb-edge from ret lib to call lib above.

Abstraction in the presence of relaxed atomics. As we noted in §1, we actually propose two library abstraction criteria: one for the full memory model described in §2, and one for programs without relaxed atomics. We discuss the former first.

Two library executions with the same history are observationally equivalent to clients, even if the executions are produced by different library implementations. By defining a sound abstraction relation over histories, we can therefore establish abstraction between libraries. To this end, we need to compare the histories of libraries under every client context. Fortunately, we need not examine every possible client: it suffices to consider behaviour under a ***most general client***, whose threads repeatedly invoke library methods in any order and with any parameters. Executions under this client generate all possible histories of the library, and thus represent all client-library interactions (with an important caveat, discussed below). We write $[\![\mathcal{L}]\!]I$ for the set of executions of the library $\mathcal{L}$ under the most general client starting from an ***initial state*** I. Initial states are defined formally in §4, but, informally, record initialisation actions such as the ones shown in (SB). The set $[\![\mathcal{L}]\!]I$ gives the denotation of the library considered in isolation from its clients, and in this sense, defines a ***library-local semantics*** of $\mathcal{L}$.

This library-local semantics allows us to define library abstraction. We now quote its definition and formulate the corresponding Abstraction Theorem, introducing some of the concepts used in them only informally. This lets us highlight their most important features that can be discussed independently of the formalities. We fill in the missing details in §6.1, after we have presented the C11 model more fully.

For the memory model with relaxed atomics, a history contains one guarantee and one deny relation.

DEFINITION 1. *A* ***history*** *is a triple $H = (A, G, D)$, where A is a set of call and return actions, and $G, D \subseteq A \times A$.*

Library abstraction is defined on pairs of libraries $\mathcal{L}$ and sets of their initial states $\mathcal{I}$. It relies on a function $\mathsf{history}(\cdot)$ extracting the history from a given execution of a library, which we lift to sets of executions pointwise. The notation $[\![\mathcal{L}, R]\!]$ and the notion of safety are explained below.

DEFINITION 2. *For histories (A_1, G_1, D_1) and (A_2, G_2, D_2), we let $(A_1, G_1, D_1) \sqsubseteq (A_2, G_2, D_2)$ if $A_1 = A_2$, $G_1 = G_2$ and $D_2 \subseteq D_1$.*

For safe $(\mathcal{L}_1, \mathcal{I}_1)$ and $(\mathcal{L}_2, \mathcal{I}_2)$, $(\mathcal{L}_1, \mathcal{I}_1)$ ***is abstracted by*** *$(\mathcal{L}_2, \mathcal{I}_2)$, written $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$, if for any relation R containing only edges from return actions to call actions, we have*

$$\forall I_1 \in \mathcal{I}_1, H_1 \in \mathsf{history}([\![\mathcal{L}_1, R]\!]I_1).$$
$$\exists I_2 \in \mathcal{I}_2, H_2 \in \mathsf{history}([\![\mathcal{L}_2, R]\!]I_2).\, H_1 \sqsubseteq H_2.$$

The overall shape of the definition is similar to that of linearizability on SC memory models [11]: any behaviour of the concrete library $\mathcal{L}_1$ relevant to the client has to be reproducible by the abstract library $\mathcal{L}_2$. However, there are several things to note.

First, we allow the execution of the abstract library to deny less than the concrete one, but require it to provide the same guarantee. Intuitively, we can strengthen the deny, because this only allows more client behaviours.

Second, we do not consider raw executions of the most general client of $\mathcal{L}_1$ and $\mathcal{L}_2$, but those whose happens-before relation can be ***extended*** with an arbitrary set R of edges between return and call actions without contradicting the axioms of the memory model; $[\![\mathcal{L}_1, R]\!]I_1$ and $[\![\mathcal{L}_2, R]\!]I_2$ denote the sets of all such extensions. The set R represents the happens-before edges that can be enforced by the client: such happens-before edges are not generated by the most general client and, in the presence of relaxed atomics, have to be considered explicitly (this is the caveat to its generality referred to above). We consider only return-to-call edges, as these are the ones that represent synchronisation inside the client (similarly to how call-to-return edges in the guarantee represent synchronisation inside the library; cf. (MP)). The definition requires that, if an extension of the concrete library is consistent with R, then so must be the matching execution of the abstract one.

Finally, the abstraction relation is defined only between ***safe*** libraries that do not access locations internal to the client and do not have faults, such as data races.

As we show in §7, the specification of the Treiber stack in Figure 1a abstracts its implementation in Figure 1b.

Abstraction theorem. We now formulate a theorem that states the correctness of our library abstraction criterion. We consider programs $\mathcal{C}(\mathcal{L})$ with a single client $\mathcal{C}$ and a library $\mathcal{L}$ (the case of multiple libraries is considered in §6.1). The Abstraction Theorem states that, if we replace the (implementation) library $\mathcal{L}_1$ in a program $\mathcal{C}(\mathcal{L}_1)$ with another (specification) library $\mathcal{L}_2$ abstracting $\mathcal{L}_1$, then the set of client behaviours can only increase. Hence, when reasoning about $\mathcal{C}(\mathcal{L}_1)$, we can soundly replace $\mathcal{L}_1$ with $\mathcal{L}_2$ to simplify reasoning.

In the theorem, $[\![\mathcal{C}(\mathcal{L})]\!]\mathcal{I}$ gives the set of executions of $\mathcal{C}(\mathcal{L})$ from initial states in a set $\mathcal{I}$, $\uplus$ combines the initial states of a client and a library, and $\mathsf{client}(\cdot)$ selects the parts of executions generated by client commands. We call $(\mathcal{C}(\mathcal{L}), \mathcal{I})$ ***non-interfering***, if $\mathcal{C}$ and $\mathcal{L}$ do not access each others' internal memory locations in executions of $\mathcal{C}(\mathcal{L})$ from initial states in $\mathcal{I}$. The notion of safety for $\mathcal{C}(\mathcal{L})$ is analogous to the one for libraries.

THEOREM 3 (Abstraction). *Assume that $(\mathcal{L}_1, \mathcal{I}_1)$, $(\mathcal{L}_2, \mathcal{I}_2)$, $(\mathcal{C}(\mathcal{L}_2), \mathcal{I} \uplus \mathcal{I}_2)$ are safe, $(\mathcal{C}(\mathcal{L}_1), \mathcal{I} \uplus \mathcal{I}_1)$ is non-interfering and $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$. Then $(\mathcal{C}(\mathcal{L}_1), \mathcal{I} \uplus \mathcal{I}_1)$ is safe and*

$$\mathsf{client}([\![\mathcal{C}(\mathcal{L}_1)]\!](\mathcal{I} \uplus \mathcal{I}_1)) \subseteq \mathsf{client}([\![\mathcal{C}(\mathcal{L}_2)]\!](\mathcal{I} \uplus \mathcal{I}_2)).$$

The requirement of non-interference is crucial, because it ensures that clients can only observe library behaviour through return values and memory-model effects, rather than by 'opening the box' and observing internal states. The drawback of Theorem 3 is that it requires us to establish the non-interference between the client $\mathcal{C}$ and the *concrete* library $\mathcal{L}_1$, e.g., via a type system or a program logic proof. As we show below, we cannot weaken this condition to allow checking non-interference on the client $\mathcal{C}$ using the *abstract* library $\mathcal{L}_2$, as is standard in data refinement on SC memory models [8]. This makes the reasoning principle given by the theorem less compositional, since establishing non-interference requires considering the composed behaviour of the client and the concrete library—precisely what library abstraction is intended to avoid! However, this does not kill compositional reasoning completely, as non-interference is often simple to check even globally. We can also soundly check other aspects of safety, such as data-race freedom, on $\mathcal{C}(\mathcal{L}_2)$. Furthermore, as we show in §6.1, the notion of library abstraction given by Definition 2 is compositional for non-interfering libraries. As we now explain, we can get the desired theorem allowing us to check non-interference on $\mathcal{C}(\mathcal{L}_2)$ for the fragment of the language excluding relaxed atomics.

Abstraction without relaxed atomics. Restricting ourselves to programs without the relaxed memory order (and augmenting the axiomatic memory model to allow incomplete program runs, as described in §4) allows strengthening our result in three ways:

1. We no longer need to quantify over client happens-before edges R, like in Definition 2. Instead, we enrich histories with an additional deny relation, which is easier to deal with in practice than the quantification. Hence, without relaxed atomics, the caveat to the generality of the most general client does not apply.
2. Abstraction on histories can be defined by inclusion on guarantees, rather than by equality.
3. We no longer need to show that the unabstracted program $\mathcal{C}(\mathcal{L}_1)$ is non-interfering. Rather, non-interference is a consequence of the safety of the abstracted program $\mathcal{C}(\mathcal{L}_2)$.

The first two differences make proofs of library abstraction slightly easier, but are largely incidental otherwise. In particular, quantification over client happens-before edges in Definition 2, although unpleasant, does not make library abstraction proofs drastically more complicated. Requiring the guarantees of the concrete and abstract executions to be equal in this definition just results in more verbose specifications in certain cases. In contrast, the last difference is substantial.

The price of satisfaction cycles. For each of the three above differences we have a counterexample showing that Theorem 3 will not hold if we change the corresponding condition to the one required in the case without relaxed atomics. All of these counterexamples involve satisfaction cycles, which can only be produced by relaxed atomics. Our results show that this language feature makes the reasoning principles for C11 programs less compositional. Due to space constraints, here we present only the counterexample for point 3 above; the others are given in [1, §D]. In §6.3, we identify the corresponding place in the proof of the Abstraction Theorem for the language without relaxed atomics where we rely on the absence of satisfaction cycles.

Consider the following pair of libraries $\mathcal{L}_1$ and $\mathcal{L}_2$:

```
L1:  atomic int x;                 L2:  atomic int x;
     int m() {                          int m() {
       store_RLX(&x,42);                  return 42;
       return load_RLX(&x);             }
     }
```

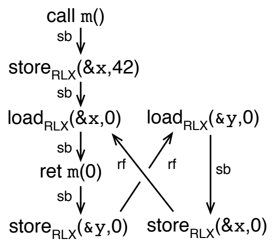
Here x is a library-internal location. We have $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$, since both method implementations behave exactly the same, assuming that the internal location x is not modified outside the library. Unsafe clients can distinguish between $\mathcal{L}_1$ and $\mathcal{L}_2$. For example, the client

```
print m(); || store_RLX(&x,0);
```

can print 0 when using $\mathcal{L}_1$, but not $\mathcal{L}_2$. However, any non-trivial library behaves erratically when clients corrupt its private data structures, and thus, it is reasonable for abstraction to take into account only well-behaved clients that do not do this. We therefore contend that $\mathcal{L}_1$ *should* be abstracted by $\mathcal{L}_2$ according to any sensible abstraction criterion.

The above misbehaved client violates non-interference when using either $\mathcal{L}_1$ or $\mathcal{L}_2$. However, we can define a more complicated client $\mathcal{C}$ such that $\mathcal{C}(\mathcal{L}_2)$ is non-interfering, but $\mathcal{C}(\mathcal{L}_1)$ is not:

```
a=m()                 || b=load_RLX(&y)
if (a==0)             || if (b==0)
  store_RLX(&y,0)     ||   store_RLX(&x,0)
```



The execution of $\mathcal{C}(\mathcal{L}_1)$ given on the right violates non-interference due to a satisfaction cycle: a fault in the client causes the library to misbehave by returning 0 instead of 42, and the effect of this misbehaviour is what causes the client fault in the first place! Since the abstract library $\mathcal{L}_2$ is completely resilient to client interference, its method will always return 42, and thus, the satisfaction cycle will not appear and the client will not access the variable x. Note that this counterexample is not specific to our notion of library abstraction: *any* such notion for C11 considering $\mathcal{L}_2$ to be a specification for $\mathcal{L}_1$ cannot allow checking non-interference using $\mathcal{L}_2$.

For expository reasons, we have given a very simple counterexample. This program would be easy to detect and eliminate, e.g., using a simple type system: one *syntactic* path in the client is guaranteed to result in the forbidden access to the library's internal state. However, the same kind of behaviour can occur with dynamically-computed addresses: the client stepping out of bounds of an array overwrites the library state, causing it to misbehave, which in turn causes the original client misbehaviour. For this kind of example, proving non-interference becomes non-trivial.

It is unclear to us whether satisfaction cycles are observable in practice. They are disallowed by even the weakest C11 target architectures, such as Power and ARM [21], because these architectures respect read-to-write dependencies created by control-flow. It is also clear that the C11 language designers would like to forbid satisfaction cycles: the C++11 standard [12, Section 29.3, Paragraph 11] states that, although (SCL) from §2 is permitted, "implementations should not allow such behaviour". This apparent contradiction is because certain compiler optimisations, such as common subexpression elimination and hoisting from loops, can potentially create satisfaction cycles (see [16] for discussion). Since avoiding them would require compilers to perform additional analysis and/or limit optimisations, the standard does not disallow satisfaction cycles outright. Our results provide an extra argument that allowing satisfaction cycles is undesirable.

4. C11: Language and Thread-Local Semantics

To define the C11 model, we use a lightly modified version of its formalisation proposed by Batty et al. [2]. In the interests of simplicity, we consider a simple core language, instead of the full C/C++, and omit some of the features of the memory model. We do not handle two categories of features: those that are orthogonal to abstraction, and more esoteric features that would complicate our results. In the first category we have dynamic memory allocation, dynamic thread creation, blocked CASes and non-atomic initialisation of atomic locations (we also do not present the treatment of locks here, although we handle a bounded number of them in our formal development; see §A). In the second category we have memory fences and release-consume atomics, discussed in §9.

The semantics of a C11 program is given by a set of executions, generated in two stages. The first stage, described in this section, generates a set of *action structures* using a sequential thread-local semantics which takes into account only the structure of every thread's statements, not the semantics of memory operations. In particular, the values of reads are chosen arbitrarily, without regard for writes that have taken place. The second stage, described in §5, filters out the action structures that are inconsistent with the C11 memory model. It does this by constructing additional relations and checking the resulting executions against the axioms of the model.

Programming language. We assume that the memory consists of locations from Loc containing values in Val. We assume a function $\mathsf{sort} : \mathsf{Loc} \to \{\mathsf{ATOM}, \mathsf{NA}\}$, showing whether memory accesses to a given location must be atomic (ATOM) or non-atomic (NA); see §2. The program syntax is as follows:

$$\begin{aligned} C &::= \mathsf{skip} \mid c \mid m \mid C;C \mid \mathsf{if}(x)\ \{C\}\ \mathsf{else}\ \{C\} \mid \\ &\quad\ \ \mathsf{while}(x)\ \{C\} \mid \mathsf{atom\_sec}\ \{c\} \\ \mathcal{L} &::= \{m = C_m \mid m \in M\} \\ \mathcal{C}(\mathcal{L}) &::= \mathsf{let}\ \mathcal{L}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n \end{aligned}$$

A program consists of a ***library*** $\mathcal{L}$ implementing methods $m \in \mathsf{Method}$ and its ***client*** $\mathcal{C}$, given by a parallel composition of threads $C_1, \ldots, C_n$. The commands include skip, an arbitrary set of base commands $c \in \mathsf{BComm}$ (e.g., atomic and non-atomic loads and stores, and CASes), method calls $m \in \mathsf{Method}$, sequential composition, branching on the value of a location $x \in \mathsf{Loc}$ and loops. Our language also includes atomic sections, ensuring that a base command executes atomically. Atomic sections are not part of C/C++, but are used here to express library specifications. We assume that every method called by the client is defined in the library, and we disallow nested method calls.

We assume that every method accepts a single parameter and returns a single value. Parameters and return values are passed by every thread via distinguished locations in memory, denoted $\mathsf{param}_t, \mathsf{retval}_t \in \mathsf{Loc}$ for $t = 1..n$, such that $\mathsf{sort}(\mathsf{param}_t) = \mathsf{sort}(\mathsf{retval}_t) = \mathsf{NA}$. The rest of memory locations are partitioned into those owned by the client (CLoc) and the library (LLoc):

$$\mathsf{Loc} = \mathsf{CLoc} \uplus \mathsf{LLoc} \uplus \{\mathsf{param}_t, \mathsf{retval}_t \mid t = 1..n\}.$$

The property of non-interference introduced in §3 then requires that a library or a client access only the memory locations belonging to them (except the ones used for passing parameters and return values). We provide pointers on how we can relax the requirement of static address space partitioning in §9.

Actions. Executions are composed of ***actions***, defined as follows:

$$\begin{array}{rcl} \lambda, \mu \in \mathsf{MemOrd} & ::= & \mathsf{NA} \mid \mathsf{SC} \mid \mathsf{ACQ} \mid \mathsf{REL} \mid \mathsf{RLX} \\ \varphi \in \mathsf{Effect} & ::= & \mathsf{store}_\lambda(x,a) \mid \mathsf{load}_\lambda(x,a) \\ & & \mid \mathsf{rmw}_{\lambda,\mu}(x,a,b) \mid \mathsf{call}\ m(a) \mid \mathsf{ret}\ m(a) \\ u,v,w,q,r \in \mathsf{Act} & ::= & (e,g,t,\varphi) \end{array}$$

Here $e \in \mathsf{AId}$ is a unique ***action identifier***, and λ, μ are memory orders (§2) of memory accesses. Every instance of an atomic section occurring in an execution has a unique identifier $g \in \mathsf{SectId}$. Atomic sections only have force when multiple actions have the same section identifier, so actions outside any section are simply assigned a unique identifier each. The domains of the rest of the variables are as follows: $t \in \{0,\dots,n\}$, $x \in \mathsf{Loc}$, $a,b \in \mathsf{Val}$. We allow actions by a dummy thread 0 to denote memory initialisation. We only consider actions whose memory orders respect location sorts given by sort, and we do not allow rmw actions of sort NA.

Loading or storing a value a at a location x generates the obvious action. A read-modify-write action $(e,g,t,\mathsf{rmw}_{\lambda,\mu}(x,a,b))$ arises from a successful compare-and-swap command. It corresponds to reading the value a from the location x and atomically overwriting it with the value b; λ and μ give the memory orders of the read and the write, respectively, and have to be different from NA. The value a in $(e,g,t,\mathsf{call}\ m(a))$ or $(e,g,t,\mathsf{ret}\ m(a))$ records the parameter param_t or the return value retval_t passed between the library method and its client. We refer to call and return actions as ***interface actions***.

For an action u we write $\mathsf{sec}(u)$ for its atomic section identifier, and we denote the set of all countable sets of actions by $\mathcal{P}(\mathsf{Act})$. We omit e, g and λ annotations from actions when they are irrelevant. We also write _ for an expression whose value is irrelevant. We use $(t, \mathsf{read}_\lambda(x,a))$ to mean any of the following:

$$\begin{array}{c} (t, \mathsf{load}_\lambda(x,a)); \quad (t, \mathsf{rmw}_{\lambda,\_}(x,a,\_)); \\ (t, \mathsf{call}\ \_(a)),\ \text{if } x = \mathsf{param}_t \text{ and } \lambda = \mathsf{NA}; \\ (t, \mathsf{ret}\ \_(a)),\ \text{if } x = \mathsf{retval}_t \text{ and } \lambda = \mathsf{NA}. \end{array}$$

We use $(t, \mathsf{write}_\lambda(x,a))$ to mean $(t, \mathsf{store}_\lambda(x,a))$ or $(t, \mathsf{rmw}_{\_,\lambda}(x,\_,a))$. We call the two classes of actions ***read actions*** and ***write actions***, respectively.

Thread-local semantics. The thread-local semantics generates a set of ***action structures***—triples $(A, \mathsf{sb}, \mathsf{ib})$, where $A \in \mathcal{P}(\mathsf{Act})$, and $\mathsf{sb}, \mathsf{ib} \subseteq A \times A$ are the sequenced-before and initialised-before relations introduced in §2. We assume that sb is transitive and irreflexive and relates actions by the same thread; ib relates initialisation actions with thread identifier 0 to the others. We do not require sb to be a total order: in C/C++, the order of executing certain program constructs is unspecified.

For a base command $c \in \mathsf{BComm}$, we assume a set $\langle c \rangle_t \in \mathcal{P}(\mathcal{P}(A) \times \mathcal{P}(A \times A))$ of all pairs of action sets and sb relations that c produces when executed by a thread t (the ib relations are missing, as they are relevant only for a whole program). Note that base commands may include conditionals and loops, and thus can give rise to an arbitrary number of actions; we give a separate explicit semantics to conditionals and loops only because they are used in the most general client in §6.1. Definitions of $\langle c \rangle_t$ for sample base commands c are given in Figure 2. Note that, in the thread-local semantics, a read from memory, such as $\mathsf{load}_\mu(y)$ in the figure, yields an arbitrary value. A CAS command generates an rmw action, if successful, and a load, otherwise.

We define an ***initial state*** of a program $\mathcal{C}(\mathcal{L})$ by a function

$$I \in (\mathsf{LLoc} \uplus \mathsf{CLoc}) \rightharpoonup_{fin} (\mathsf{Val} \times \mathsf{MemOrd}),$$

giving the initial values of memory locations, together with the memory orders of initial writes to them. We define the set of action structures $\langle \mathcal{C}(\mathcal{L}) \rangle I$ of a program $\mathcal{C}(\mathcal{L})$ in Figure 3. Note that this set of action structures corresponds to complete runs of $\mathcal{C}(\mathcal{L})$. The clause for $\mathsf{atom\_sec}\ \{c\}$ assigns the same atomic section identifier to all actions generated by c. The clause for a call to a method m brackets structures generated by its implementation C_m with call and return actions. We have omitted the clause for loops [1, §B]. For simplicity, we assume that the variable in the condition of a branch is always non-atomic.

Assumptions. We make several straightforward assumptions about the structures in $\langle c \rangle_t$ for $c \in \mathsf{BComm}$:

- Structures in $\langle c \rangle_t$ are finite and contain only load, store and read-modify-write actions by t with unique action identifiers.
- For any $(A, \mathsf{sb}) \in \langle c \rangle_t$, sb is transitive and irreflexive.
- Structures in $\langle c \rangle_t$ are insensitive to the choice of action and atomic section identifiers: applying any bijection to such identifiers from a structure in $\langle c \rangle_t$ produces another structure in $\langle c \rangle_t$.
- Atomic sections in every $(A, \mathsf{sb}) \in \langle c \rangle_t$ are contiguous in sb:

$$\forall u,v,q.\, \mathsf{sec}(u) = \mathsf{sec}(v) \wedge u \xrightarrow{\mathsf{sb}} q \xrightarrow{\mathsf{sb}} v \implies \mathsf{sec}(q) = \mathsf{sec}(u).$$

So as not to obfuscate presentation, we only consider programs $\mathcal{C}(\mathcal{L})$ that use param_t and retval_t correctly. We assume that in any action structure of a program, only library actions by t read param_t and write to retval_t, and only client actions by t read retval_t and write to param_t. We also require that param_t and retval_t be initialised before an access: in any structure $(A, \mathsf{sb}, \mathsf{ib})$ of $\mathcal{C}(\mathcal{L})$,

$$\begin{array}{c} (\forall u = (t, \mathsf{call}\ \_) \in A.\, \exists w = (t, \mathsf{write}(\mathsf{param}_t, \_)).\, w \xrightarrow{\mathsf{sb}} u) \wedge \\ (\forall u = (t, \mathsf{ret}\ \_) \in A.\, \exists w = (t, \mathsf{write}(\mathsf{retval}_t, \_)).\, w \xrightarrow{\mathsf{sb}} u). \end{array}$$

Additional assumptions without relaxed atomics. For our result without relaxed atomics (§6.2), we currently require the following additional assumptions:

- The structure set $\langle c \rangle_t$ accounts for c fetching any values from the memory locations it reads (see [1, §B] for formalisation).
- Any structure of a base command c inside an atomic section accesses at most one atomic location. This is sufficient for our purposes, since a library specification usually accesses a single such location containing the abstract state of the library.
- We modify the standard C11 model by requiring that a program's semantics include structures corresponding to execution prefixes. In the standard C11 model, all executions are complete (although possibly infinite). We define $\langle C_i \rangle_t^p$ for a thread C_i by

$$\begin{array}{l} \langle C_i \rangle_t^p = \{(A_p, \mathsf{sb}_p) \mid \exists (A, \mathsf{sb}) \in \langle C_i \rangle_t. \\ \qquad A_p \subseteq A \wedge \forall u \in A, v \in A_p.\, u \xrightarrow{\mathsf{sb}} v \implies u \in A_p\}. \end{array}$$

This is necessary due to the interaction between our prototype atomic section semantics and the C11 model. It weakens the notion of atomicity: atomic sections at the end of a prefix may be partially executed, and therefore more weakly ordered than their completed counterparts. Eliminating it will require a deeper understanding of the relationship between atomicity and the notion of incomplete program runs. See §6.3 for its use in the proof.

5. C11: Axiomatic Memory Model

The axiomatic portion of the C11 model takes a set of action structures of the program, generated by the thread-local semantics in §4, and filters out those which are inconsistent with the memory model. To formulate it, we enrich action structures with extra relations.

Executions. The semantics of a program consists of a set of ***executions***, $X = (A, \mathsf{sb}, \mathsf{ib}, \mathsf{rf}, \mathsf{sc}, \mathsf{mo}, \mathsf{sw}, \mathsf{hb})$, where $A \in \mathcal{P}(\mathsf{Act})$ and the rest are relations on A:

- sb, ib, rf and hb introduced in §4; rf is such that its reverse is a partial function from read to write actions on the same location and with the same value.
- ***sequentially consistent order*** (sc), ordering SC reads from and writes to the same location. The projection of sc to each atomic location is a transitive, irreflexive total order, while writes to distinct locations are unrelated[3].
- ***modification order*** (mo), ordering writes (but not reads!) to the same atomic (i.e., of the sort ATOM) location. The projection of mo to each atomic location is a transitive, irreflexive total order.
- ***synchronises-with*** (sw), defining synchronisation.

We write $\mathsf{r}(X)$ for the component r of the execution X.

We can now define the denotation $[\![\mathcal{C}(\mathcal{L})]\!]$ of a program and the notion of safety and non-interference introduced informally in §3. An execution X is ***valid***, if it satisfies the validity axioms shown in Figure 4; it is ***safe***, if it satisfies the safety axioms in Figure 5, and it is ***non-interfering*** if it satisfies the NONINTERF axiom from Figure 5. We explain the axioms shown in the rest of this section. Intuitively, validity axioms correspond to properties that are enforced by the runtime, while safety axioms correspond to properties that the programmer must ensure to avoid faults. To simplify the following explanations, Figures 4 and 5 do not show axioms dealing with CASes and locks. To keep the presentation tractable, we have also omitted some corner cases from SWDEF and SCREADS in Figure 4. The missing axioms and cases are given in §A, and our results are established for the memory model including these (the correctness of the stack in Figure 1 actually relies on a corner case in SWDEF).

For a program $\mathcal{C}(\mathcal{L})$ and an initial state I, we let $[\![\mathcal{C}(\mathcal{L})]\!]I$ be the set of valid executions X, whether safe or not, such $(A, \mathsf{sb}(X), \mathsf{ib}(X)) \in \langle \mathcal{C}(\mathcal{L}) \rangle I$. We write $[\![\mathcal{C}(\mathcal{L})]\!]\mathcal{I}$ to stand for its obvious lifting to sets $\mathcal{I}$ of initial states. A program $\mathcal{C}(\mathcal{L})$ is ***safe*** when run from I if every one of its valid executions is (and similarly for non-interference and sets of initial states). An unsafe program has undefined behaviour.

The validity axioms define sw and hb directly in terms of the other relations (SWDEF and HBDEF). The hb relation is constructed from the sb, ib and sw, and as follows from ACYCLICITY, has to be irreflexive. The $/\!\sim$ operator in the definition of hb is needed to handle atomic sections; for now the reader should ignore it. The sw relation is derived from sc and rf. The rf, sc and mo relations are only constrained by the axioms, not defined directly. We explain the validity axioms by first considering a language fragment with non-atomic memory accesses only and then gradually expanding it to include the other memory orders.

Non-atomic memory accesses. The values read by non-atomic reads are constrained by DETREAD and RFNONATOMIC. DETREAD requires every read to have an associated rf edge when the location read was previously initialised, i.e., when there is a write to it that happened before the read. Executions with reads from uninitialised locations are valid, but, as we explain below, unsafe. RFNONATOMIC requires that a read only reads from the write to the same location immediately preceding it in hb; cf. (RD). In the absence of other synchronisation, this means that a thread can read only from its own previous writes or initial values, since by HBDEF, $\mathsf{sb} \cup \mathsf{ib} \subseteq \mathsf{hb}$. Threads can establish the necessary synchronisation using atomic operations (which we explain now) or locks (which we elide here; see §A).

[3] In the original C11 model [2], sc is a total order on SC operations over all locations. The formulation here is is equivalent to the original one [1, §C], but more convenient for defining library abstraction.

SC atomics. The strong semantics of SC actions is enforced by organising all SC reads and writes over a *single* location into a total order sc, which cannot form a cycle with hb (ACYCLICITY). According to SCREADS, an SC read can only read from the closest sc-preceding write. Thus, if all memory accesses are annotated as SC in (SB) from §2, the result shown there is forbidden. Indeed, by SCREADS and ACYCLICITY, the store of 1 to y has to follow the load of 0 from y in sc, and similarly for x. This yields a cycle in $\mathsf{sb} \cup \mathsf{sc}$, contradicting ACYCLICITY. Note that the model requires the existence of sc, but does not include all of it into hb. As a consequence, one cannot use the ordering of, say, two SC reads in sc to constrain the values that can be read according to RFNONATOMIC.

By SWDEF, an rf edge between an SC write and an SC read generates an sw edge, which is then included into hb by HBDEF. Release-acquire atomics have the same effect, as we now explain.

Release-acquire atomics. By SWDEF, an ACQ read synchronises with the REL write it reads from. For example, if in (SCL) we annotated all writes with REL and all reads with ACQ, then the rf edges would be included into hb and the execution would be prohibited by ACYCLICITY.

For atomics weaker than SC, there is no total order on all operations over a given location analogous to sc; this is why (SB) is allowed. Instead, they satisfy a weaker property of ***coherence***: all writes (but not reads) to a single atomic location are organised into a total modification order mo, which has to be consistent with hb (HBVSMO). SC writes to the location are also included into mo, and in such cases the latter has to be consistent with sc (MOVSSC).

Since reads are not included into mo, we do not have an analogue of SCREADS, and thus, a read has more freedom to choose which write it reads from. The only constraints on atomic accesses weaker than SC are given by coherence axioms—COWR, CORW and CORR. For example, COWR says that a read r that happened after a write w_2 cannot read from a write w_1 earlier in mo.

Relaxed atomics. Like release-acquire atomics, relaxed atomics respect coherence, given by the mo order and the axioms COWR, CORW and CORR. However, rf edges involving them do not generate synchronisation edges sw. The only additional constraint on relaxed reads is given by RFATOMIC, which prohibits reads 'from the future', i.e., from writes later in hb; cf. (RD). This and the fact that coherence axioms enforce no constraints on actions over distinct locations allows (SCL). If all the loads and stores in (SCL) were to the same location, it would be forbidden by CORW.

Safety axioms. The safety axioms in Figure 5 define the conditions under which a program is faulty. DRF constrains pairs of actions over the same location, with at least one write. It requires that such pairs on distinct threads, one of which is a non-atomic access, are related by hb, and on the same thread, by sb (recall that in C/C++, the order of executing certain program constructs is unspecified, and thus, sb is partial). SAFEREAD prohibits reads from uninitialised locations.

The NONINTERF axiom is not part of the C11 memory model, but formalises the property of non-interference required for our results to hold (§3); it is technically convenient for us to consider it together with the other safety axioms. NONINTERF requires that the library and the client only read from and write to the locations they own, except param_t and retval_t used for communication (§4). The axiom classifies an action as performed by the library or the client depending on its position in sb with respect to calls and returns.

Atomic sections. Atomic sections are a widespread idiom for defining library specifications. In an SC memory model, we can

$$\langle \mathsf{store}_\lambda(x, \mathsf{load}_\mu(y))\rangle_t \;=\; \{(\{u,v\},\{(u,v)\}) \mid \exists a', e_1, e_2, g_1, g_2.\, e_1 \neq e_2 \wedge g_1 \neq g_2 \wedge u = (e_1, g_1, t, \mathsf{load}_\mu(y,a')) \wedge v = (e_2, g_2, t, \mathsf{store}_\lambda(x,a'))\}$$

$$\langle *y = \mathsf{CAS}_{\lambda,\mu}(x,a,b)\rangle_t \;=\; \{(\{u,v\},\{(u,v)\}) \mid \exists e_1, e_2, g_1, g_2, a'.\, e_1 \neq e_2 \wedge g_1 \neq g_2 \wedge a' \neq a \wedge (u = (e_1,g_1,t,\mathsf{rmw}_{\lambda,\mu}(x,a,b)) \wedge v = (e_2,g_2,t,\mathsf{store}_{\mathsf{NA}}(y,1))) \vee (u = (e_1,g_1,t,\mathsf{load}_\lambda(x,a')) \wedge v = (e_2,g_2,t,\mathsf{store}_{\mathsf{NA}}(y,0)))\}$$

Figure 2. Definitions of $\langle c\rangle_t$ for sample base commands. Here $x, y \in \mathsf{Loc}$ and $a, b \in \mathsf{Val}$ are constants.

$$\langle \mathsf{skip}\rangle_t \;=\; \{(\emptyset,\emptyset)\}$$

$$\langle C_1; C_2\rangle_t \;=\; \{(A_1 \uplus A_2, \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(u,v) \mid u \in A_1 \wedge v \in A_2\}) \mid (A_1, \mathsf{sb}_1) \in \langle C_1\rangle_t \wedge (A_2, \mathsf{sb}_2) \in \langle C_2\rangle_t\}$$

$$\langle \mathsf{if}(x)\ \{C_1\}\ \mathsf{else}\ \{C_2\}\rangle_t \;=\; \{(\{u\} \uplus A, \mathsf{sb} \cup \{(u,v) \mid v \in A\}) \mid \exists a.\, (A, \mathsf{sb}) \in \langle C_1\rangle_t \wedge u = (\_,\_,t,\mathsf{load}_{\mathsf{NA}}(x,a)) \wedge a \neq 0\} \cup \{(\{u\} \uplus A, \mathsf{sb} \cup \{(u,v) \mid v \in A\}) \mid (A, \mathsf{sb}) \in \langle C_2\rangle_t \wedge u = (\_,\_,t,\mathsf{load}_{\mathsf{NA}}(x,0))\}$$

$$\langle \mathsf{atom\_sec}\ \{C\}\rangle_t \;=\; \{(\{(e,g,t,\varphi) \mid (e,\_,t,\varphi) \in A\}, \{((e_1,g,t,\varphi_1),(e_2,g,t,\varphi_2)) \mid ((e_1,\_,t,\varphi_1),(e_2,\_,t,\varphi_2)) \in \mathsf{sb}\}) \mid (A, \mathsf{sb}) \in \langle C\rangle_t \wedge g \in \mathsf{SectId}\}$$

$$\langle m\rangle_t \;=\; \{(A \uplus \{u\} \uplus \{v\}, \mathsf{sb} \cup \{(u,v)\} \cup \{(u,q),(q,v) \mid q \in A\}) \mid (A, \mathsf{sb}) \in \langle C_m\rangle_t \wedge u = (\_,\_,t,\mathsf{call}\ m(\_)) \wedge v = (\_,\_,t,\mathsf{ret}\ m(\_))\}$$

$$\langle \mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n\rangle I \;=\; \Big\{(A_0 \uplus (\textstyle\biguplus_{t=1}^n A_t), \bigcup_{t=1}^n \mathsf{sb}_t, (A_0 \times (\biguplus_{t=1}^n A_t))) \mid (\forall t = 1..n.\, (A_t, \mathsf{sb}_t) \in \langle C_t\rangle_t) \wedge (\forall t = 1..n.\, \forall u.\, \exists \text{ finitely many } v.\, (v,u) \in \mathsf{sb}_t) \wedge (A_0 = \biguplus\{(e,g,0,\mathsf{store}_\lambda(x,a)) \mid I(x) = (a,\lambda) \wedge e \in \mathsf{AId} \wedge g \in \mathsf{SectId}\})\Big\}$$

Figure 3. Thread-local semantics. $A \uplus B$ is the union of the sets of actions A and B with disjoint sets of action and atomic section identifiers.

HBDEF. $\mathsf{hb} = ((\mathsf{sb} \cup \mathsf{ib} \cup \mathsf{sw})/\!\sim)^+$, where
$R/\!\sim\, = R \cup \{(u,v) \mid \mathsf{sec}(u) \neq \mathsf{sec}(v) \wedge \exists u', v'.\, \mathsf{sec}(u) = \mathsf{sec}(u') \wedge \mathsf{sec}(v) = \mathsf{sec}(v') \wedge u' \xrightarrow{R} v'\}$

SWDEF$^{\approx}$. $\forall w, r.\, w \xrightarrow{\mathsf{sw}} r \iff \begin{pmatrix} \exists t_1, t_2, \lambda, \mu, x.\, t_1 \neq t_2 \wedge \lambda \in \{\mathsf{SC}, \mathsf{REL}\} \wedge \mu \in \{\mathsf{SC}, \mathsf{ACQ}\} \\ \wedge\, w = (t_1, \mathsf{write}_\lambda(x,\_)) \wedge r = (t_2, \mathsf{read}_\mu(x,\_)) \wedge w \xrightarrow{\mathsf{rf}} r \end{pmatrix}$

ACYCLICITY. $\mathsf{hb} \cup \mathsf{sc}$ is acyclic

DETREAD. $\forall r.\, (\exists x, w'.\, w' \xrightarrow{\mathsf{hb}} r \wedge w' = (\_, \mathsf{write}(x,\_)) \wedge r = (\_, \mathsf{read}(x,\_))) \iff (\exists w.\, w \xrightarrow{\mathsf{rf}} r)$

RFATOMIC.

$$\neg\exists r, w. \quad \begin{tikzcd} r \arrow[r, bend left, "\mathsf{hb}"] & w \arrow[l, bend left, "\mathsf{rf}"] \end{tikzcd}$$

RFNONATOMIC. $\forall w, r, x.\, w \xrightarrow{\mathsf{rf}} r \wedge w = (\_, \mathsf{write}(x,\_)) \wedge r = (\_, \mathsf{read}(x,\_)) \wedge \mathsf{sort}(x) = \mathsf{NA}$
$\implies w \xrightarrow{\mathsf{hb}} r \wedge \neg\exists w'.\, w' = (\_, \mathsf{write}(x,\_)) \wedge w \xrightarrow{\mathsf{hb}} w' \xrightarrow{\mathsf{hb}} r$

SCREADS$^{\approx}$. $\forall w, r.\, w \xrightarrow{\mathsf{rf}} r \wedge r = (\_, \mathsf{read}_{\mathsf{SC}}(x,\_)) \wedge w = (\_, \mathsf{write}_{\mathsf{SC}}(x,\_)) \implies w \xrightarrow{\mathsf{sc}} r \wedge \neg\exists w'.\, w' = (\_, \mathsf{write}(x,\_)) \wedge w \xrightarrow{\mathsf{sc}} w' \xrightarrow{\mathsf{sc}} r$

HBVSMO. $\neg\exists w_1, w_2.$

$$\begin{tikzcd} w_1 \arrow[r, bend left, "\mathsf{hb}"] & w_2 \arrow[l, bend left, "\mathsf{mo}"] \end{tikzcd}$$

MOVSSC. $\neg\exists w_1, w_2.$

$$\begin{tikzcd} w_1 \arrow[r, bend left, "\mathsf{mo}"] & w_2 \arrow[l, bend left, "\mathsf{sc}"] \end{tikzcd}$$

COWR. $\neg\exists w_1, w_2.$

$$\begin{tikzcd} w_1 \arrow[r, "\mathsf{mo}"] \arrow[dr, "\mathsf{rf}"'] & w_2 \arrow[d, "\mathsf{hb}"] \\ & r \end{tikzcd}$$

CORW. $\neg\exists r, w_1, w_2.$

$$\begin{tikzcd} w_2 \arrow[r, "\mathsf{rf}"] & r \arrow[d, "\mathsf{hb}"] \\ & w_1 \arrow[ul, "\mathsf{mo}"] \end{tikzcd}$$

CORR. $\neg\exists r_1, r_2, w_1, w_2.$

$$\begin{tikzcd} w_1 \arrow[r, "\mathsf{rf}"] & r_1 \arrow[d, "\mathsf{hb}"] \\ w_2 \arrow[u, "\mathsf{mo}"] \arrow[r, "\mathsf{rf}"'] & r_2 \end{tikzcd}$$

ASMO. $\forall u, v.\, u \xrightarrow{\mathsf{mo}} v \wedge \mathsf{sec}(u) = \mathsf{sec}(v) \implies \neg\exists q.\, u \xrightarrow{\mathsf{mo}} q \xrightarrow{\mathsf{mo}} v \wedge \mathsf{sec}(u) \neq \mathsf{sec}(q)$

ASSC. $\forall u, v.\, u \xrightarrow{\mathsf{sc}} v \wedge \mathsf{sec}(u) = \mathsf{sec}(v) \implies \neg\exists q.\, u \xrightarrow{\mathsf{sc}} q \xrightarrow{\mathsf{sc}} v \wedge \mathsf{sec}(u) \neq \mathsf{sec}(q)$

Figure 4. Selected validity axioms of the C11 memory model. Axioms simplified for the purposes of presentation are marked by $^{\approx}$.

DRF. $\forall u, v, x, t_1, t_2.\, (u, v \in A \wedge u \neq v \wedge u = (t_1, \_(x,\_)) \wedge v = (t_2, \_(x,\_)) \wedge (u = (t_1, \mathsf{write}(x,\_)) \vee v = (t_2, \mathsf{write}(x,\_)))) \implies$
$((t_1 \neq t_2 \implies (u \xrightarrow{\mathsf{hb}} v \vee v \xrightarrow{\mathsf{hb}} u \vee \mathsf{sort}(x) = \mathsf{ATOM})) \wedge (t_1 = t_2 \implies (u \xrightarrow{\mathsf{sb}} v \vee v \xrightarrow{\mathsf{sb}} u)))$

SAFEREAD. $\forall r.\, r \in A \wedge r = (\_, \mathsf{read}(\_,\_)) \implies \exists w.\, w \xrightarrow{\mathsf{rf}} r$

NONINTERF. $\forall u, x, t.\, (u \in A \wedge t \neq 0 \wedge u = (t, \_(x,\_)) \wedge x \notin \{\mathsf{param}_t, \mathsf{retval}_t \mid t = 1..n\}) \implies$
$((\exists v.\, v = (\_, \mathsf{call}\ \_) \wedge v \xrightarrow{\mathsf{sb}} u \wedge \neg\exists q.\, q = (\_, \mathsf{ret}\ \_) \wedge v \xrightarrow{\mathsf{sb}} q \xrightarrow{\mathsf{sb}} u) \iff (x \in \mathsf{LLoc}))$

Figure 5. Selected safety axioms of the C11 memory model

DETREAD$_l$. $\forall r, t.\, ((\exists x.\, r = (\_, \mathsf{read}(x,\_)) \wedge x \neq \mathsf{param}_t \wedge \exists w.\, w \xrightarrow{\mathsf{hb}} r \wedge w = (\_, \mathsf{write}(x,\_))) \iff \exists w'.\, w' \xrightarrow{\mathsf{rf}} r) \wedge$
$(\forall a, b.\, r = (t, \mathsf{read}(\mathsf{param}_t, a)) \in A \wedge u = (\_, \mathsf{call}\ \_(b)) \wedge u \xrightarrow{\mathsf{sb}} r \wedge (\neg\exists v.\, v = (\_, \mathsf{ret}\ \_) \wedge u \xrightarrow{\mathsf{sb}} v \xrightarrow{\mathsf{sb}} r) \implies a = b)$

SAFEREAD$_l$. $\forall r, x, t.\, r \in A \wedge r = (t, \mathsf{read}(x,\_)) \wedge x \neq \mathsf{param}_t \implies \exists w.\, w \xrightarrow{\mathsf{rf}} r$

DETREAD$_c$. $\forall r, t.\, ((\exists x.\, r = (\_, \mathsf{read}(x,\_)) \wedge x \neq \mathsf{retval}_t \wedge \exists w.\, w \xrightarrow{\mathsf{hb}} r \wedge w = (\_, \mathsf{write}(x,\_))) \iff \exists w'.\, w' \xrightarrow{\mathsf{rf}} r) \wedge$
$(\forall a, b.\, r = (t, \mathsf{read}(\mathsf{retval}_t, a)) \wedge u = (\_, \mathsf{ret}\ \_(b)) \wedge u \xrightarrow{\mathsf{sb}} r \wedge (\neg\exists v.\, v = (\_, \mathsf{call}\ \_) \wedge u \xrightarrow{\mathsf{sb}} v \xrightarrow{\mathsf{sb}} r) \implies a = b)$

SAFEREAD$_c$. $\forall r, x, t.\, (r \in A \wedge r = (t, \mathsf{read}(x,\_)) \wedge x \neq \mathsf{retval}_t \implies \exists w.\, w \xrightarrow{\mathsf{rf}} r) \wedge$
$(r \in A \wedge r = (t, \mathsf{read}(\mathsf{retval}_t,\_)) \wedge r \neq (\_, \mathsf{ret}\ \_) \implies \exists u.\, u \xrightarrow{\mathsf{hb}} r \wedge u = (t, \mathsf{ret}\ \_))$

Figure 6. Axioms for library (§6.1) and client (§6.3) executions

define their semantics by simply requiring that no other events are interleaved with actions inside an atomic section. Unfortunately, the relaxed memory model of C11 does not admit such a simple definition. The straightforward solution of imposing a total order on all instances of atomic sections would rule out relaxed specifications that we would like to give, such as the Treiber specification from §2. Hence, we have extended C11 with a prototype notion of atomic sections suitable for its relaxed-memory setting (inspired by the semantics of transactions in [6]). This notion represents only the first step towards a natural specification language for relaxed C11, which is an interesting problem in itself.

The axioms defining the semantics of atomic sections are HBDEF, ASMO and ASSC in Figure 4 and ATOMAS in Figure 7 (deferred to §A for brevity). They capture the expected properties of atomicity. Thus, in HBDEF, we factor $\mathsf{sb} \cup \mathsf{ib} \cup \mathsf{sw}$ over atomic sections using $/\!\sim$: e.g., if an action u happens-before another action v, then u also happens-before any other action from the same atomic section as v. ASMO and ASSC require that actions from the same atomic section be contiguous in mo and sc. ATOMAS constrains relaxed actions, which do not generate hb edges. ASMO, ASSC and ATOMAS are trivially satisfied when every action has a unique atomic section identifier. Additionally, in this case HBDEF simplifies to $\mathsf{hb} = (\mathsf{sb} \cup \mathsf{ib} \cup \mathsf{sw})^+$, which is how it is defined in standard C11 [2]. Thus, if every action executes in a separate atomic section, our augmented model coincides with standard C11.

6. Library Abstraction in Detail

We first define formally the concepts used in the definition of library abstraction (Definition 2) and the Abstraction Theorem (Theorem 3) from §3 for the memory model with relaxed atomics. We then show how the Abstraction Theorem can be strengthened for a fragment of the language excluding them (§6.2) and give the proof outlines for both theorems (§6.3).

6.1 Library Abstraction in the Presence of Relaxed Atomics

History definition. We formally define the history function, which selects a history in the sense of Definition 2 from a library execution. For an execution X, we let

$$\mathsf{history}(X) = (\mathsf{interf}(X), \mathsf{hbL}(X), \mathsf{scL}(X))$$

and lift history to sets of executions pointwise. Here $\mathsf{interf}(X)$ is the projection of $A(X)$ to interface (call and return) actions. The hbL selector computes the guarantee part of the history. We let $\mathsf{hbL}(X)$ be the projection of $\mathsf{hb}(X)$ to pairs of actions of the form $((\_, \mathsf{call}\ \_), (\_, \mathsf{ret}\ \_))$ and pairs of calls and returns (in any order) by the same thread. We record only edges of the above form, since it can be shown that any happens-before edge between interface actions in a library execution under its most general client can be obtained as a transitive closure of such edges. Intuitively, call-to-return edges are the ones that represent the synchronisation between library method invocations, as illustrated by (MP) in §2.

The scL selector computes the deny part of the history. We let $\mathsf{scL}(X)$ be the projection of $((\mathsf{hb}(X) \cup \mathsf{sc}(X))^+)^{-1}$ to pairs of actions of the form $((\_, \mathsf{ret}\ \_), (\_, \mathsf{call}\ \_))$. This component is needed, since the ACYCLICITY axiom (Figure 4) mandates that sc cannot form a cycle with hb, but does not include sc into hb. Thus, when a library relates a call action u to a return action v with hb and sc, the client cannot relate v to u with the same relations, as this would invalidate ACYCLICITY. We add only return-to-call edges into $\mathsf{scL}(X)$, as these are the edges that represent synchronisation inside the client (similarly to how call-to-return edges represent synchronisation inside the library in $\mathsf{hbL}(X)$). One might think that the deny component of the history should have included edges recording potential violations of other similar axioms, e.g., HBVSMO, as suggested by (DN). However, in the case of the model with relaxed atomics, we are forced to quantify over client happens-before edges R in Definition 2. As it happens, this makes it unnecessary to consider axioms other than ACYCLICITY (see the proof of the Theorem 3 in [1, §C]). These axioms, however, have to be taken into account in the case without relaxed atomics (§6.2).

Library-local semantics. We define the most general client as follows. Take $n \geq 1$ and let $\{m_1, \ldots, m_l\}$ be the methods implemented by a library $\mathcal{L}$. We let

$$\mathsf{MGC}_n(\mathcal{L}) = (\mathsf{let}\ \mathcal{L}\ \mathsf{in}\ C_1^{\mathsf{mgc}} \parallel \ldots \parallel C_n^{\mathsf{mgc}}),$$

where C_t^{mgc} is

$$\mathsf{while}(\mathsf{nondet}())\ \{\ \mathsf{if}(\mathsf{nondet}())\ \{m_1\}\ \mathsf{else\ if}(\mathsf{nondet}())\ \{m_2\}\ \ldots\ \mathsf{else}\ \{m_l\}\ \}$$

Here we use the obvious generalisation of loops and conditionals to branch expressions that yield a non-deterministic value. To allow parameters of methods to be chosen arbitrarily, we replace the axioms DETREAD from Figure 4 and SAFEREAD from Figure 5 by DETREAD$_l$ and SAFEREAD$_l$ from Figure 6, which mandate that reads from param_t lack associated rf edges, while nonetheless yielding identical values within a single method call. As part of the proof of the Theorem 3 (§6.3), we show that this client is indeed most general in a certain formal sense (with the caveat concerning the need to extend its executions with client happens-before edges mentioned in §3).

We note that some libraries require their clients to pass only certain combinations of parameters or issue only certain sequences of method calls. Such contracts could be accommodated in our framework by restricting the most general client appropriately; we do not handle them here so as not to complicate the presentation.

For an initial library state $I \in \mathsf{LLoc} \rightharpoonup_{fin} \mathsf{Val} \times \mathsf{MemOrd}$, a ***library execution*** of $\mathcal{L}$ from I is an execution from $[\![\mathsf{MGC}_n(\mathcal{L})]\!]I$ for some $n \geq 1$. A library execution is ***valid*** if it satisfies the validity axioms with DETREAD$_l$ instead of DETREAD; it is ***safe*** if it satisfies the safety axioms with SAFEREAD$_l$ instead of SAFEREAD. We let $[\![\mathcal{L}]\!]I$ be the set of all valid library executions of $\mathcal{L}$ from I and lift $[\![\mathcal{L}]\!]$ to sets of initial states pointwise. We say that a library $\mathcal{L}$ is safe when run from I if so is every execution in $[\![\mathcal{L}]\!]I$; for a set $\mathcal{I}$ of initial states, $(\mathcal{L}, \mathcal{I})$ is safe if $\mathcal{L}$ is safe when run from any $I \in \mathcal{I}$. The notion of a ***non-interfering*** library is defined similarly.

Extended executions. In Definition 2, we use library executions whose happens-before relation is extended with extra edges recording constraints enforced by the client. Consider an execution X and a relation R over interface actions from $A(X)$. The ***extension*** of X with R is an execution that has the same components as X, except the happens-before relation is replaced by $(\mathsf{hb}(X) \cup R)^+$. An extension of a library execution with R is ***admissible*** when it satisfies the corresponding validity axioms, but with HBDEF replaced by

EXTHBDEF. $\mathsf{hb} = ((\mathsf{sb} \cup \mathsf{ib} \cup \mathsf{sw} \cup R)/\!\sim)^+$.

For an initial library state I, we let $[\![\mathcal{L}, R]\!]I$ be the set of admissible executions of $\mathcal{L}$ from I extended with R. This completes the definition of components used in Definition 2.

Execution projections. Finally, we define the client function used in Theorem 3. Consider a valid execution X of $\mathcal{C}(\mathcal{L})$. An action $u \in A$ is a ***library action***, if it is a call or return action, an action of the form $(0, \mathsf{write}(x, \_))$ for $x \in \mathsf{LLoc}$, or if

$$\exists v. v = (\_, \mathsf{call}\ \_) \wedge v \xrightarrow{\mathsf{sb}(X)} u \wedge \neg\exists q. q = (\_, \mathsf{ret}\ \_) \wedge v \xrightarrow{\mathsf{sb}(X)} q \xrightarrow{\mathsf{sb}(X)} u.$$

An action $u \in A$ is a ***client action***, if it is a call or return action, it is an action of the form $(0, \mathsf{write}(x, \_))$ for $x \in \mathsf{CLoc}$, or the negation of the above property holds. We define the execution $\mathsf{lib}(X)$ by restricting the action set to library actions and projecting all the relations in X accordingly. We use a similar projection $\mathsf{client}(X)$ to client actions and lift client and lib to sets of executions pointwise.

Properties of library abstraction. For the fragment of the language with an SC semantics—i.e., allowing only non-atomic memory accesses and SC atomics—Definition 2 implies classical linearizability. This follows from Theorem 3 and the fact that classical linearizability is equivalent to observational abstraction on the SC memory model [7]. However, the converse is not true: since our notion of library abstraction validates Theorem 3 for clients in the full C11, it distinguishes between SC libraries that classical linearizability would consider equivalent (see [1, §D]).

Using Theorem 3, we can obtain the expected property that, like the classical notion of linearizability, our notion of library abstraction is compositional (with a caveat that non-interference among libraries has to be checked globally). Formally, consider libraries $\mathcal{L}_1, \dots, \mathcal{L}_k$ with disjoint sets of declared methods and assume the splitting of the library address space into regions belonging to each library: $\mathsf{LLoc} = \mathsf{LLoc}_1 \uplus \dots \uplus \mathsf{LLoc}_k$. Consider sets of initial states $\mathcal{I}_1, \dots, \mathcal{I}_k$ such that $\forall j = 1..k.\, \forall I \in \mathcal{I}_j.\, \mathsf{dom}(I) \subseteq \mathsf{LLoc}_j$. We adjust the notion of library safety so that NONINTERF for $\mathcal{L}_j$ is checked with respect to locations in LLoc_j. Let $(\mathcal{L}'_1, \mathcal{I}'_1), \dots, (\mathcal{L}'_k, \mathcal{I}'_k)$ be corresponding library specifications. We define $\mathcal{L}$, respectively, $\mathcal{L}'$ as the library implementing all methods of $\mathcal{L}_1, \dots, \mathcal{L}_k$, respectively, $\mathcal{L}'_1, \dots, \mathcal{L}'_k$ and having the set of initial states $\mathcal{I}_1 \uplus \dots \uplus \mathcal{I}_k$, respectively, $\mathcal{I}'_1 \uplus \dots \uplus \mathcal{I}'_k$. We assume that any combination of implementations or specifications of different libraries is non-interfering. The following theorem is shown by abstracting $\mathcal{L}_j$ to $\mathcal{L}'_j$ one by one using Theorem 3.

THEOREM 4. *If $(\mathcal{L}_j, \mathcal{I}_j) \sqsubseteq (\mathcal{L}'_j, \mathcal{I}'_j)$ for $j = 1..k$, then $\mathcal{L} \sqsubseteq \mathcal{L}'$.*

6.2 Library Abstraction without Relaxed Atomics

We call an action with at least one RLX annotation ***relaxed***. In this section, we restrict ourselves to programs whose action structures do not have any relaxed actions, and we augment the C11 thread-local semantics as described in the assumptions of §4. Among other things, these changes allow us to remove the quantification over client happens-before edges R from Definition 2, at the expense of including an additional deny relation into the history.

DEFINITION 5. *An **extended history** is a quadruple (A, G, D, D'), where A is a set of interface actions, and $G, D, D' \subseteq A \times A$.*

For an execution $X = (A, \mathsf{sb}, \mathsf{ib}, \mathsf{rf}, \mathsf{sc}, \mathsf{mo}, \mathsf{sw}, \mathsf{hb})$, we let

$$\mathsf{Ehistory}(X) = (\mathsf{interf}(X), \mathsf{hbL}(X), \mathsf{scL}(X), \mathsf{denyL}(X)),$$

which we lift to sets of executions pointwise. The selectors interf, hbL and scL are defined as in §6.1.

The relation $\mathsf{denyL}(X)$ is defined similarly to scL, but whereas the latter records client hb and sc edges that can violate ACYCLICITY, the former includes the client hb edges that can violate the axioms HBVSMO, COWR and SCREADS (Figure 4). We do not have to consider other similar axioms, such as RFATOMIC, CORW and CORR since in any valid execution X without relaxed actions, we have $\mathsf{rf}(X) \subseteq \mathsf{hb}(X)$. Due to this, the hb client edges violating, HBVSMO and CORW, COWR and CORR can be covered by the same relations. For the case of HBVSMO and COWR, $\mathsf{denyL}(X)$ includes the dashed edges of all possible instantiations of the following diagrams:

$$\begin{tikzcd}
w_1 \arrow[r, "\mathsf{hb}"] & u \arrow[r, dotted] & v \arrow[r, "\mathsf{hb}"] & w_2 \arrow[lll, bend left, "\mathsf{mo}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
w_1 \arrow[r, "\mathsf{mo}"] \arrow[rddd, "\mathsf{rf}"'] & w_2 \arrow[d, "\mathsf{hb}"] \\
 & u \arrow[d, dotted] \\
 & v \arrow[d, "\mathsf{hb}"] \\
 & r
\end{tikzcd}$$

where $u = (\_, \mathsf{ret}\ \_)$ and $v = (\_, \mathsf{call}\ \_)$. This records client hb edges that violate HBVSMO or COWR (cf. (DN) in §3). The diagrams are thus constructed systematically by 'breaking' the hb edge. For the case of SCREADS, $\mathsf{denyL}(X)$ includes edges corresponding to a corner case in this axiom omitted from Figure 4. The full version of the axiom and the corresponding deny diagram are given in §A.

DEFINITION 6. *We let $(A_1, G_1, D_1, D'_1) \preceq (A_2, G_2, D_2, D'_2)$, if $A_1 = A_2$, $G_2 \subseteq G_1$, $D_2 \subseteq D_1$ and $D'_2 \subseteq D'_1$.*

*For safe $(\mathcal{L}_1, \mathcal{I}_1)$ and $(\mathcal{L}_2, \mathcal{I}_2)$, $(\mathcal{L}_1, \mathcal{I}_1)$ **is abstracted by** $(\mathcal{L}_2, \mathcal{I}_2)$, written $(\mathcal{L}_1, \mathcal{I}_1) \preceq (\mathcal{L}_2, \mathcal{I}_2)$, if*

$$\forall I_1 \in \mathcal{I}_1, H_1 \in \mathsf{Ehistory}([\![\mathcal{L}_1]\!] I_1).$$
$$\exists I_2 \in \mathcal{I}_2, H_2 \in \mathsf{Ehistory}([\![\mathcal{L}_2]\!] I_2).\, H_1 \preceq H_2.$$

Unlike in Definition 2, here an abstracted history can guarantee *fewer* happens-before edges to the client: without relaxed atomics, removing edges from happens-before can only permit more client behaviours or make the client unsafe. We note that checking the inclusion between the components of the history given by denyL, required by Definition 6, is simpler in practice than quantifying over client happens-before edges R in Definition 2.

For executions X and Y, we write $X \preceq Y$ when all their components except hb are equal, and $\mathsf{hb}(Y) \subseteq \mathsf{hb}(X)$.

THEOREM 7 (Abstraction without relaxed atomics). *Assume that $(\mathcal{L}_1, \mathcal{I}_1)$, $(\mathcal{L}_2, \mathcal{I}_2)$ and $(\mathcal{C}(\mathcal{L}_2), \mathcal{I} \uplus \mathcal{I}_2)$ are safe and $(\mathcal{L}_1, \mathcal{I}_1) \preceq (\mathcal{L}_2, \mathcal{I}_2)$. Then $(\mathcal{C}(\mathcal{L}_1), \mathcal{I} \uplus \mathcal{I}_1)$ is safe and*

$$\forall X \in \mathsf{client}([\![\mathcal{C}(\mathcal{L}_1)]\!](\mathcal{I} \uplus \mathcal{I}_1)).$$
$$\exists Y \in \mathsf{client}([\![\mathcal{C}(\mathcal{L}_2)]\!](\mathcal{I} \uplus \mathcal{I}_2)).\, X \preceq Y.$$

Unlike Theorem 3, this one allows the programmer to check non-interference on $\mathcal{C}(\mathcal{L}_2)$—i.e., with respect to a library specification—and to conclude that $\mathcal{C}(\mathcal{L}_1)$ is non-interfering. Since the abstract library can have a smaller guarantee than the concrete one, the happens-before of the execution $\mathcal{C}(\mathcal{L}_2)$ may also be smaller than that of the execution $\mathcal{C}(\mathcal{L}_1)$.

Like the notion of library abstraction from §6.1, the one proposed here implies classical linearizability for the SC fragment of the language (but not vice versa) and is compositional [1, §C]. However, here the latter property does not require us to check non-interference globally: a composition of several non-interfering libraries is non-interfering.

6.3 Proof Outlines

Client-local semantics. We start by defining the ***client-local semantics*** of a client $\mathcal{C} = C_1 \parallel \dots \parallel C_n$, which is a counterpart of the library-local semantics defined in §6.1. Let M be the set of methods that can be called by $\mathcal{C}$. Consider the program

$$\mathcal{C}(\cdot) = (\mathsf{let}\ \{m = \mathsf{skip} \mid m \in M\}\ \mathsf{in}\ C_1 \parallel \dots \parallel C_n),$$

where every method is implemented by a stub that returns immediately after having been called. Moreover, we allow methods to return arbitrary values by replacing the axioms DETREAD from Figure 4 and SAFEREAD from Figure 5 by DETREAD$_c$ and SAFEREAD$_c$ from Figure 6. We call executions of the above program ***client executions*** of $\mathcal{C}$. A client execution is ***valid***, if it satisfies the validity axioms with DETREAD$_c$ instead of DETREAD; it is ***safe***, if it satisfies the safety axioms with SAFEREAD$_c$ instead of SAFEREAD. For an initial client state $I \in \mathsf{CLoc} \rightharpoonup_{fin} (\mathsf{Val} \times \mathsf{MemOrd})$, let $[\![\mathcal{C}]\!] I$ be the set of all valid client executions of $\mathcal{C}$ from I; for a set $\mathcal{I}$ of initial states we define $[\![\mathcal{C}]\!]\mathcal{I}$ as expected. The notion of an extended client execution is similar to the one of an extended library execution from §6.1. For an extended execution

X we let $\mathsf{core}(X)$ be the execution obtained from X by recomputing $\mathsf{hb}(X)$ from $\mathsf{sb}(X)$, $\mathsf{ib}(X)$ and $\mathsf{sw}(X)$ according to HBDEF.

Client-side history selectors. Consider a client execution X. We define $\mathsf{hbC}(X), \mathsf{scC}(X) \subseteq \mathsf{interf}(X) \times \mathsf{interf}(X)$, which are analogous to hbL and scL from §6.1, but select the information about the client part of the execution that is relevant to the library. We let $\mathsf{hbC}(X)$ be the projection of $\mathsf{hb}(X)$ to pairs of actions of the form $((\_, \mathsf{ret}\ \_), (\_, \mathsf{call}\ \_))$ and pairs of calls and returns (in any order) by the same thread. We select edges of this form as they are the ones that record synchronisation enforced by the client. We let $\mathsf{scC}(X)$ be the projection of $((\mathsf{hb}(X) \cup \mathsf{sc}(X))^+)^{-1}$ to pairs of actions of the form $((\_, \mathsf{call}\ \_), (\_, \mathsf{ret}\ \_))$.

Proof outline for Theorem 3. Consider an execution X of $\mathcal{C}(\mathcal{L}_1)$ from the initial state $I \uplus I_1$, where $I \in \mathcal{I}$ and $I_1 \in \mathcal{I}_1$. We start by decomposing X into a client execution $\mathsf{client}(X)$ and a library execution $\mathsf{lib}(X)$ and showing that

$$\mathsf{client}(X) \in [\![\mathcal{C},\ \mathsf{hbL}(\mathsf{core}(\mathsf{lib}(X)))]\!]I;$$
$$\mathsf{lib}(X) \in [\![\mathcal{L}_1,\ \mathsf{hbC}(\mathsf{core}(\mathsf{client}(X)))]\!]I_1.$$

The second inclusion justifies that the most general client of the library defined in §6.1 is indeed most general, as it can reproduce the behaviour of $\mathcal{L}_1$ under any client $\mathcal{C}$. This comes with the caveat that an execution of the most general client of $\mathcal{L}_1$ has to be extended with $\mathsf{hbL}(\mathsf{core}(\mathsf{lib}(X)))$ to obtain $\mathsf{lib}(X)$, as the library-local semantics does not generate happens-before edges enforced by client synchronisation (and similarly for $\mathsf{client}(X)$ in the first inclusion). The above decomposition step relies on X being non-interfering. Using the fact that $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$, we prove that there exist $I_2 \in \mathcal{I}_2$ and $Y \in [\![\mathcal{L}_2, \mathsf{hbC}(\mathsf{core}(\mathsf{client}(X)))]\!]I_2$ such that $\mathsf{history}(\mathsf{lib}(X)) \sqsubseteq \mathsf{history}(Y)$. Here we use the quantification of client happens-before edges R in Definition 2 to handle the extension of the library execution with $\mathsf{hbC}(\mathsf{core}(\mathsf{client}(X)))$. We then compose the executions X and Y into the desired execution of $\mathcal{C}(\mathcal{L}_2)$. This step uses the fact that the deny component of $\mathsf{history}(Y)$ is smaller than that of $\mathsf{history}(\mathsf{lib}(X))$. □

Proof outline for Theorem 7. Unlike in Theorem 3, here we need to consider the case when an execution X of $\mathcal{C}(\mathcal{L}_1)$ has actions violating non-interference. In this case, we identify an "earliest" faulting action u and construct a valid execution that is a prefix of X ending just before u and is thus non-interfering. This is only possible because, without relaxed atomics, we do not have satisfaction cycles, and because the theorem is stated over an augmented thread-local semantics (§4), with prefix executions added to the semantics. We convert the resulting execution into one of $\mathcal{C}(\mathcal{L}_2)$ as in the proof of Theorem 3 and conjoin the action the action u to it. This yields an execution of $\mathcal{C}(\mathcal{L}_2)$ violating non-interference and contradicting the assumption about its safety.

In the case when X is non-interfering, the proof is similar to that of Theorem 3. Some additional work is needed to deal with the fact that Definition 6 does not have a quantification over client happens-before edges and allows the abstract guarantee to be smaller than the concrete one. □

7. Establishing Library Abstraction

In this section, we discuss the proof process for establishing abstraction between libraries in the sense of Definitions 2 and 6. To reason about programs on the C11 memory model, we use axiomatisations of the action structures generated by them, which give a simple mathematical interface to the program semantics. To prove library safety, required by Definitions 2 and 6, we consider all the execution shapes of the most general client, and check these against the C11 safety axioms. We now explain how we prove the correspondence between the executions of concrete and abstract libraries, using Definition 2 for illustration.

Effect points. Consider an execution $X_1 \in [\![\mathcal{L}_1, R]\!]I_1$. We construct an execution $X_2 \in [\![\mathcal{L}_2, R]\!]I_2$ whose history witnesses the existential in Definition 2 using an adapted version of the *linearization point* method for proving linearizability. The method constructs the abstract execution by calling a method specification at a fixed linearization point in every method invocation of the concrete execution; intuitively, it is at this point that the concrete method 'takes effect'. In our adaptation, we construct X_2 by substituting calls to library method implementations in X_1 for the corresponding calls to specifications, and choosing appropriate values for reads and different orders between actions. These values and orders are chosen based on the orders over actions we call ***effect points***, picked for each concrete method invocation in X_1. Thus, the various partial orders over the effect points in the concrete execution dictate the order of precedence for the effects of method invocations in the abstract execution. In contrast with the original linearization point method, the latter order does not have to be linear. We now explain this technique in more detail on an example.

Example: Treiber stack. We have proved the correctness of the stack in Figure 1b with respect to its specification in Figure 1a according to Definition 2 (full details are given in [1, §E]). As effect points in X_1, we pick the `rmw` actions modifying the stack's top pointer `T`, which correspond to successful CASes (this is the same choice as the that of linearization points when proving the linearizability of Treiber's stack on an SC memory model). The order $\mathsf{mo}(X_1)$ over these actions defines a total order over successful `push` and `pop` invocations. When substituting invocations of method implementations for invocations of specifications, we use $\mathsf{mo}(X_1)$ to decide $\mathsf{rf}(X_2)$ and $\mathsf{mo}(X_2)$ for the abstract location `S`. Namely, suppose we have two method invocations U and V in X_1, such that the `rmw` of U is the immediate predecessor to the `rmw` of V in $\mathsf{mo}(X_1)$. When substituting corresponding invocations of specification methods U' and V', we set up $\mathsf{rf}(X_2)$ and $\mathsf{mo}(X_2)$ so that the load from `S` in V' reads from the `rmw` in U', and $\mathsf{mo}(X_2)$ orders the `rmw` on `S` in U' right before `rmw` on it in V'. Fixing $\mathsf{rf}(X_2)$ and $\mathsf{mo}(X_2)$ in the abstract execution immediately fixes the values of reads, which finishes the construction of X_2.

Example: producer-consumer queue. We have similarly verified a non-blocking producer-consumer queue according to Definition 6 (see [1, §E]). The queue is intended for communication between a single producer thread and a single consumer thread and provides three methods: `init`, `enq` and `deq`. The implementation stores values in a finite cyclic array, while the specification stores them using the abstract data type of a sequence. To ensure that the consumer calling `deq` observes up-to-date values, it must synchronise with the producer calling `enq` using release-acquire atomics.

8. Related Work

Relaxed-memory behaviour has become widespread in real concurrent systems. As a result, some algorithm designers have begun to publish algorithms with memory-model annotations such as fences [3, 17, 18]. However, the corresponding formulations of correctness properties and their proofs are generally informal. While there has been some work on formally verifying programs on weak memory models [14, 20, 23], none has proposed a compositional reasoning method, like we do. Ours is the first approach that formulates the notion of a correct library specification and provides a method for establishing it on C11, or any similarly relaxed model.

Our work is an evolution of *linearizability* [11], a correctness criterion that has been widely adopted in the concurrent algorithms community. In the SC fragment of C11, our definition of library abstraction implies classical linearizability. History abstraction in classical linearizability is defined by linearization of the partial order over non-overlapping methods invocations, and the guarantee portion in our histories can be seen as lifting this to the C11 setting. Classical linearizability has no equivalent to our deny relation; the conflicts between relations that are captured by deny do not occur in an SC setting where all events are totally ordered.

Recent work has formalised the intuition that linearizability corresponds to observational abstraction [7] and has extended it to handle liveness [9], resource-transferring programs [10] and the x86 memory model [5]. The latter work is the closest to this paper; in particular, we borrow the decompose-compose approach in the proof of the Abstraction Theorem (§6.3) from it. However, while its objective is the same as ours—abstraction for relaxed-memory concurrent libraries—the technical challenges and the machinery developed to address them are very different. The x86 memory model can be defined by a small-step operational semantics, which has an underlying total order on abstract machine memory events [19]. Linearizability for x86 is therefore a relatively mild extension to classical linearizability, which simply represents some of these events in (linear) histories. In contrast, C11 constrains relaxed behaviour through partial ordering, controlled by an axiomatic semantics. There are no abstract machine events to linearize, which motivates our novel definition of a history as a set of partial orders and history abstraction as inclusion over them. Furthermore, x86 is a substantially stronger model than C11, with (SB) from §2 being the only significant relaxation [19]. Our approach is the first technique for specifying client-visible effects of relaxations inside libraries on weaker memory models.

9. Conclusion and Future Work

We have proposed the first sound criterion for library abstraction suitable for the C11 memory model and demonstrated its practicality on two small, but typical, relaxed libraries. Our criterion is certainly complex, but much of this complexity arises from the real-world intricacies of the C11 memory model. In turn, the complexity of the model arises from a multitude of target platforms of C and C++—two of the world's most widely-used programming languages. Despite this complexity, the criterion allows developers to establish that C11 libraries satisfy simple, reusable specifications that precisely describe the level of consistency guaranteed. This is an essential ingredient for supporting modular development of complex software on relaxed memory models.

In addition, our approach is the first compositional reasoning technique for an axiomatically defined relaxed memory model, and highlights general principles for abstraction on such models. We have good reason to believe that our techniques can be reused for other memory models, as the conditions for library abstraction fall out naturally from obligations arising when trying to prove the Abstraction Theorem according to the approach in §6.3. In particular, the histories in our approach are constructed uniformly, with deny relations obtained straightforwardly from axioms by 'breaking' hb edges (§6.2). In particular, our preliminary investigations show that these techniques can be used to define a notion of library abstraction for an axiomatic formulation of the x86 memory model [19].

Our specifications describe precisely the level of synchronisation provided by the library, although in some cases this makes them more verbose. This is motivated by the fact that libraries on C11 can offer relaxed interfaces to clients, without either giving up all synchronisation guarantees or enforcing sequential consistency. If information about the internal synchronisation used to ensure library correctness were not described in these interfaces, clients would have to duplicate it, thereby decreasing performance. On the other hand, requiring library interfaces to be SC would rule out libraries that use weaker memory orders to achieve efficiency while preserving basic correctness properties, again decreasing performance. Our prototype atomic section semantics represents a first attempt at a syntactic specification idiom for relaxed algorithms, albeit with limitations as described in §4. We leave a more comprehensive treatment of relaxed atomic sections to future work.

Our two formulations of library abstraction (Definitions 2 and 6) and the Abstraction Theorem (Theorems 3 and 7) identify the feature of the current C/C++ memory model that does not allow fully compositional reasoning about libraries. As we argued in §3, this deficiency is not specific to our definition of library abstraction, but would be inherent to *any* sensible one. We hope that these insights will inform future revisions of the C/C++ memory model.

Our development omits memory fences and release-consume atomics, which are the more advanced features of the C11 memory model. A memory fence is a synchronisation construct that affects many memory actions, rather than just one. Release-consume is a special-purpose memory order which compiles more efficiently to Power and ARM processors. We conjecture that our methods can be used to handle these features. As both of them generate more possible client-library interactions, this will require adding additional relations to histories.

To concentrate on the core challenges of library abstraction in C11, we assumed that the data structures of the client and its libraries are completely disjoint (§4). We hope to lift this restriction by combining our results with a recent generalisation of classical linearizability allowing transfers of memory ownership [10]. Similarly, a previous generalisation of linearizability to handle liveness properties [9] could be used to strengthen specifications of the kind shown in Figure 1a to guarantee properties such as lock-freedom.

Acknowledgements. We would like to thank Hans Boehm, Richard Bornat, Paul McKenney, Robin Morisset, Madan Musuvathi, Peter Sewell, Jaroslav Ševčík, Hongseok Yang and John Wickerson for helpful comments. We acknowledge funding from EPSRC grants EP/F036345 and EP/H005633.

References

[1] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency (extended version). University of York Technical Report YCS-2012-479, 2012.

[2] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[3] H.-J. Boehm. Can seqlocks get along with programming language memory models? In *MSPC*, 2012.

[4] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[5] S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, 2012.

[6] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[7] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *ESOP*, 2009.

[8] I. Filipović, P. O'Hearn, N. Torp-Smith, and H. Yang. Blaiming the client: On data refinement in the presence of pointers. *FAC*, 22, 2010.

[9] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP*, 2011.

[10] A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR*, 2012.

[11] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12, 1990.

[12] ISO/IEC. *Programming Languages – C++, 14882:2011.*

[13] ISO/IEC. *Programming Languages – C, 9899:2011.*

[14] M. Kuperstein, M. T. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, 2011.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, 28, 1979.

[16] J. Manson. The Java memory model. PhD Thesis. Department of Computer Science, University of Maryland, 2004.

Action structures for locks

$$\langle \mathsf{lock}(\ell)\rangle_t \quad = \{(\{(e,g,t,\mathsf{lock}(\ell))\},\emptyset),(\{(e,g,t,\mathsf{block}(\ell))\},\emptyset) \mid e \in \mathsf{AId} \wedge g \in \mathsf{SectId}\}$$
$$\langle \mathsf{unlock}(\ell)\rangle_t = \{(\{(e,g,t,\mathsf{unlock}(\ell))\},\emptyset) \mid e \in \mathsf{AId} \wedge g \in \mathsf{SectId}\}$$
$$\begin{aligned}\langle \mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n\rangle I = \big\{&(A_0 \uplus (\textstyle\biguplus_{t=1}^n A_t), \bigcup_{t=1}^n \mathsf{sb}_t, (A_0 \times (\biguplus_{t=1}^n A_t))) \mid \\ &(\forall t = 1..n.\, (A_t, \mathsf{sb}_t) \in \langle C_t\rangle_t) \wedge (\forall t = 1..n.\, \forall u.\, \exists \text{ finitely many } v.\, (v,u) \in \mathsf{sb}_t) \wedge \\ &(\neg\exists u,v,t.\, (u,v) \in \mathsf{sb}_t \wedge u = (\_,\_,\_,\mathsf{block}(\_))) \wedge (A_0 = \textstyle\biguplus\{(e,g,0,\mathsf{store}_\lambda(x,a)) \mid I(x) = (a,\lambda) \wedge e \in \mathsf{AId} \wedge g \in \mathsf{SectId}\})\big\}\end{aligned}$$

Additional validity axioms

ATOMRMW. $\forall w,u.\, w \xrightarrow{\mathsf{rf}} u \wedge u = (\_,\mathsf{rmw}(\_)) \implies w \xrightarrow{\mathsf{mo}} u \wedge \neg\exists w'.\, w \xrightarrow{\mathsf{mo}} w' \xrightarrow{\mathsf{mo}} u$

ATOMAS. $\forall w,w',r,u,v,x.\, w \xrightarrow{\mathsf{rf}} r \wedge \mathsf{sec}(u) = \mathsf{sec}(r) \neq \mathsf{sec}(w) = \mathsf{sec}(v) \wedge r = (\_,\mathsf{read}(x,\_)) \wedge \mathsf{sort}(x) = \mathsf{ATOM} \implies \neg(w \xrightarrow{\mathsf{mo}} v)$
$\wedge\, (u = (\_,\mathsf{write}(x,\_)) \implies w \xrightarrow{\mathsf{mo}} u \wedge \neg\exists w''.\, \mathsf{sec}(w'') \neq \mathsf{sec}(u) \wedge w \xrightarrow{\mathsf{mo}} w'' \xrightarrow{\mathsf{mo}} u)$
$\wedge\, (u = (\_,\mathsf{read}(x,\_)) \wedge w' \xrightarrow{\mathsf{rf}} r \wedge \mathsf{sec}(w') \neq \mathsf{sec}(r) \implies w'' = w)$

LOCKS. $\forall u,v.\, u = (\_,\mathsf{lock}(\ell)) \wedge v = (\_,\mathsf{lock}(\ell)) \wedge u \xrightarrow{\mathsf{sc}} v \implies \exists q.\, q = (\_,\mathsf{unlock}(\ell)) \wedge u \xrightarrow{\mathsf{sc}} q \xrightarrow{\mathsf{sc}} v$

SWDEF. $\forall w,r.\, w \xrightarrow{\mathsf{sw}} r \iff (\exists \ell.\, w \xrightarrow{\mathsf{sc}} r \wedge w = (\_,\mathsf{unlock}(\ell)) \wedge r = (\_,\mathsf{lock}(\ell))) \vee$
$(\exists t_1,t_2,\lambda,\mu,x,w'.\, t_1 \neq t_2 \wedge \lambda \in \{\mathsf{SC},\mathsf{REL}\} \wedge \mu \in \{\mathsf{SC},\mathsf{ACQ}\} \wedge w = (t_1,\mathsf{write}_\lambda(x,\_)) \wedge r = (t_2,\mathsf{read}_\mu(x,\_)) \wedge w \xrightarrow{\mathsf{rs}}_{t_1} w' \xrightarrow{\mathsf{rf}} r)$,
where $w \xrightarrow{\mathsf{rs}}_t w' \overset{\mathrm{def}}{\iff} \exists w_1.\, w \xrightarrow{\mathsf{mo}}{}^* w_1 \wedge (\forall w_2.\, w \xrightarrow{\mathsf{mo}}{}^* w_2 \xrightarrow{\mathsf{mo}}{}^* w' \implies (w_2 = (t,\_) \vee w_2 = (\_,\mathsf{rmw}(\_))))$

SCREADS.
$\forall w,r,x.\, w \xrightarrow{\mathsf{rf}} r \wedge r = (\_,\mathsf{read}_{\mathsf{SC}}(x,\_)) \implies$
$((w = (\_,\mathsf{write}_{\mathsf{SC}}(x,\_)) \wedge w \xrightarrow{\mathsf{sc}} r \wedge \neg\exists w'.\, w' = (\_,\mathsf{write}(x,\_)) \wedge w \xrightarrow{\mathsf{sc}} w' \xrightarrow{\mathsf{sc}} r) \vee$
$(\exists \lambda.\, w = (\_,\mathsf{write}_\lambda(x,\_)) \wedge \lambda \neq \mathsf{SC} \wedge \forall w_1.\, (w_1 = (\_,\mathsf{write}(x,\_)) \wedge w_1 \xrightarrow{\mathsf{sc}} r \wedge \neg\exists w_2.\, w_2 = (\_,\mathsf{write}(x,\_)) \wedge w_1 \xrightarrow{\mathsf{sc}} w_2 \xrightarrow{\mathsf{sc}} r) \implies$
$\neg(w \xrightarrow{\mathsf{hb}} w_1)))$

Additional safety axiom

SAFELOCK. $(\forall v,t,\ell.\, v \in A \wedge v = (t,\mathsf{unlock}(\ell)) \implies \exists u.\, u = (t,\mathsf{lock}(\ell)) \wedge u \xrightarrow{\mathsf{sb}} v \wedge \neg\exists q.\, q = (\_,\_(\ell)) \wedge u \xrightarrow{\mathsf{sc}} q \xrightarrow{\mathsf{sc}} v) \wedge$
$(\neg\exists u,v,t,\ell.\, u = (t,\mathsf{lock}(\ell)) \wedge v = (t,\mathsf{block}(\ell)) \wedge u \xrightarrow{\mathsf{sb}} v \wedge \neg\exists q.\, q = (\_,\_(\ell)) \wedge u \xrightarrow{\mathsf{sc}} q \xrightarrow{\mathsf{sc}} v)$

Figure 7. Action structures for locks and additional C11 memory model axioms

[17] M. M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI*, 2004.

[18] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *PPOPP*, 2009.

[19] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

[20] T. Ridge. A rely-guarantee proof system for x86-TSO. In *VSTTE*, 2010.

[21] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.

[22] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[23] I. Wehrman and J. Berdine. A proposal for weak-memory local reasoning. In *LOLA*, 2011.

A. Additional Definitions for the C11 Model

In §4–5, we omitted the treatment of locks and CASes from the description of the memory model and simplified some of the axioms, even though the proofs of our theorems do not make such simplifications. Here we provide the missing definitions.

We handle programs with bounded numbers of locks $\ell \in \mathsf{Lock}$, acquired and released using commands $\mathsf{lock}(\ell)$ and $\mathsf{unlock}(\ell)$, respectively. We thus extend the set of actions as follows: $\varphi ::= \ldots \mid \mathsf{lock}(\ell) \mid \mathsf{unlock}(\ell) \mid \mathsf{block}(\ell)$, where $\ell \in \mathsf{Lock}$. An action $(e,g,t,\mathsf{block}(\ell))$ represents a deadlocked attempt to acquire a lock ℓ. We split the set of locks into client and library ones ($\mathsf{Lock} = \mathsf{CLock} \uplus \mathsf{LLock}$) and consider only programs where the client and the library use locks from CLock and LLock, respectively.

In Figure 7, we give the actions structures for lock operations, omitted from Figure 3, and the C11 axioms omitted from Figures 4 and 5. Note that the action structures of a whole program do not include those where a thread executes actions after blocking. To adjust the axioms to the model with locks, we require that the sc relation totally orders actions of the form $(\_,\mathsf{lock}(\_))$, $(\_,\mathsf{block}(\_))$ or $(\_,\mathsf{unlock}(\_))$ for each lock, in addition to SC actions. The axioms ATOMRMW and LOCKS define the behaviour of CAS commands and locks. ATOMAS is an additional axiom for atomic sections, in the spirit of ATOMRMW. SWDEF and SCREADS are full versions of the axioms presented in a simplified form in Figure 4. The former adds synchronisation via ***release sequences*** [2] (defined by the rs relation), and the latter allows SC reads from non-SC writes. SAFELOCK is an additional safety axiom, flagging a double unlock or a double lock in the same thread as a fault.

All the theorems stated in the paper stay valid for this extension of the model. To account for the full version of the SCREADS axiom, we add additional edges to $\mathsf{denyL}(X)$. For $X = (A,\mathsf{sb},\mathsf{ib},\mathsf{rf},\mathsf{sc},\mathsf{mo},\mathsf{sw},\mathsf{hb})$, $\mathsf{denyL}(X)$ includes the dashed edges of all possible instantiations of the following diagram:

$$\begin{pmatrix} w = (\_,\mathsf{write}_\lambda(x,\_)) \wedge \lambda \neq \mathsf{SC} \wedge \\ r = (\_,\mathsf{read}_{\mathsf{SC}}(x,\_)) \wedge \\ w_1 = (\_,\mathsf{write}(x,\_)) \wedge \\ (\neg\exists w_2.\, w_2 = (\_,\mathsf{write}(x,\_)) \wedge \\ w_1 \xrightarrow{\mathsf{sc}} w_2 \xrightarrow{\mathsf{sc}} r) \end{pmatrix} \wedge$$

$$\begin{tikzcd} r & w \arrow[l, "\mathsf{rf}"'] \arrow[d, "\mathsf{hb}"] \\ & u \arrow[d, dashed] \\ & v \arrow[d, "\mathsf{hb}"] \\ & w_1 \arrow[uuul, "\mathsf{sc}"] \end{tikzcd}$$

where u is a return, and v is a call. Its counterpart $\mathsf{denyC}(X)$ contains the dashed edges assuming u is a call, and v is a return.