An Operational Approach to Library Abstraction under Relaxed Memory Concurrency (Extended Version)

ABHISHEK KR SINGH, Tel Aviv University, Israel
ORI LAHAV, Tel Aviv University, Israel

Concurrent data structures and synchronization mechanisms implemented by expert developers are indispensable for modular software development. In this paper, we address the fundamental problem of library abstraction under weak memory concurrency, and identify a general library correctness condition allowing clients of the library to reason about program behaviors using the specification code, which is often much simpler than the concrete implementation. We target (a fragment of) the RC11 memory model, and develop an equivalent operational presentation that exposes knowledge propagation between threads, and is sufficiently expressive to capture library behaviors as totally ordered operational execution traces. We further introduce novel access modes to the language that allow intricate specifications accounting for library internal synchronization that is not exposed to the client, as well as the library's demands on external synchronization by the client. We illustrate applications of our approach in several examples of different natures.

CCS Concepts: • **Theory of computation → Concurrency**; **Operational semantics**; **Program verification**; • **Software and its engineering → Semantics**; **Software verification**; **Abstraction, modeling and modularity**.

Additional Key Words and Phrases: Relaxed memory consistency, Concurrent objects, Linearizability, Library abstraction

ACM Reference Format:
Abhishek Kr Singh and Ori Lahav. 2023. An Operational Approach to Library Abstraction under Relaxed Memory Concurrency (Extended Version). *Proc. ACM Program. Lang.* 7, POPL, Article 53 (January 2023), 45 pages. https://doi.org/10.1145/3571246

1 INTRODUCTION

53

Library abstraction constitutes a powerful means to achieve modularity in software development. It allows expert library developers to write optimized implementations of common programming tasks, and "once and for all" establish that these implementations admit their corresponding specifications. In turn, users of these implementations, called *clients* of the library, may reason about program behaviors assuming only the libraries' specifications, with no understanding, or even no access, to the implementations. From a formal standpoint, a perquisite for applying library abstraction is to identify a condition that provably allows library developers to abstract away from a particular client program when verifying their implementations, while ensuring the soundness of client reasoning using the specifications in any (valid) client program.

In the case of sequential programs, library abstraction is straightforward: specifications may be given using pre- and post-conditions, and it is easy to establish the soundness of client reasoning that is based on them. Concurrent programs, however, require more attention. Assuming

Authors' addresses: Abhishek Kr Singh, Tel Aviv University, Israel, abhishek.uor@gmail.com; Ori Lahav, Tel Aviv University, Israel, orilahav@tau.ac.il.

2475-1421/2023/1-ART53
https://doi.org/10.1145/3571246

an underlying sequentially consistent (SC) memory system [Lamport 1979], classical linearizability (w.r.t. some sequential specification) [Herlihy and Wing 1990] ensures refinement w.r.t. a sequential object [Filipović et al. 2010], and can be seen as a library abstraction condition. More recent approaches employ "code as specification" (see, e.g., [Gotsman and Yang 2011]), where implementations are specified using (simple) code, and library abstraction amounts to contextual refinement between two pieces of code. Linearizability can be seen as a particular instance of this approach, where specifications are obtained by wrapping a sequential implementation within a global lock [Bouajjani et al. 2015]. Furthermore, to make such specification formalism more useful, one often enriches the given programming language with special specification constructs, that are employed only in the specification code and are relatively easy to reason about (e.g., atomic blocks in concurrent programs).

The current paper studies library abstraction under relaxed (a.k.a. weak) shared-memory concurrency. In particular, we consider a C11-style weak memory model with several kinds of memory-access modes (a.k.a. memory orderings), placed differently on the spectrum between efficient implementation (with an optimizing compiler targeting a modern multicore hardware) and strong consistency guarantees. The infamous complexity of programming under such a model makes library abstraction indispensable. In particular, library abstraction allows clients to use the programming guarantees supplied by the model, e.g., clients should be able to rely on the synchronization induced by a library (such as a lock library) for ensuring data-race freedom in their programs, and then apply the model's guarantee that data-race free programs have strong semantics.

Remark 1. Like many formal verification frameworks for C11, we are unable to work with the original model, which allows unrestricted cycles in the union of 'program order' and 'reads-from', and thus exhibits "out-of-thin-air" behaviors and fails to provide the most basic data-race-freedom guarantee. We follow [Boehm and Demsky 2014] and its formalization in the RC11 model in [Lahav et al. 2017] to conservatively disallow *all* such cycles. This entails a certain performance penalty for maintaining the load-store order between relaxed accesses [Ou and Demsky 2018]. We also note that the fragment of the RC11 handled in this paper lacks release/acquire fences and sequentially consistent accesses, which are left to future work.

Our main contribution is a correctness condition for libraries that provably ensures (contextual) refinement between implementations and their respective specifications under the weak memory model, with several distinctive properties:

- Our proposed condition is based on *totally ordered execution traces* of the library in question (a decision motivated below). Accordingly, we present a novel equivalent operational version of the (originally) declarative memory model, and use it in the correctness condition.
- To account for various restrictions that libraries may impose on their clients (e.g., refrain from data races between two specific methods), our result supports rich library calling policies, and refinement is conditioned on the client's adherence to the calling policy. Importantly, whether a program adheres to the policy or not is checked against the library specification, which allows the application of the abstraction theorem by the client without any knowledge of the implementation.
- Our specification language allows simple *lock-based* specifications of libraries that do not provide synchronization guarantees for their clients, and dually of libraries that rely on the client for performing the synchronization between library calls. For that matter, we introduce a novel access mode to the programming language (and memory model).

We illustrate the application of our approach for an RCU synchronization mechanism which we specify by relying solely on locks (§8.1), as well as for a relaxed concurrent queue object that does not expose its internal synchronization to its clients (§8.2). We also derive a (local) data-race-freedom guarantee for the memory model as an instance of library abstraction (§8.3).

Outline. This paper is organized as follows. In §2 we present an informal overview of the challenges we address and our solutions. In §3 we define concurrent programs and their semantics independent of a memory model. In §4 we present the memory model semantics, which we call dRC11 (d for 'declarative'), as a fragment of RC11 extended with novel constraints for the new access modes. In §5 we present an operational version of the memory model, which we call pRC11 (p for 'propagation'), that is needed for defining the library correctness condition. In §6 we introduce the notions required to formulate the library abstraction theorem. In §7 we state and prove the main theorem. In §8 we demonstrate applications of the abstraction theorem (which can assist in understanding the crux of the abstraction theorem even before reading the more technical material in Section 3 to 7). We discuss the relation to other work in §9 and conclude in §10. Additional detailed proofs are given in §A.

2 KEY CHALLENGES AND IDEAS

We outline the main challenges and the key ideas in our solutions. We keep the discussion and examples informal, leaving the formal development to later sections.

2.1 Library Correctness Criterion

The main challenge lies in establishing a library correctness condition that will ensure contextual refinement between a library specification $L^{\#}$ (given as code) and its implementation L under weak memory semantics. Naturally, such condition should consider each object in isolation and avoid quantification over all possible clients. Moreover, we opt for an *operational* condition that, like standard linearizability, is based on totally ordered histories generated by the object in question. The latter desideratum is in contrast with previous work on library specifications in (R)C11 [Batty et al. 2013; Raad et al. 2019] and adaptations of linearizability to weak memory concurrency (see, e.g., [Dongol et al. 2018]), which employ multiple partial orders in their correctness criteria. We believe that an operational approach, based on *one* total timeline, may be more intuitive for users (also considering informal arguments), and will allow easier adoption of standard techniques and tools that were applied before to verify refinement between transition systems (see e.g., our initial experiment with the FDR refinement checker in §8.1).

While having an operational correctness criterion may seem contradicting to the fact that standard weak memory formulations, e.g., (R)C11, are *declarative* (a.k.a. axiomatic memory models), unlike previous work (e.g., [Batty et al. 2013]), we consider this to be only a superficial matter. Indeed, declarative specifications of models that forbid "reads from the future" (i.e., impose acyclicity of the union of the 'program order' and the 'read-from' relations; called an 'in-order' semantics in [Cho et al. 2021]) can be equally characterized as operational memory models: the states consist of the current execution trace ordered by multiple partial order relations as needed, and transitions are only between *consistent* states as specified by the consistency condition of the declarative model (see, e.g., the RAG transition system for the release/acquire fragment of C11 in [Lahav and Margalit 2019]). More concise (and possibly more intuitive) formulations of such semantics may use timestamps and views, and the equivalence to the "operationalized" declarative model is witnessed by a standard forward simulation relation (see, e.g., [Kaiser et al. 2017; Kang et al. 2017]).

Now, operational semantics (in particular, operationalized versions of declarative models), naturally give rise to a set of *library histories*, and allow the application of standard linearizability. For that matter, one considers a "most general client" program that invokes the library methods in the most general way expected by the library. Then, the set of library histories consists of all sequences of invocations (with the values of the arguments) and responses (with the returned values) of the library methods by different threads that are obtained in operational traces of that client. Under SC, inclusion of the sets of histories between two libraries may serve as a library abstraction condition (i.e., if every history of a library L is also a history of $L^{\#}$, then L refines $L^{\#}$).

Aiming to apply a similar condition, following previous work on abstraction under TSO [Burckhardt et al. 2012] and under non-volatile memory [Khyzha and Lahav 2022], we observe that under weak memory semantics, inclusion of sets of histories is generally *unsound* as a library correctness condition.[1] Next, we provide a simple (contrived) example of this issue.

Example 2.1. Consider a library L with two methods `foo` and `bar` and the following specification (here and henceforth we assume that all variables are initialized to 0):

```
foo() :  store(x, 1, rel); return();
```

```
bar() :  a := load(x, acq); return(a);
```

Suppose that the library enforces a policy on its clients: (1) `foo` and `bar` must be called *exactly once* in *different* threads; and (2) `bar` must be called *after* `foo` in the execution order. For example, the following program adheres to this call policy:

```
foo();               ||  b := load(y, rlx);
store(y, 1, rlx);    ||  if b = 1  then
                     ||    c := bar();
```

The accesses to y ensure that in the generated traces the call to `bar` indeed appears only after `foo` returns. Nevertheless, speaking in C11 terminology, since these accesses are relaxed (as annotated by `rlx` mode), they do *not* induce a *happens-before* order between the calls, and thus, following the specification, the read from x inside `bar` may return 0 (the initial value) or 1. Accordingly, a "naive" library developer may attempt to efficiently implement L as follows (where $\oplus$ denotes a non-deterministic choice):

```
foo() :  return();
```

```
bar() :  return(0) ⊕ return(1);
```

Is this implementation correct? In histories of the most general client of L that respects the library's calling policy (i.e., in histories generated by executing the above program) one cannot observe a difference between the implementation and the specification. But, nevertheless, contextual refinement does *not* hold (so the implementation cannot be considered correct) for two different reasons illustrated by the following client programs:

```
store(z, 1, rlx);    ||  b := load(y, rlx);  // 1
foo();               ||  if b = 1  then
store(y, 1, rlx);    ||    c := bar();  // 1
                     ||    d := load(z, rlx);  // 0
```

```
foo();               ||  b := load(y, acq);  // 1
store(y, 1, rel);    ||  if b = 1  then
                     ||    c := bar();  // 0
```

The annotated behaviors in both examples are allowed when the suggested implementation of `foo` and `bar` is used, but disallowed when the specification is used. Indeed, in the example on the left, since the specification ensures release/acquire synchronization when `bar` returns 1 (via the `rel` and `acq` annotations in the accesses to x), the write to z happens-before the read from z, and in this case the model disallows the read from z to observe the overwritten initial value. In turn, in the example on the right, since the *client* uses release/acquire accesses, the write in `foo` happens-before the read in `bar`, which similarly ensures that the read from x cannot observe the initial value.

To address this challenge, we need to make library histories more expressive, so we can avoid the (SC) pitfall that identifies the execution order and the synchronization order. Concretely, in the

[1]Interestingly, it follows from our result that for a language that employs *only* RC11-style relaxed accesses, inclusion of sets of histories is a sound condition (like it is for SC). Such a language is, however, too weak to be considered useful.

example above, we need to expose the facts that: (1) `bar` returning 1 *entails* a happens-before relation from *the call of* `foo` to *the return of* `bar`; and (2) `bar` returning 0 *forbids* a happens-before relation from *the return of* `foo` to *the call of* `bar`. What does that mean in operational terms without talking about the happens-before partial order between call and return actions? Our solution consists of the following:

- We introduce a novel operational semantics of the memory model and show that it is equivalent to the original declarative model. The operational semantics makes *knowledge propagation* between threads explicit. Roughly speaking, an access that is executed by one thread is first unknown to the other threads, and later non-deterministically propagates to each other threads. Access modes impose certain constraints on the propagation order and the ability to read from (locally) unknown events. In the simplest case—for programs with only release/acquire accesses—the propagation order has to follow the program order and reads can only read from known writes.
- We include propagation of method invocations and responses in the memory trace and, in turn, in library histories. These steps have no effect on the outcomes of the operational semantics (i.e., what values can be read and when), but they serve as crucial "markers" in library history, making histories expressive enough for validating contextual refinement given the inclusion of the sets of histories. In other words, we keep the inclusion of the sets of library histories as the correctness criterion, and recover its soundness by including appropriate markers (call and return propagation among different threads) in the histories.

In particular, revisiting the example above, the proposed implementation will have a history in which the propagation of the call marker of `foo` is after the return of `bar` but `bar` returns 1; as well as a history in which the propagation of the return marker of `foo` is before the call of `bar` but `bar` returns 0. Both histories are impossible for the specification, so library correctness does not hold.

2.2 Specification under Relaxed Memory Concurrency

A second challenge that we address is related to specification of libraries that expose weak behaviors. A straightforward approach to specifying concurrent data structures is to take a simple sequential implementation of the data structure, and wrap every operation inside a per-object lock. In fact, correctness according to the classical linearizability criterion [Herlihy and Wing 1990] is equivalent to refinement with respect to such lock-based specifications [Bouajjani et al. 2015; Filipović et al. 2010]. Since a lock entails a total order on all operations, the resulting specification is typically easy to understand and enables reasoning similar to reasoning about sequential programs.

This approach, however, has major shortcomings under relaxed memory concurrency, as it identifies the execution order with the synchronization order, which is only justified assuming SC. For instance, consider the following program that uses a concurrent queue (inspired by [Mével and Jourdan 2021]):

x := 1;	a := q.dequeue; // 1
q.enqueue(1);	b := x; // 0

Assuming a lock-based specification of the queue, the client may easily conclude that the annotated behavior ($a = 1$ and $b = 0$, where 0 is the initial value of x and the queue is initially empty) is impossible. Indeed, the fact that the dequeue operation returned the enqueued value entails that the lock of the dequeue operation must have been acquired after the lock of the enqueue operation was released. In turn, since locks necessarily provide release/acquire synchronization, the write to x happens-before the read from x, and thus the read cannot observe the overwritten initial value. Now, while the implementation of a queue may provide such synchronization guarantees for its client, it is also possible that the queue is implemented using efficient C11 relaxed accesses, and thus comes with weaker guarantees that do not allow the client to rely on library-induced synchronization (so the accesses to x above are racy and $a = 1$ but $b = 0$ is possible). In that case the lock-based specification would be too strong.

We note that such weak guarantees do not imply that the queue is not "linearizable": there may still exist a total order on all queue operations that agrees with the execution order (which can be any total order extending the union of the program order and the reads-from relation), and the queue exhibits FIFO behavior w.r.t. that order. A weak queue may be useful transferring elements of base types, whereas, in order to be used for transferring ownership via pointers, the client is responsible to appropriately place fences before and after invoking the weak queue methods.

Remark 2. While release/acquire synchronization may seem natural between enqueue and dequeue of the same element, the lock-based specification implies synchronization also in other cases, which may or may not be supplied (the queue studied in [Mével and Jourdan 2021] provides a case in point). For example, a client relying on a lock-based specification can deduce that the following annotated outcomes are all disallowed (where ⊥ denotes an empty queue):

x := 1; q.enqueue(1);	q.enqueue(2); a := x; // *0*	b := q.dequeue; // *1* c := q.dequeue; // *2*

x := 1; a := q.dequeue; // ⊥	q.enqueue(1); b := x; // *0*

Accordingly, the challenge lies in identifying specification constructs that can be used instead of standard locks and be sufficiently flexible to account for different synchronization guarantees for the client. Ideally, like locks, they should allow straightforward reasoning about the library behaviors. We observe that existing concurrency models, C11 in particular, lack such constructs.

To address this challenge, we propose to extend the language with specialized instructions. Concretely, we identify that there is a range of possible guarantees that lie between C11's release/acquire accesses (which provide synchronization) and relaxed accesses (which do not), and introduce a novel type of memory accesses that lies on this spectrum. Roughly speaking, these memory accesses, which we call *partial release/acquire*, ensure synchronization only for certain variables (see §4 for the formal definition).[2] C11's release/acquire accesses can be seen as partial release/acquire accesses that ensure synchronization for *all* variables, whereas C11's relaxed accesses can be seen as partial release/acquire accesses that do *not* ensure synchronization for *any* variable.

Partial release/acquire accesses can be used to construct locks that provide synchronization *only* for library variables, and these special locks can be used to specify libraries with weak synchronization guarantees. In particular, in the queue example above, locks implemented by *partial* release/acquire accesses would behave just like standard locks from library perspective, but will not allow the clients to rely on their induced synchronization. More generally, these accesses allow a library-internal release/acquire synchronization, which is needed for correct behaviors of the library, but is considered invisible from the point of view of the client.

3 CONCURRENT PROGRAMS: SYNTAX AND MEMORY-INDEPENDENT SEMANTICS

In this section we begin to present the formal preliminaries for our results. As standard in memory models, it is convenient to break the semantics into: a *program* semantics (a.k.a. thread subsystem) and a *memory* semantics. Next, we focus on the program part, presenting syntax (§3.1), semantics (§3.2), and the synchronization with a (parametric) memory system (§3.3).

[2]We generally require libraries and clients to operate on different "address spaces" and never access the same shared variable. Lifting this simplifying assumption (e.g., as was done for SC in [Gotsman and Yang 2013]) is beyond our current scope.

3.1 Program Syntax

We employ a simple programming language, which supports the distinction between clients and libraries variable spaces.[3] We use the following domains (and metavariables ranging over them):

(variable) spaces	$X, Y \in \mathsf{Space} \triangleq \{\mathsf{X}, \mathsf{Y}, \dots\}$	*variables*	$x, y \in \mathsf{Loc} \triangleq \{\mathsf{x}, \mathsf{y}, \dots\}$
thread identifiers	$\tau, \pi \in \mathsf{Tid} = \{\mathsf{T}_1, \mathsf{T}_2, \dots, \mathsf{T}_\mathsf{N}\}$	*registers*	$r \in \mathsf{Reg} = \{\mathsf{a}, \mathsf{b}, \dots\}$
read modes	$o_\mathsf{R} \in \mathsf{Mod}_\mathsf{R} \triangleq \{\mathsf{na}, \mathtt{rlx}, \mathtt{pacq}, \mathtt{acq}\}$	*values*	$v \in \mathsf{Val} \triangleq \{0, 1, 2, \dots\}$
write modes	$o_\mathsf{W} \in \mathsf{Mod}_\mathsf{W} \triangleq \{\mathsf{na}, \mathtt{rlx}, \mathtt{prel}, \mathtt{rel}\}$	*method names*	$f \in \mathsf{F}$ $\quad \mathtt{main} \notin \mathsf{F}$

In particular, as we will see below, every memory access is to a particular variable in a particular variable space (like address spaces in operating systems); registers are thread-local variables; and the name `main` is reserved for non-library operations. Memory instructions have access modes (a.k.a. memory orderings), which determine their "strength" (na for non-atomics, `rlx` for relaxed, `prel`/`pacq` for partial release/acquire, and `rel`/`acq` for release/acquire). We also assume a relation $\sqsubset$ that orders modes according to their strength (following the left-to-right enumeration of the sets Mod_R and Mod_W above).

The language provides the following constructs, inspired by C11 atomics:

- Expressions are constructed with arithmetic and boolean operations over registers and values. We use e to range over expressions, and leave the exact expression grammar parametric.
- Thread local instructions (which do not interact with the shared-memory system):
  - Assignments of the form $r := e$, used for storing an expression e in a register r.
  - Conditionals of the form `if` e `goto` $n_1 \mid \dots \mid n_m$ (where $n_1, \dots, n_m \in \mathbb{N}$), used to non-deterministically jump to some program counter among $\{n_1, \dots, n_m\}$ when e evaluates to non-zero or, otherwise, skipping.
  - Additional local instructions (standard loop constructs, unconditional non-deterministic choice, etc.). Below, we will use such instructions in our examples, and their semantics should be clear.
- Shared-memory single accesses:
  - Write (a.k.a. store) instructions to memory of the form $\mathtt{store}(X, x, e, o_\mathsf{W})$ for storing into a shared variable x in space X the value that e evaluates to with access mode o_W.
  - Read (a.k.a. load) instructions from memory of the form $r := \mathtt{load}(X, x, o_\mathsf{R})$ for loading the value from a shared variable x in space X into a register r with access mode o_R.
- Read-modify-write (RMW) instructions:
  - Fetch-and-add (FAA) instructions of the form $r := \mathtt{FADD}(X, x, e, o_\mathsf{R}, o_\mathsf{W})$ for atomically incrementing a variable x in space X by the value of e with read mode o_R and write mode o_W.
  - Compare-and-swap (CAS) instructions of the form $r := \mathtt{CAS}(X, x, e_\mathsf{R}, e_\mathsf{W}, o_\mathsf{R}, o_\mathsf{W}, o_\mathsf{R}^{\mathtt{fail}})$. This instruction atomically loads the value from x in space X into r, compares it to the value e_R, and overwrites it by the value of e_W in case if the loaded value coincides with the value e_R. The load part will have mode o_R if comparison succeeds and $o_\mathsf{R}^{\mathtt{fail}}$ otherwise; and the store part (if it happens) has mode o_W.
- Library interaction: $\mathtt{call}(f)$ for calling a method f and `return` for returning to the caller. For simplicity, we do not provide any argument passing mechanism and we will use the full register store for that matter. (If needed, each component may store the values it needs in the memory, and reload them later on.)

To construct programs we introduce three syntactic categories, each of which builds on the previous one:

[3] The partition to variable spaces is only needed to support the "partial release/acquire" access mode in library specifications, and can be completely ignored for specifications without such accesses.

- *Instruction sequences* represent the (sequential) implementation of each method (including `main`). Formally, an instruction sequence I is a function from a non-empty finite domain of the form $\{0, \dots, n\}$ (representing the possible program counters) to the set of instructions. We say that an instruction sequence is *flat* if it does not include `call`(_) instructions.
- *Sequential programs* consist of a "main" method accompanied with implementations of every method $f \in \mathsf{F}$. Formally, a sequential program *sPr* is a function assigning an instruction sequence to every $f \in \{\texttt{main}\} \cup \mathsf{F}$. To avoid modeling a call stack and simplify the framework, we require that $sPr(f)$ is a flat instruction sequence for every $f \in \mathsf{F}$.
- *Concurrent programs*, which we often call *programs*, are top-level parallel compositions of sequential programs, all accompanied by the same method implementations. Formally, a (concurrent) program *Pr* is a mapping assigning a sequential program to every $\tau \in \mathsf{Tid}$, with $Pr(\tau)(f) = Pr(\pi)(f)$ for every $\tau, \pi \in \mathsf{Tid}$ and $f \in \mathsf{F}$.

In our examples, we often write instruction sequences as sequences of instructions delimited by ";", and concurrent programs using '‖' between the main method of each thread. We also refer to the program threads as $\mathsf{T}_1, \mathsf{T}_2, \dots$ following their left-to-right order in the program listing.

3.2 Program Semantics

We give semantics to the syntactic objects above using labeled transition systems.

Definition 3.1. A *labeled transition system* (LTS) is a tuple $A = \langle \Sigma, Q, q_0, T \rangle$, where Σ is a set of *transition labels*, Q is a set of *states*, $q_0 \in Q$ is the *initial state*, and $T \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. We denote by $A.\Sigma$, $A.\mathsf{Q}$, $A.\mathsf{q_0}$, and $A.\mathsf{T}$ the components of an LTS A. We write $q \xrightarrow{\sigma} q'$ to denote a transition $\langle q, \sigma, q' \rangle$, $\xrightarrow{\sigma}_A$ for the relation $\{\langle q, q' \rangle \mid q \xrightarrow{\sigma} q' \in A.\mathsf{T}\}$, and $\rightarrow_A$ for $\bigcup_{\sigma \in \Sigma} \xrightarrow{\sigma}_A$. For a sequence $t \in A.\Sigma^*$, we write $\xrightarrow{t}_A$ for the composition $\xrightarrow{t(1)}_A ; \dots ; \xrightarrow{t(|t|)}_A$. A sequence $t \in A.\Sigma^*$ such that $A.\mathsf{q_0} \xrightarrow{t}_A q$ for some $q \in A.\mathsf{Q}$ is called a *trace* of A. We denote by $\mathtt{traces}(A)$ the set of all traces of A. A state $q \in A.\mathsf{Q}$ is called *reachable* in A if $A.\mathsf{q_0} \xrightarrow{t}_A q$ for some $t \in \mathtt{traces}(A)$. For a trace t and a set $\Theta \subseteq \Sigma$ of transition labels, we write $t|_\Theta$ for the longest subsequence of t over Θ.

Next, we define the LTSs induced by instruction sequences, sequential programs, and concurrent programs. We often identify the syntactic objects with the LTS they induce (e.g., when writing expressions like $sPr.\mathsf{Q}$ for a sequential program *sPr*). The transition labels of these LTSs feature *action labels*, which represent the interactions that a program may have with the memory.

Definition 3.2. An *action label* l takes one of the following forms: a read $\mathtt{R}(X, x, v_\mathtt{R}, o_\mathtt{R})$, a write $\mathtt{W}(X, x, v_\mathtt{W}, o_\mathtt{W})$, a read-modify-write $\mathtt{RMW}(X, x, v_\mathtt{R}, v_\mathtt{W}, o_\mathtt{R}, o_\mathtt{W})$, a call $\mathtt{CALL}(f, \phi)$, and a return $\mathtt{RET}(\phi)$, where $X \in \mathsf{Space}$, $x \in \mathsf{Loc}$, $v_\mathtt{R}, v_\mathtt{W}, \in \mathsf{Val}$, $o_\mathtt{R} \in \mathsf{Mod_R}$, $o_\mathtt{W} \in \mathsf{Mod_W}$, $f \in \mathsf{F}$, and $\phi : \mathsf{Reg} \rightarrow \mathsf{Val}$. We denote by Lab the set of all action labels. The functions $\mathtt{typ}$, $\mathtt{sp}$, $\mathtt{loc}$, $\mathtt{val_R}$, $\mathtt{val_W}$, $\mathtt{mod_R}$, $\mathtt{mod_W}$, $\mathtt{callee}$, and $\mathtt{store}$ respectively retrieve (when applicable) the type ($\mathtt{R}/\mathtt{W}/\dots$), space ($X$), variable ($x$), read value ($v_\mathtt{R}$), written value ($v_\mathtt{W}$), read mode ($o_\mathtt{R}$), write mode ($o_\mathtt{W}$), callee method name ($f$), and store ($\phi$) of an action label.

Next, we define the LTS induced by an instruction sequence.

Definition 3.3. An *instruction sequence state* is a pair $\langle pc, \phi \rangle$, where $pc \in \mathbb{N}$, called *program counter*, stores the current instruction pointer inside the sequence, and $\phi : \mathsf{Reg} \rightarrow \mathsf{Val}$, called *local store*, records the values of the registers. Local stores are extended to apply on expressions in the standard way. The *LTS induced by an instruction sequence I* is an LTS over instruction sequence states with: $\mathsf{Lab}_\epsilon \triangleq \mathsf{Lab} \cup \{\epsilon\}$ as the set of transition labels (that is, the set of all action labels extended with ϵ for silent transitions); $\langle 0, \phi_\mathsf{Init} \rangle$ where $\phi_\mathsf{Init} \triangleq \lambda r.\, 0$ as the initial state; and the transitions as given in Fig. 1 (additional thread local instructions can be standardly added).

$$\frac{I(pc) = r := e \quad \phi' = \phi[r \mapsto \phi(e)]}{\langle pc, \phi \rangle \xrightarrow{\epsilon}_I \langle pc+1, \phi' \rangle}
\qquad
\frac{I(pc) = \texttt{if}\ e\ \texttt{goto}\ n_1 \mid \ldots \mid n_m \quad \phi(e) \neq 0 \implies pc' \in \{n_1, \ldots, n_m\} \quad \phi(e) = 0 \implies pc' = pc+1}{\langle pc, \phi \rangle \xrightarrow{\epsilon}_I \langle pc', \phi \rangle}
\qquad
\frac{I(pc) = r := \texttt{load}(X, x, o_\mathsf{R}) \quad l = \mathsf{R}(X, x, v, o_\mathsf{R}) \quad \phi' = \phi[r \mapsto v]}{\langle pc, \phi \rangle \xrightarrow{l}_I \langle pc+1, \phi' \rangle}$$

$$\frac{I(pc) = \texttt{store}(X, x, e, o_\mathsf{W}) \quad l = \mathsf{W}(X, x, \phi(e), o_\mathsf{W})}{\langle pc, \phi \rangle \xrightarrow{l}_I \langle pc+1, \phi \rangle}
\qquad
\frac{I(pc) = r := \texttt{FADD}(X, x, e, o_\mathsf{R}, o_\mathsf{W}) \quad l = \mathsf{RMW}(X, x, v, v + \phi(e), o_\mathsf{R}, o_\mathsf{W}) \quad \phi' = \phi[r \mapsto v]}{\langle pc, \phi \rangle \xrightarrow{l}_I \langle pc+1, \phi' \rangle}$$

$$\frac{I(pc) = r := \texttt{CAS}(X, x, e_\mathsf{R}, e_\mathsf{W}, o_\mathsf{R}, o_\mathsf{W}, o_\mathsf{R}^{\texttt{fail}}) \quad v = \phi(e_\mathsf{R}) \implies l = \mathsf{RMW}(X, x, v, \phi(e_\mathsf{W}), o_\mathsf{R}, o_\mathsf{W}) \quad v \neq \phi(e_\mathsf{R}) \implies l = \mathsf{R}(X, x, v, o_\mathsf{R}^{\texttt{fail}}) \quad \phi' = \phi[r \mapsto v]}{\langle pc, \phi \rangle \xrightarrow{l}_I \langle pc+1, \phi' \rangle}$$

Fig. 1. Transitions of LTS induced by an instruction sequence

Recall that program semantics is separate from memory semantics, which is why the read and RMW transitions in Fig. 1 can observe any value. It is only important that each transition that interacts with the memory announces itself in the transition label. The `call(_)` and `return` instructions are not handled at the level of instruction sequences, but receive special semantics at the level of sequential programs, as defined next.

Definition 3.4. A *sequential program state* is a tuple $q = \langle pc, \phi, pc_\mathsf{s}, f \rangle$, where: $\langle pc, \phi \rangle$ is an instruction sequence state storing the state of the sequence currently running; $pc_\mathsf{s} \in \mathbb{N} \cup \{\bot\}$, called *the stored program counter*, is used to remember the program position to jump to when the current instruction sequence returns ($pc_\mathsf{s} = \bot$ means that the main method is currently running); and $f \in \mathsf{F} \cup \{\texttt{main}\}$, called *the active method*, tracks the method that is currently running. We denote by $q.\mathsf{pc}$, $q.\phi$, $q.\mathsf{pc_s}$, and $q.\mathsf{f}$ the components of a sequential program state q.

Definition 3.5. The *LTS induced by a sequential program sPr* is given by:

- The set of transition labels is $\mathsf{Lab}_\epsilon \times (\mathsf{F} \cup \{\texttt{main}\})$. The functions `lab` and `method` respectively retrieve the action label (or ϵ) and method name of a transition label. All functions on action labels (`typ`, `sp`, `loc`, ...) are lifted to sequential program transition labels in the obvious way.
- The states are sequential program states, as defined in Def. 3.4.
- The initial state is $\langle 0, \phi_{\mathsf{Init}}, \bot, \texttt{main} \rangle$.
- The transitions are given by:

$$\frac{f \in \{\texttt{main}\} \cup \mathsf{F} \qquad \langle pc, \phi \rangle \xrightarrow{l_\epsilon}_{sPr(f)} \langle pc', \phi' \rangle}{\langle pc, \phi, pc_\mathsf{s}, f \rangle \xrightarrow{l_\epsilon, f}_{sPr} \langle pc', \phi', pc_\mathsf{s}, f \rangle}$$

$$\frac{sPr(\texttt{main})(pc) = \texttt{call}(f) \quad l = \mathsf{CALL}(f, \phi)}{\langle pc, \phi, \bot, \texttt{main} \rangle \xrightarrow{l, \texttt{main}}_{sPr} \langle 0, \phi, pc+1, f \rangle}
\qquad
\frac{sPr(f)(pc) = \texttt{return} \quad l = \mathsf{RET}(\phi)}{\langle pc, \phi, pc_\mathsf{s}, f \rangle \xrightarrow{l, f}_{sPr} \langle pc_\mathsf{s}, \phi, \bot, \texttt{main} \rangle}$$

The first transition, which applies for any method (`main` or other), lifts the instruction-sequence transition to the level of sequential programs. The second transition passes control from the main method to some other method, jumping the program counter to the first instruction and storing the return point ($pc + 1$). Finally, the third transition passes control back using the stored return point. (We do not need to record a call stack since we assume that $sPr(f)$ is flat for every $f \in \mathsf{F}$.)

Finally, the LTS induced by a concurrent program interleaves the thread transitions.

Definition 3.6. A *(concurrent) program state* $\overline{p}$ is a mapping assigning a sequential program state to every $\tau \in \mathsf{Tid}$. The *LTS induced by a program Pr* is an LTS over program states, with: $\mathsf{ProgLab} \triangleq \mathsf{Tid} \times \mathsf{Lab}_\epsilon \times (\mathsf{F} \cup \{\mathtt{main}\})$ as the set of transition labels; $\overline{p}_{\mathsf{Init}} \triangleq \lambda\tau.\, \langle 0, \phi_{\mathsf{Init}}, \bot, \mathtt{main}\rangle$ as the initial state; and the following transitions:

$$\frac{\overline{p}(\tau) \xrightarrow{l_\epsilon, f}_{Pr(\tau)} q'}{\overline{p} \xrightarrow{\tau, l_\epsilon, f}_{Pr} \overline{p}[\tau \mapsto q']}$$

Below, for a program transition label $\alpha \in \mathsf{ProgLab}$, the functions `tid`, `lab`, and `method` respectively retrieve the thread identifier (τ), the action label (or ϵ) (l_ϵ), and the method name (f) of α. Functions on action labels (`typ`, `sp`, `loc`, ...) are lifted to program transition labels in the obvious way.

3.3 Synchronizing Programs and Memories

To give semantics to programs running under a particular memory model, we synchronize the transitions of a program *Pr* with a memory system. For now, we leave the memory system parametric, and assume it is represented by some LTS $\mathcal{M}$ whose set of transition labels consists of non-silent program transition labels (elements of $\mathsf{Tid} \times \mathsf{Lab} \times (\mathsf{F} \cup \{\mathtt{main}\})$) as well as a (disjoint) set $\mathcal{M}.\Theta$ of internal memory actions (which we use later for non-deterministic propagation of knowledge between threads).

Definition 3.7. The composition of a program *Pr* and a memory system $\mathcal{M}$, denoted by $Pr \bowtie \mathcal{M}$, is the LTS whose transition labels are the elements of $\mathsf{ProgLab} \cup \mathcal{M}.\Theta$; states are pairs $\langle \overline{p}, M\rangle \in Pr.\mathsf{Q} \times \mathcal{M}.\mathsf{Q}$; initial state is $\langle \overline{p}_{\mathsf{Init}}, \mathcal{M}.\mathsf{q}_0\rangle$; and transitions are given by:

$$\frac{\begin{array}{c}\alpha \in \mathsf{Tid} \times \mathsf{Lab} \times (\mathsf{F} \cup \{\mathtt{main}\}) \\ \overline{p} \xrightarrow{\alpha}_{Pr} \overline{p}' \qquad M \xrightarrow{\alpha}_{\mathcal{M}} M'\end{array}}{\langle \overline{p}, M\rangle \xrightarrow{\alpha}_{Pr \bowtie \mathcal{M}} \langle \overline{p}', M'\rangle} \qquad \frac{\begin{array}{c}\alpha \in \mathsf{Tid} \times \{\epsilon\} \times (\mathsf{F} \cup \{\mathtt{main}\}) \\ \overline{p} \xrightarrow{\alpha}_{Pr} \overline{p}'\end{array}}{\langle \overline{p}, M\rangle \xrightarrow{\alpha}_{Pr \bowtie \mathcal{M}} \langle \overline{p}', M\rangle} \qquad \frac{\begin{array}{c}\alpha \in \mathcal{M}.\Theta \\ M \xrightarrow{\alpha}_{\mathcal{M}} M'\end{array}}{\langle \overline{p}, M\rangle \xrightarrow{\alpha}_{Pr \bowtie \mathcal{M}} \langle \overline{p}, M'\rangle}$$

The above transitions are "synchronized transitions" of *Pr* and $\mathcal{M}$, using the labels to decide what to synchronize on. Both the program and the memory take the same step for transition labels that are common to both LTSs, only the program steps for transition labels that are program internal (i.e., with $\mathtt{lab}(\alpha) = \epsilon$) and only the memory steps for transition labels that are memory internal.

Example 3.8. The most well-known memory system is the one of sequential consistency, denoted here by SC. This memory system simply tracks the most recent value written to each variable, and has no internal transitions ($\mathsf{SC}.\Theta = \emptyset$). Formally, it is defined by $\mathsf{SC}.\mathsf{Q} \triangleq (\mathsf{Space} \times \mathsf{Loc}) \to \mathsf{Val}$, $\mathsf{SC}.\mathsf{q}_0 \triangleq \lambda X, x.\, 0$, and $\to_{\mathsf{SC}}$ is given by:

$$\frac{\begin{array}{c} l = \mathtt{R}(X, x, v_{\mathtt{R}}, \_) \\ m(X, x) = v_{\mathtt{R}}\end{array}}{m \xrightarrow{\tau, l, f}_{\mathsf{SC}} m} \qquad \frac{\begin{array}{c} l = \mathtt{W}(X, x, v_{\mathtt{W}}, \_) \\ m' = m[\langle X, x\rangle \mapsto v_{\mathtt{W}}]\end{array}}{m \xrightarrow{\tau, l, f}_{\mathsf{SC}} m'} \qquad \frac{\begin{array}{c} l = \mathtt{RMW}(X, x, v_{\mathtt{R}}, v_{\mathtt{W}}, \_, \_) \\ m(X, x) = v_{\mathtt{R}} \\ m' = m[\langle X, x\rangle \mapsto v_{\mathtt{W}}]\end{array}}{m \xrightarrow{\tau, l, f}_{\mathsf{SC}} m'} \qquad \frac{\mathtt{typ}(l) \in \{\mathtt{CALL}, \mathtt{RET}\}}{m \xrightarrow{\tau, l, f}_{\mathsf{SC}} m}$$

4 THE dRC11 MEMORY MODEL

In this section we introduce the weak memory model that we assume in this paper. This model, which we call dRC11, is a declarative (a.k.a. axiomatic) model forming an extension of (a fragment of) the RC11 model [Lahav et al. 2017] with specialized semantics for the novel `prel`/`pacq` accesses.

Notation 4.1 (Relational notations). *Given a (binary) relation R, dom(R) and codom(R) denote its domain and codomain, and $R^?$, R^+, and R^* denote its reflexive, transitive, and reflexive-transitive closures. The inverse of a relation R is denoted by R^{-1}, and the (left) composition of two* *relations* R_1 *and* R_2 *is denoted by* $R_1\,;R_2$. *We denote by* $[A]$ *the identity relation on a set* A. *In particular,* $[A]\,;R\,;[B] = R \cap (A \times B)$. *When* A *is finite, we write* $[a_1, \dots, a_n]$ *instead of* $[\{a_1, \dots, a_n\}]$.

We start by defining *execution graphs*. Their nodes, called *events*, represent memory accesses, and their directed edges are of different kinds: *program order* represents the order imposed by the program; *reads-from* mapping maps each read event to the write event it obtains its value from; and *modification order* (a.k.a. *coherence order*) provides a total order on the writes to every variable. The precise definitions are given next.

Definition 4.2. An *event* e is a tuple $\langle \tau, s, l, f\rangle$, where $\tau \in \mathsf{Tid} \uplus \{\bot\}$, called the event's *thread identifier* ($\bot$ is used for initialization events); $s \in \mathbb{N}$, called the event's *serial identifier*; $l \in \mathsf{Lab}$, called the event's *label* (as defined in Def. 3.2), and $f \in \mathsf{F} \cup \{\mathtt{main}\}$, called the event's *method*. The functions $\mathtt{tid}$, $\mathtt{sn}$, $\mathtt{lab}$, and $\mathtt{method}$ return the thread identifier (τ), identifier (s), action label (l), and method of an event (f). All functions on action labels ($\mathtt{typ}$, $\mathtt{sp}$, $\mathtt{loc}$, ...) are lifted to events in the obvious way. We denote by E the set of all events, and define the following subsets:

$$\mathsf{R} \triangleq \{e \in \mathsf{E} \mid \mathtt{typ}(e) \in \{\mathsf{R}, \mathsf{RMW}\}\} \qquad \mathsf{W} \triangleq \{e \in \mathsf{E} \mid \mathtt{typ}(e) \in \{\mathtt{W}, \mathtt{RMW}\}\} \qquad \mathsf{RMW} \triangleq \mathsf{R} \cap \mathsf{W}$$
$$\mathsf{CALL} \triangleq \{e \in \mathsf{E} \mid \mathtt{typ}(e) = \mathtt{CALL}\} \qquad \mathsf{RET} \triangleq \{e \in \mathsf{E} \mid \mathtt{typ}(e) = \mathtt{RET}\} \qquad \mathsf{CR} \triangleq \mathsf{CALL} \cup \mathsf{RET}$$

We employ subscripts and superscripts to restrict sets of events to certain properties, e.g., $\mathsf{W}_X = \{w \in \mathsf{W} \mid \mathtt{sp}(w) = X\}$, $\mathsf{R}_{X,x}^{\sqsupseteq \mathsf{pacq}} = \{r \in \mathsf{R} \mid \mathtt{sp}(r) = X \land \mathtt{loc}(r) = x \land \mathtt{mod}_{\mathtt{W}}(r) \sqsupseteq \mathsf{pacq}\}$, $\mathsf{CALL}_F = \{e \in \mathsf{CALL} \mid \mathtt{callee}(e) \in F\}$, $\mathsf{RET}_F = \{e \in \mathsf{RET} \mid \mathtt{method}(e) \in F\}$, $\mathsf{CR}_F = \mathsf{CALL}_F \cup \mathsf{RET}_F$, and $E^\tau = \{e \in E \mid \mathtt{tid}(e) = \tau\}$ for any $E \subseteq \mathsf{E}$. The set Init of *initialization events* is given by:

$$\mathsf{Init} \triangleq \{\langle \bot, 0, \mathtt{W}(X, x, 0, \mathtt{rlx}), \mathtt{main}\rangle \mid X \in \mathsf{Space}, x \in \mathsf{Loc}\}.$$

Definition 4.3. An *execution graph* G is a tuple $\langle E, po, rf, mo\rangle$, where:

- E is a finite set of events, such that $\mathsf{Init} \subseteq E$ and $\mathtt{tid}(e) \neq \bot$ for every $e \in E \setminus \mathsf{Init}$.
- po is a *program order for* E, that is: $po = (\biguplus_{\tau \in \mathsf{Tid}} po_\tau) \uplus (\mathsf{Init} \times (E \setminus \mathsf{Init}))$, for some relations po_τ, such that each po_τ is a strict total order on E^τ.
- rf is a *reads-from relation for* E, that is a relation on E satisfying:
  - If $\langle w, r\rangle \in rf$, then $w \in \mathsf{W}$ and $r \in \mathsf{R}$.
  - If $\langle w, r\rangle \in rf$, then $\mathtt{sp}(w) = \mathtt{sp}(r)$, $\mathtt{loc}(w) = \mathtt{loc}(r)$, and $\mathtt{val}_{\mathtt{W}}(w) = \mathtt{val}_{\mathtt{R}}(r)$.
  - $w_1 = w_2$ whenever $\langle w_1, r\rangle, \langle w_2, r\rangle \in rf$ (each read reads from at most one write).
  - $E \cap \mathsf{R} \subseteq codom(rf)$ (each read reads from some write).
- mo is a *modification order for* E, that is: $mo = \biguplus_{X \in \mathsf{Space}, x \in \mathsf{Loc}} mo_{X,x}$, for some relations $mo_{X,x}$, such that each $mo_{X,x}$ is a strict *total* order on $E \cap \mathsf{W}_{X,x}$.

We denote the components of G by $G.\mathsf{E}$, $G.\mathtt{po}$, $G.\mathtt{rf}$, and $G.\mathtt{mo}$. For any set $E' \subseteq \mathsf{E}$, we write $G.E'$ for $G.\mathsf{E} \cap E'$ (e.g., $G.\mathsf{W} = G.\mathsf{E} \cap \mathsf{W}$).

To formally associate execution graphs with programs we use a memory system called FG (for "free graphs"), whose states are execution graphs. This system allows *all* possible program transitions, while recording them in its state, which is the current execution graph with (almost) arbitrary reads-from and modification order relations.

Definition 4.4. The memory system FG is the LTS whose transition labels are program transition labels (i.e., there are not any internal memory actions); states are execution graphs; initial state is G_{Init}, defined by $G_{\mathsf{Init}}.\mathsf{E} \triangleq \mathsf{Init}$ and $G_{\mathsf{Init}}.\mathsf{X} \triangleq \emptyset$ for every other component of G; and transitions are given in Fig. 2.

The transitions of FG are based on helper notations used to extend an execution graph G with a fresh event e at the end of the executing thread. For memory accesses, it requires to pick a write

$$\frac{\begin{array}{c}\textsc{write/read/rmw}\\ \mathsf{typ}(l) \in \{\mathsf{W},\mathsf{R},\mathsf{RMW}\} \qquad e = \mathsf{NextEvent}(G.\mathsf{E},\tau,l,f) \\ w \in G.\mathsf{W}_{\mathtt{sp}(e),\mathtt{loc}(e)} \qquad e \in \mathsf{R} \implies \mathtt{val}_{\mathsf{W}}(w) = \mathtt{val}_{\mathsf{R}}(e) \\ G' = \mathsf{Add}(G,e,w)\end{array}}{G \xrightarrow{\tau,l,f}_{\mathsf{FG}} G'} \qquad \frac{\begin{array}{c}\textsc{call/return}\\ \mathsf{typ}(l) \in \{\mathsf{CALL},\mathsf{RET}\} \\ e = \mathsf{NextEvent}(G.\mathsf{E},\tau,l,f) \\ G' = \mathsf{Add}(G,e)\end{array}}{G \xrightarrow{\tau,l,f}_{\mathsf{FG}} G'}$$

Fig. 2. Transitions of FG system for generating all candidate execution graphs

event w in G, called the *write-predecessor* of e, that is: (1) the $\mathtt{rf}$-source of e if e is a read; (2) the $\mathtt{mo}$-immediate predecessor of e if e is a write; and (3) both the $\mathtt{rf}$-source and the $\mathtt{mo}$-immediate predecessor of e if e is an RMW. For that matter, we employ the following notations:

Notation 4.5. *Given a set E of events, $\tau \in \mathsf{Tid}$, $l \in \mathsf{Lab}$, and $f \in \mathsf{F}$, $\mathsf{NextEvent}(E,\tau,l,f)$ denotes the event with thread identifier τ, label l, method f, and a minimal fresh serial identifier w.r.t. E, that is: $\mathsf{NextEvent}(E,\tau,l,f) \triangleq \langle \tau, s, l, f\rangle$, where $s = \min\{n \in \mathbb{N} \mid \langle \tau, n, l, f\rangle \notin E\}$.*

Notation 4.6. *For an execution graph G and events e and w, $\mathsf{Add}(G,e,w)$ denotes the tuple $\langle E', po', rf', mo'\rangle$, where:*

$$E' = G.\mathsf{E} \cup \{e\} \qquad\qquad po' \triangleq G.\mathsf{po} \cup ((G.\mathsf{E}^{\mathtt{tid}(e)} \cup \mathsf{Init}) \times \{e\})$$

$$rf' = \begin{cases} G.\mathtt{rf} \cup \{\langle w,e\rangle\} & e \in \mathsf{R} \\ G.\mathtt{rf} & \textit{otherwise}\end{cases} \qquad mo' = \begin{cases} G.\mathtt{mo} \cup \mathit{dom}(G.\mathtt{mo}^? \,;[w]) \times \{e\} & e \in \mathsf{W} \\ \quad \cup \{e\} \times \mathit{codom}([w]\,; G.\mathtt{mo}) & \\ G.\mathtt{mo} & \textit{otherwise}\end{cases}$$

Similarly, for an execution graph G and event e, $\mathsf{Add}(G,e)$ denotes the tuple $\langle E', po', G.\mathtt{rf}, G.\mathtt{mo}\rangle$, where E' and po' are defined as above.

In the sequel, it will be useful to note that all execution graphs generated by FG when synchronized with a given program satisfy the following well-formedness property:

Definition 4.7. An execution graph G is *well-formed* if the following hold for every $f \in \mathsf{F}$:

- $[\{e \in \mathsf{E} \mid \mathtt{method}(e) \neq f\}]\,; G.\mathsf{po}\,; [\{e \in \mathsf{E} \mid \mathtt{method}(e) = f\}] \subseteq G.\mathsf{po}^?\,; [\mathsf{CALL}_{\{f\}}]\,; G.\mathsf{po}$.
- $[\{e \in \mathsf{E} \mid \mathtt{method}(e) = f\}]\,; G.\mathsf{po}\,; [\{e \in \mathsf{E} \mid \mathtt{method}(e) \neq f\}] \subseteq G.\mathsf{po}^?\,; [\mathsf{RET}_{\{f\}}]\,; G.\mathsf{po}$.

Proposition 4.8. *If $\langle \overline{p}, G\rangle$ is reachable in $Pr \bowtie \mathsf{FG}$ for some program Pr, then G is well-formed.*

Now, to filter for *consistent* graphs among all candidate execution graphs generated by FG for a given program, we define several derived relations (some parametrized by $X \in \mathsf{Space}$):

$$G.\mathtt{fr} \triangleq (G.\mathtt{rf}^{-1}\,; G.\mathtt{mo}) \setminus [\mathsf{E}] \qquad \text{(from-read, a.k.a. read-before)}$$

$$G.\mathtt{sw}_{\mathrm{base}} \triangleq [\mathsf{W}^{\mathtt{rel}}]\,; G.\mathtt{rf}^+\,; [\mathsf{R}^{\mathtt{acq}}] \qquad \text{(global synchronization)}$$

$$G.\mathtt{sw}_X \triangleq [\mathsf{W}_X^{\sqsupseteq\mathtt{prel}}]\,; G.\mathtt{rf}^+\,; [\mathsf{R}_X^{\sqsupseteq\mathtt{pacq}}] \qquad \text{(per-space synchronization)}$$

$$G.\mathtt{hb}_X \triangleq (G.\mathsf{po} \cup G.\mathtt{sw}_{\mathrm{base}} \cup G.\mathtt{sw}_X)^+ \qquad \text{(per-space happens-before)}$$

The $\mathtt{fr}$ relation is standard in weak memory models [Alglave et al. 2014], relating every read (or RMW) to subsequent writes (or RMWs) as dictated by $\mathtt{rf}$ and $\mathtt{mo}$ (every write w that is $\mathtt{mo}$-after the $\mathtt{rf}$-source of a read r is $\mathtt{fr}$-after r). The $\mathtt{sw}_{\mathrm{base}}$ and $\mathtt{sw}_X$ relations formally capture synchronization: $\mathtt{sw}_{\mathrm{base}}$ is for "global" synchronization which is formed by reads-from edges between $\mathtt{rel}$ and $\mathtt{acq}$ accesses (just like in C11 [Lahav et al. 2017]), whereas $\mathtt{sw}_X$ is for "space-internal" synchronization that affects only accesses to space X and is formed by reads-from edges *in space* X between $\mathtt{prel}$

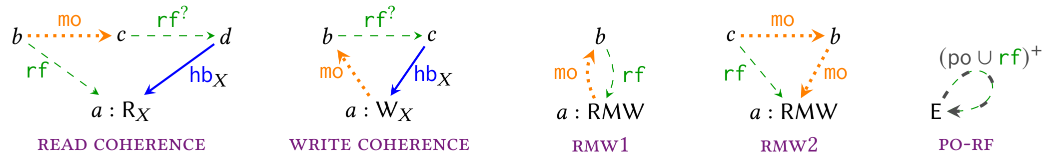


Fig. 3. Illustration of forbidden patterns in dRC11

and pacq. The use of $\mathtt{rf}^+$ (rather than just $\mathtt{rf}$) is for supporting *release sequences* as in C11, which ensures the synchronization between a release write w and an acquire read r also if there is chain of reads-from edges between them ($\langle w, u_1 \rangle, \langle u_1, u_2 \rangle, \dots, \langle u_{n-1}, u_n \rangle, \langle u_n, r \rangle \in \mathtt{rf}$ where $u_1, \dots, u_n$ are RMWs, which can be relaxed).[4] Finally, paths composed of the program order (po) and the $\mathtt{sw}_{\mathrm{base}}$ and $\mathtt{sw}_X$ relations form the per-space *happens-before* relation, which again refines the C11 happens-before relation (defined as $(G.\mathrm{po} \cup G.\mathtt{sw}_{\mathrm{base}})^+$).

Using the above definitions, the consistency of an execution graph is defined as follows.

Definition 4.9. An execution graph G is dRC11-*consistent* if the following hold:

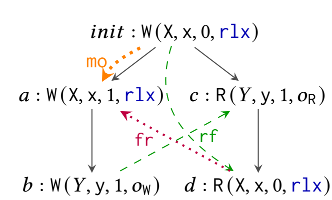
- For every $X \in \mathrm{Space}$, $[\mathsf{R}_X] ; G.\mathtt{fr} ; G.\mathtt{rf}^? ; G.\mathtt{hb}_X$ is irreflexive. (READ COHERENCE)
- For every $X \in \mathrm{Space}$, $[\mathsf{W}_X] ; G.\mathtt{mo} ; G.\mathtt{rf}^? ; G.\mathtt{hb}_X$ is irreflexive. (WRITE COHERENCE)
- $G.\mathtt{mo} ; G.\mathtt{rf}$ is irreflexive. (RMW1)
- $G.\mathtt{fr} ; G.\mathtt{mo}$ is irreflexive. (RMW2)
- $G.\mathrm{po} \cup G.\mathtt{rf}$ is acyclic. (PO-RF)

These constraints, depicted in Fig. 3, are variants of the ones in (R)C11, where the only essential difference is in the coherence constraints that here use $G.\mathtt{hb}_X$ for restricting accesses to X.[5] The RMW1 and RMW2 constraints are needed for ensuring the right behavior of RMWs including their atomicity. The PO-RF constraint is an addition of RC11 on top of C11, which is a conservative solution to the “out-of-thin-air” problem that arises if $\mathrm{po} \cup \mathtt{rf}$-cycles are allowed [Batty et al. 2015; Kang et al. 2017].

The next example demonstrates the role of the coherence constraints:

Example 4.10. Consider the following standard “message-passing” litmus test (parametric in the space Y and the access modes $o_{\mathsf{W}}, o_{\mathsf{R}}$):

$\mathtt{store}(\mathsf{X}, \mathsf{x}, 1, \mathtt{rlx})$;	$\mathsf{a} := \mathtt{load}(Y, \mathsf{y}, o_{\mathsf{R}})$; // 1
$\mathtt{store}(Y, \mathsf{y}, 1, o_{\mathsf{W}})$;	$\mathsf{b} := \mathtt{load}(\mathsf{X}, \mathsf{x}, \mathtt{rlx})$; // 0



The annotated behavior is disallowed only if the synchronization on y (in space Y) is visible for the read of x (in space X). This will be the case only if (1) $o_{\mathsf{W}} = \mathtt{rel}$ and $o_{\mathsf{R}} = \mathtt{acq}$; or (2) $o_{\mathsf{W}} \sqsupseteq \mathtt{prel}$, $o_{\mathsf{R}} \sqsupseteq \mathtt{pacq}$, and $Y = \mathsf{X}$ (i.e., the four accesses are to the same space). (In particular, if $Y \neq \mathsf{X}$, then only $o_{\mathsf{W}} = \mathtt{rel}$ and $o_{\mathsf{R}} = \mathtt{acq}$ would forbid the annotated behavior.) To see how this follows from

[4] Another component of C11’s release sequence, which allows forming synchronization using a relaxed write po-after a release write to the same location, was omitted from the standard in C++20. (See https://en.cppreference.com/w/cpp/atomic/memory_order [Accessed July 2022].)

[5] There are some presentational differences w.r.t. the model in [Lahav et al. 2017]: (1) RC11 uses two events related by an rmw-edge to represent RMWs, so RMW1 is not needed and RMW2 has a different formulation (called ATOMICITY in [Lahav et al. 2017]); and (2) RC11 uses the “extended coherence order” (eco) which allows it to merge both coherence constraints.

the coherence constraints, note that these conditions are needed to ensure $\langle b, c\rangle \in \mathtt{sw}_{\text{base}} \cup \mathtt{sw}_{\mathsf{X}}$, which in turn implies $\langle a, d\rangle \in \mathtt{hb}_{\mathsf{X}}$. Now, since $\langle init, a\rangle \in \mathtt{po} \subseteq \mathtt{hb}_{\mathsf{X}}$, WRITE COHERENCE ensures that $\langle init, a\rangle \in \mathtt{mo}$, and so $\langle d, a\rangle \in \mathtt{fr}$. In turn, READ COHERENCE disallows $\mathtt{hb}_{\mathsf{X}}$ from a to d.

By connecting the definition of candidate execution graphs for a given program (using the FG system) and dRC11-consistency, we define the allowed program behaviors under dRC11 (more precisely, possible reachable program states).

Definition 4.11. A program state $\overline{p}$ is *reachable for a program Pr under* dRC11 if $\langle \overline{p}, G\rangle$ is reachable in $Pr \bowtie \mathsf{FG}$ for some dRC11-consistent execution graph G.

Finally, again following (R)C11, data races on non-atomics are considered as programming errors (so non-atomic accesses can be heavily optimized by the hardware and the compiler). Accordingly, we define racy execution graphs and racy programs.

Definition 4.12. Two events e_1 and e_2 form a *race* in an execution graph G, if $e_1, e_2 \in G.\mathsf{W} \cup G.\mathsf{R}$, $e_1 \neq e_2$, $\mathtt{sp}(e_1) = \mathtt{sp}(e_2)$, $\mathtt{loc}(e_1) = \mathtt{loc}(e_2)$, $|\{e_1, e_2\} \cap \mathsf{W}| \geq 1$, $|\{e_1, e_2\} \cap \mathsf{E}^{\mathsf{na}}| \geq 1$, and $\langle e_1, e_2\rangle \notin G.\mathtt{hb}_{\mathtt{sp}(e_1)} \cup G.\mathtt{hb}^{-1}_{\mathtt{sp}(e_1)}$. An execution graph G is *racy* if some two events form a race in G.

Definition 4.13. A program Pr is *racy* under dRC11 if $\langle \overline{p}, G\rangle$ is reachable in $Pr \bowtie \mathsf{FG}$ for some program state $\overline{p}$ and racy dRC11-consistent execution graph G.

5 THE OPERATIONAL MEMORY SYSTEM: pRC11

In this section we introduce an operational version of the dRC11 memory model, called pRC11, which exposes knowledge propagation steps as internal memory steps, and is particularly suitable for library abstraction.

Since dRC11-consistency is *prefix-closed* w.r.t. $\mathtt{po} \cup \mathtt{rf}$ [Kokologiannakis et al. 2017], dRC11 can be easily made operational by adapting the LTS FG (Def. 4.4) to require dRC11-consistency after each step of the execution graph generation (rather than one time at the end). However, following §2.1, we opt for a more elaborate model with explicit point-to-point *propagation transitions* marking the steps in which some event of thread τ becomes visible to another thread π. In particular, this allows us to observe the propagation of the call/return events in memory traces, which is essential in our definition of the library correctness condition in §7. We expose propagation transitions in traces of pRC11 as internal memory steps labeled with *propagation labels* as defined next.

Definition 5.1. A *propagation label* is a triple, denoted by $p = \mathsf{EP}(e, \tau, X)$, where $e \in \mathsf{E}$ (propagated event), $\tau \in \mathsf{Tid} \setminus \{\mathtt{tid}(e)\}$ (destination thread identifier), and $X \in \mathsf{Space}$ (destination space). We use E, $\mathtt{ptid}$ and $\mathtt{psp}$ to retrieve the components (e, τ, and X, respectively) of a propagation label p. All functions on events and action labels ($\mathtt{tid}$, $\mathtt{typ}$...) are lifted to propagation labels in the obvious way. We denote by PLab the set of all propagation labels.

Then, the pRC11 memory system is defined as follows.

Definition 5.2. The pRC11 memory system is an LTS whose set of transition labels is ProgLab $\cup$ PLab (i.e., pRC11.Θ = PLab); states are pairs of the form $M = \langle G, K\rangle$, where G is an execution graph and K is a *knowledge mapping* for G, that is a function in $\mathsf{Tid} \to \mathsf{Space} \to \mathcal{P}(G.\mathsf{E})$; initial state is $M_{\mathsf{Init}} \triangleq \langle G_{\mathsf{Init}}, K_{\mathsf{Init}}\rangle$, where $K_{\mathsf{Init}} \triangleq \lambda\tau.\, \lambda X.\, \mathsf{Init}$ (G_{Init} is defined in Def. 4.4); and transitions are as given in Fig. 4.

In addition to the current execution graph G, pRC11's states record a *knowledge mapping* K that records for each thread τ and space X all events that have already propagated to τ for space X (as well as all events associated with actions executed by τ itself). We refer to the set $K(\tau)(X)$ as the X-knowledge of thread τ.

$$\textsc{write/read/rmw}\ \frac{\begin{array}{c}\mathsf{typ}(l) \in \{\mathtt{W}, \mathtt{R}, \mathtt{RMW}\} \qquad e = \mathsf{NextEvent}(G.\mathsf{E}, \tau, l, f) \qquad X = \mathsf{sp}(e) \\ w \in G.\mathsf{W}_{X,\mathtt{loc}(e)} \qquad e \in \mathsf{R} \implies \mathsf{val}_{\mathtt{W}}(w) = \mathsf{val}_{\mathtt{R}}(e) \\ w \notin \mathit{dom}(G.\mathtt{mo}\,;G.\mathtt{rf}^?\,;[K(\tau)(X)]) \qquad e \in \mathsf{W} \implies w \notin \mathit{dom}(G.\mathtt{rf}\,;[\mathsf{RMW}]) \\ e \in \mathsf{R}^{\mathtt{pacq}} \implies \mathit{dom}(G.\mathtt{rf}^*\,;[w]) \subseteq K(\tau)(X) \qquad e \in \mathsf{R}^{\mathtt{acq}} \implies \forall Y.\, \mathit{dom}(G.\mathtt{rf}^*\,;[w]) \subseteq K(\tau)(Y) \\ G' = \mathsf{Add}(G, e, w) \qquad k' = \lambda Y.\, K(\tau)(Y) \cup \{e\} \qquad K' = K[\tau \mapsto k']\end{array}}{\langle G, K\rangle \xrightarrow{\tau,l,f}_{\mathsf{pRC11}} \langle G', K'\rangle}$$

$$\textsc{call/return}\ \frac{\begin{array}{c}\mathsf{typ}(l) \in \{\mathtt{CALL}, \mathtt{RET}\} \qquad e = \mathsf{NextEvent}(G.\mathsf{E}, \tau, l, f) \\ G' = \mathsf{Add}(G, e) \qquad k' = \lambda Y.\, K(\tau)(Y) \cup \{e\} \qquad K' = K[\tau \mapsto k']\end{array}}{\langle G, K\rangle \xrightarrow{\tau,l,f}_{\mathsf{pRC11}} \langle G', K'\rangle}$$

$$\textsc{propagate}\ \frac{\begin{array}{c}e \in G.\mathsf{E} \setminus K(\tau)(X) \\ e \in \mathsf{W}_X^{\mathtt{prel}} \cup \mathsf{W}^{\mathtt{rel}} \cup \mathsf{CR} \implies (\mathsf{E}_X \cup \mathsf{CR}) \cap \mathit{dom}(G.\mathtt{hb}_X\,;[e]) \subseteq K(\tau)(X) \\ k' = K(\tau)[X \mapsto K(\tau)(X) \cup \{e\}] \qquad K' = K[\tau \mapsto k']\end{array}}{\langle G, K\rangle \xrightarrow{\mathsf{EP}(e,\tau,X)}_{\mathsf{pRC11}} \langle G, K'\rangle}$$

Fig. 4. Transitions of pRC11.

The WRITE/READ/RMW transition in Fig. 4 executes a memory access by thread τ, by adding a corresponding event e to the current execution graph while imposing certain conditions on w, the write-predecessor of e. Intuitively, the main imposed condition, $w \notin \mathit{dom}(G.\mathtt{mo}\,;G.\mathtt{rf}^?\,;[K(\tau)(X)])$, requires that w is not overwritten by any other write that τ is already aware of for the space of e. More precisely, if e is an access in space X, then τ should not have in its X-knowledge any write that is mo-later than w or any read that reads from a write that is mo-later than w. In addition:

- If e is a write (or RMW), then w should not be already read by an RMW. This condition is needed to ensure the atomicity of RMWs (corresponds to RMW2 in Def. 4.9).
- If e is a read (or RMW) with acq or pacq mode, then w (the write that e reads from) has to be already present in the thread's knowledge: in its X-knowledge if e is pacq, or in Y-knowledge for all Y if e is acq. (Moreover, if w is an RMW, to account for release sequences, this should hold not only for w, but also for every event on the rf-chain entering w.)

Finally, every thread certainly knows about its own actions, so in addition to extending G, this step also extends the knowledge of τ by adding the event e to all spaces.

The CALL/RETURN transition is simple: it adds a corresponding event e to the current execution graph and extends the knowledge of the executing thread to include e.

The PROPAGATE transition is a non-deterministic internal memory step that extends the threads' knowledge. It picks some event e that is not in τ's X-knowledge and adds e to $K(\tau)(X)$. When e is a prel-write to space X, a rel-write (to any space), or a call or return marker, then the propagation of e to the X-knowledge of thread τ can be done only after all accesses to X, as well as all call and return markers, that are $G.\mathtt{hb}_X$-before e have propagated to the X-knowledge of thread τ. What is *not* constrained is equally important: for instance, relaxed writes can propagate "out-of-order".

Example 5.3. pRC11's transitions are best understood via the "message-passing" litmus test presented in Example 4.10. Consider first the case that $o_{\mathtt{W}} = \mathtt{rel}$ and $o_{\mathtt{R}} = \mathtt{acq}$. Then, due to the constraints on acquire reads and release writes: (1) since the read of y is acquire, it has to read from a write that is in T_2's X-knowledge hold for every space X: and (2) since the write of y is release, this write can only propagate to the X-knowledge of T_2 (for every X) after the po-earlier write to x has propagated there. This means that if the read of y retrieves 1, then T_2 already has the write to x in its X-knowledge, so when later reading x it cannot read from the overwritten initial value. Similarly, if $o_{\mathsf{W}} = \mathtt{prel}$, $o_{\mathsf{R}} = \mathtt{pacq}$, and $Y = \mathsf{X}$, then, (i) and (ii) apply for $X = \mathsf{X}$, and the same reasoning holds. Nevertheless, in every other case, T_2 can read the overwritten initial value: because either (1) the *read* from y is too weak and it allows to read from an event that has not yet propagated to the thread's X-knowledge; or (2) the *write* to y is too weak and it can propagate to the thread's X-knowledge before the po-earlier write to X has propagated.

Example 5.4. Being equivalent to dRC11, pRC11 provides "per-location-SC" (a.k.a. coherence). As a concrete example, $\mathsf{a} = 1 \wedge \mathsf{b} = 0$ in the following program on the right is disallowed. To see this in pRC11, note that read events are also included in the threads' knowledge. Concretely, after executing the first read, the second thread has its own read in its X-knowledge. Then, reading later from the (implicit) initialization write is forbidden by pRC11 since that write and the first read will be have $G.\mathtt{mo} \mathbin{;} G.\mathtt{rf}$ between them.

`store(X, x, 1, rlx);`	`a := load(X, x, rlx);` `b := load(X, x, rlx);`

Remark 3. It is also instructive to consider a simplified fragment with only one variable space and without na/rlx accesses. In this fragment, we can simply talk about the knowledge of each thread (instead of the X-knowledge for each X), and observe that: reads by thread τ can only read from writes that thread τ already knows about; and the propagation order respects hb (in particular, it follows the program order). This actually makes the reads deterministic: they have to read-from the mo-latest write among all known writes to the relevant variable. The resulting model is similar to message passing models for causal consistency [Beillahi et al. 2021].

Based on the pRC11 memory system, we define reachable program states and racy programs.

Definition 5.5. A program state $\overline{p}$ is *reachable for a program Pr under* pRC11 if $\langle \overline{p}, \langle G, K\rangle\rangle$ is reachable in $Pr \bowtie$pRC11 for some $\langle G, K\rangle \in$ pRC11.Q. A program Pr is *racy* under pRC11 if $\langle \overline{p}, \langle G, K\rangle\rangle$ is reachable in $Pr \bowtie$pRC11 for some program state $\overline{p}$ and $\langle G, K\rangle \in$ pRC11.Q such that G is a racy execution graph (see Def. 4.12).

5.1 Equivalence of pRC11 and dRC11

We state our equivalence result between pRC11 and dRC11, relating Def. 5.5 to Definitions 4.11 and 4.13.

Theorem 5.6 (Equivalence of the Models). *A program state $\overline{p}$ is reachable for a program Pr under* dRC11 *iff it is reachable for Pr under* pRC11. *Furthermore, Pr is racy under* dRC11 *iff it is racy under* pRC11.

Next, we describe the main steps in the proof (the full proof is given in §A.1). First, for the right-to-left directions, it suffices to establish the following invariants on reachable pRC11-states.

Definition 5.7. A knowledge mapping K is *well-formed* for an execution graph G if the following hold for every $\tau \in \mathsf{Tid}$ and $X \in \mathsf{Space}$:

(1) $G.\mathsf{E}^{\tau} \subseteq K(\tau)(X)$.
(2) $(\mathsf{E}_X \cup \mathsf{CR}) \cap dom(G.\mathtt{hb}_X \mathbin{;} [(\mathsf{W}_X^{\mathtt{prel}} \cup \mathsf{W}^{\mathtt{rel}}) \cap K(\tau)(X)]) \subseteq K(\tau)(X)$.
(3) $(\mathsf{E}_X \cup \mathsf{CR}) \cap dom(G.\mathtt{hb}_X \mathbin{;} [\mathsf{E}^{\tau}]) \subseteq K(\tau)(X)$.

Proposition 5.8. *If $\langle G, K\rangle$ is reachable in* pRC11*, then K is well-formed for G.*

Lemma 5.9. *If* $\langle G, K\rangle$ *is reachable in* pRC11*, then* G *is* dRC11*-consistent.*

For the converse, one starts with an dRC11-consistent execution graph G, and has to traverse its events (following the program order, so it can be syncronized with the program), and intersperse propagation actions to make it a valid trace of pRC11. To construct this traversal, we define the following relations, where $P \triangleq \{p \in \mathsf{PLab} \mid \mathsf{E}(p) \in G.\mathsf{E}\}$:

$$R_{\mathsf{prop}} \triangleq \{\langle e, p\rangle \in G.\mathsf{E} \times P \mid \mathsf{E}(p) = e\}$$

$$T \triangleq \left\{\langle p, p'\rangle \in P \times P \;\middle|\; \begin{array}{l}\mathtt{ptid}(p) = \mathtt{ptid}(p') \wedge \mathtt{psp}(p) = \mathtt{psp}(p') \wedge \\ \langle \mathsf{E}(p), \mathsf{E}(p')\rangle \in [\mathsf{E}_{\mathtt{psp}(p)} \cup \mathsf{CR}] \,;G.\mathsf{hb}_{\mathtt{psp}(p)} \,;[\mathsf{W}^{\mathtt{prel}}_{\mathtt{psp}(p)} \cup \mathsf{W}^{\mathtt{rel}} \cup \mathsf{CR}]\end{array}\right\}$$

$$R_{\mathsf{rfp}} \triangleq \{\langle p, e\rangle \in P \times G.\mathsf{E} \mid \langle \mathsf{E}(p), e\rangle \in G.\mathsf{rf}^{+} \wedge \mathtt{ptid}(p) = \mathtt{tid}(e) \wedge e \in \mathsf{R}^{\mathtt{pacq}}_{\mathtt{psp}(p)}\}$$

$$R_{\mathsf{rf}} \triangleq \{\langle p, e\rangle \in P \times G.\mathsf{E} \mid \langle \mathsf{E}(p), e\rangle \in G.\mathsf{rf}^{+} \wedge \mathtt{ptid}(p) = \mathtt{tid}(e) \wedge e \in \mathsf{R}^{\mathtt{acq}}\}$$

$$R_{\mathsf{fr}} \triangleq \{\langle r, p\rangle \in G.\mathsf{E} \times P \mid \langle r, \mathsf{E}(p)\rangle \in G.\mathsf{fr} \,; G.\mathsf{rf}^{?} \wedge \mathtt{ptid}(p) = \mathtt{tid}(r) \wedge \mathtt{psp}(p) = \mathtt{sp}(r)\}$$

$$R_{\mathsf{mo}} \triangleq \{\langle w, p\rangle \in G.\mathsf{E} \times P \mid \langle w, \mathsf{E}(p)\rangle \in G.\mathsf{mo} \,; G.\mathsf{rf}^{?} \wedge \mathtt{ptid}(p) = \mathtt{tid}(w) \wedge \mathtt{psp}(p) = \mathtt{sp}(w)\}$$

$$R \triangleq G.\mathsf{po} \cup G.\mathsf{rf} \cup R_{\mathsf{prop}} \cup T \cup R_{\mathsf{rfp}} \cup R_{\mathsf{rf}} \cup R_{\mathsf{fr}} \cup R_{\mathsf{mo}}$$

Then, the proof proceeds by showing that R is acyclic, and that every total order of $G.\mathsf{E} \cup P$ extending R induces a trace of pRC11. Indeed, the relations above are in one-to-one correspondence with the conditions in the steps of pRC11.

6 LIBRARIES AND THEIR CLIENTS

In this section we list the necessary definitions for the library abstraction theorem, and state the key properties that are used in its proof.

Client-library composition. A library L is a function mapping a set $dom(L) \subseteq \mathsf{F}$ of method names to flat instruction sequences representing the method bodies. We only consider the case where libraries and their clients never access the same variable space. To formally define this syntactic restriction, we use the following notations for spaces used by libraries and their clients:

- $\mathsf{Space}(I)$ denotes the set of variable spaces mentioned in an instruction sequence I.
- For a library L, $\mathsf{Space}(L) \triangleq \bigcup_{f \in dom(L)} \mathsf{Space}(L(f))$.
- For a program Pr and a set $F \subseteq \mathsf{F}$, $\mathsf{Space}(Pr \setminus F) \triangleq \bigcup_{\tau \in \mathsf{Tid}, f \in (\mathsf{F} \cup \{\mathtt{main}\}) \setminus F} \mathsf{Space}(Pr(\tau)(f))$.

Then, client-library composition is defined as follows.

Definition 6.1. A library L is *safe* for a program Pr if $\mathsf{Space}(L) \cap \mathsf{Space}(Pr \setminus dom(L)) = \emptyset$. When L is safe for Pr, we write $Pr[L]$ for the program obtained from Pr by setting $Pr(\tau)(f) = L(f)$ for every $\tau \in \mathsf{Tid}$ and $f \in dom(L)$.

Next, we observe that the safety condition above ensures that execution graphs generated by $Pr[L]$ satisfy the following conditions relating the location spaces and the invoked methods of the graph events.

Proposition 6.2. *Let* L *be a library that is safe for a program* Pr*. Suppose that* $\langle \overline{p}, G\rangle$ *is reachable in* $Pr[L] \bowtie \mathsf{FG}$*. Then, the following hold for every* $e \in G.\mathsf{R} \cup G.\mathsf{W}$*:*

- *If* $\mathtt{sp}(e) \in \mathsf{Space}(L)$*, then* $\mathtt{method}(e) \in dom(L)$*.*
- *If* $\mathtt{sp}(e) \in \mathsf{Space}(Pr \setminus dom(L))$*, then* $\mathtt{method}(e) \notin dom(L)$*.*

Definition 6.3. A set $F \subseteq \mathsf{F}$ is *encapsulated in an execution graph* G if for every $e_1, e_2 \in G.\mathsf{R} \cup G.\mathsf{W}$ with $\mathtt{method}(e_1) \in F$ and $\mathtt{method}(e_2) \notin F$, we have $\mathtt{sp}(e_1) \neq \mathtt{sp}(e_2)$.

From Prop. 6.2, we obtain the following.

Proposition 6.4. *Let L be a library that is safe for a program Pr. If $\langle \overline{p}, G\rangle$ is reachable in $Pr[L] \bowtie \mathsf{FG}$, then $dom(L)$ is encapsulated in G.*

The notions of well-formedness and encapsulated set of methods are lifted to memory states in the obvious way:

Definition 6.5. Let $M = \langle G, K\rangle$ be a memory state.

- $M = \langle G, K\rangle$ is *well-formed* if G is well-formed (Def. 4.7) and K is well-formed for G (Def. 5.7).
- F is *encapsulated in M* if it is encapsulated in G (Def. 6.3).

Since every reachable state in $Pr[L] \bowtie \mathsf{pRC11}$ is also reachable in $Pr[L] \bowtie \mathsf{FG}$, the following is an immediate consequence of Propositions 4.8, 5.8 and 6.4.

Proposition 6.6. *Let L be a library that is safe for a program Pr. If $\langle \overline{p}, M\rangle$ is reachable in $Pr[L] \bowtie \mathsf{pRC11}$, then M is well-formed and $dom(L)$ is encapsulated in M.*

Client-library program states. We define the composition of a program state $\overline{p}_{\mathsf{cl}}$ representing a client state and a program state $\overline{p}_{\mathsf{lib}}$ representing a library state as follows.

Definition 6.7. The *composition of two program states $\overline{p}_{\mathsf{cl}}$ and $\overline{p}_{\mathsf{lib}}$ w.r.t. a set $F \subseteq \mathsf{F}$*, denoted by $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$, is given by:

$$\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}] \triangleq \lambda\tau. \begin{cases} \langle \overline{p}_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}_{\mathsf{lib}}(\tau).\phi, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc_s}, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{f}\rangle & \overline{p}_{\mathsf{cl}}(\tau).\mathsf{f} \in F \\ \overline{p}_{\mathsf{cl}}(\tau) & \text{otherwise} \end{cases}$$

This definition uses $\overline{p}_{\mathsf{cl}}$ for threads that are not currently inside a method in F, and $\overline{p}_{\mathsf{lib}}$, but with the stored program counter and active method of $\overline{p}_{\mathsf{cl}}$, for threads that are inside a method in F.

Histories. Histories record the interactions between libraries and clients. Formally, a *history h* of a library L is a sequence of transition labels representing a call to a method of L, a return from a method of L, or propagation of these call and return events. To define the history induced by a program, we employ the following notations (for every $F \subseteq \mathsf{F}$):

$$\begin{array}{rr} \mathsf{Call}_F \triangleq \{\alpha \in \mathsf{ProgLab} \mid \mathtt{typ}(\alpha) = \mathtt{CALL} \land \mathtt{callee}(\alpha) \in F\} & \mathsf{CP}_F \triangleq \{p \in \mathsf{PLab} \mid \mathsf{E}(p) \in \mathsf{CALL}_F\} \\ \mathsf{Ret}_F \triangleq \{\alpha \in \mathsf{ProgLab} \mid \mathtt{typ}(\alpha) = \mathtt{RET} \land \mathtt{method}(\alpha) \in F\} & \mathsf{RP}_F \triangleq \{p \in \mathsf{PLab} \mid \mathsf{E}(p) \in \mathsf{RET}_F\} \end{array}$$
$$\mathsf{HLab}_F \triangleq \mathsf{Call}_F \cup \mathsf{Ret}_F \cup \mathsf{CP}_F \cup \mathsf{RP}_F$$

Definition 6.8. Let $F \subseteq \mathsf{F}$. The *F-history* induced by a trace t of $Pr \bowtie \mathsf{pRC11}$ for some program Pr, denoted by $\mathsf{H}_F(t)$, is given by $\mathsf{H}_F(t) \triangleq t|_{\mathsf{HLab}_F}$. This notion is extended to sets of traces in the obvious way. The set of *F-histories of a program Pr*, denoted by $\mathsf{H}_F(Pr)$, is given by $\mathsf{H}_F(Pr) \triangleq \mathsf{H}_F(\mathsf{traces}(Pr \bowtie \mathsf{pRC11}))$.

Client-library trace restrictions. We extract library and client transitions from a given trace as follows.

Definition 6.9. For $F \subseteq \mathsf{F}$, the *F-restriction* and the *$\overline{F}$-restriction* of a trace t of $Pr \bowtie \mathsf{pRC11}$, denoted by $t|_F$ and $t|_{\overline{F}}$ (respectively), are given by:

$$t|_F \triangleq t|_{\{\alpha \in \mathsf{ProgLab} \cup \mathsf{PLab} \mid \mathtt{method}(\alpha) \in F\} \cup \mathsf{Call}_F \cup \mathsf{CP}_F} \qquad t|_{\overline{F}} \triangleq t|_{\{\alpha \in \mathsf{ProgLab} \cup \mathsf{PLab} \mid \mathtt{method}(\alpha) \notin F\} \cup \mathsf{Ret}_F \cup \mathsf{RP}_F}$$

Note that both the F-restriction and the $\overline{F}$-restriction of a trace t contain the F-history induced by t as a subsequence.

Restricting memory states. Similarly, we will need to restrict memory states to client/library, as defined next.

Definition 6.10. The *restriction of an execution graph* G w.r.t. a set $E \subseteq \mathsf{E}$, denoted by $G|_E$, is defined by: $G|_E.\mathsf{E} \triangleq E \cup \mathsf{Init}$ and $G|_E.\mathsf{X} \triangleq [G|_E.\mathsf{E}] \,; G.\mathsf{X} \,; [G|_E.\mathsf{E}]$ for every other component (i.e., $\mathsf{X} \in \{\mathsf{po}, \mathsf{rf}, \mathsf{mo}\}$).

Definition 6.11. The *restriction of a memory state* $M = \langle G, K\rangle$ w.r.t. a set $E \subseteq \mathsf{E}$, denoted by $M|_E$, is given by $M|_E \triangleq \langle G|_E, K|_E\rangle$, where $K|_E \triangleq \lambda\tau.\, \lambda X.\, K(\tau)(X) \cap E$.

Definition 6.12. Let $F \subseteq \mathsf{F}$. The F*-events* and the $\overline{F}$*-events* denoted by E_F and $\mathsf{E}_{\overline{F}}$ (respectively), are given by:

$$\mathsf{E}_F \triangleq \{e \in \mathsf{E} \mid \mathtt{method}(e) \in F\} \cup \mathsf{CALL}_F \qquad \mathsf{E}_{\overline{F}} \triangleq \{e \in \mathsf{E} \mid \mathtt{method}(e) \notin F\} \cup \mathsf{RET}_F$$

The F*-restriction* and the $\overline{F}$*-restriction* of a memory state $M = \langle G, K\rangle$, denoted by $M|_F$ and $M|_{\overline{F}}$ (respectively), are given by $M|_F \triangleq M|_{\mathsf{E}_F}$ and $M|_{\overline{F}} \triangleq M|_{\mathsf{E}_{\overline{F}}}$.

Again, we note that both the F-restriction and the $\overline{F}$-restriction of a memory M contain the call events invoking methods in F and the return events that complete these invocations.

Restriction and merge properties. The following lemmas summarize the critical properties of the pRC11 memory system that allow us to compose memory traces of clients and libraries. In the proof of the abstraction theorem below we rely only on these properties of pRC11.

Lemma 6.13 (Restriction-1). *Suppose that* $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$ *and let* $F \subseteq \mathsf{F}$ *that is encapsulated in* M'*. Then, the following hold:*

(1) If $\mathtt{method}(\alpha) \in F$ *or* $\alpha \in \mathsf{HLab}_F$*, then* $M|_F \xrightarrow{\alpha}_{\mathsf{pRC11}} M'|_F$.
(2) If $\mathtt{method}(\alpha) \notin F$ *or* $\alpha \in \mathsf{HLab}_F$*, then* $M|_{\overline{F}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M'|_{\overline{F}}$.

Lemma 6.14 (Restriction-2). *Suppose that* $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$ *and let* $F \subseteq \mathsf{F}$*. Then, the following hold:*

(1) If $\mathtt{method}(\alpha) \notin F$ *and* $\alpha \notin \mathsf{HLab}_F$*, then* $M|_F = M'|_F$.
(2) If $\mathtt{method}(\alpha) \in F$ *and* $\alpha \notin \mathsf{HLab}_F$*, then* $M|_{\overline{F}} = M'|_{\overline{F}}$.

Lemma 6.15 (Merge). *Suppose that* $F \subseteq \mathsf{F}$ *is encapsulated in a well-formed memory state* $M = \langle G, K\rangle$*. Then, the following hold:*

(1) Let α *be an* pRC11 *transition label with* $\mathtt{method}(\alpha) \in F$*. Suppose that if* $\alpha \in \mathsf{ProgLab}$ *and* $\mathtt{typ}(\alpha) \in \{\mathtt{W}, \mathtt{R}, \mathtt{RMW}\}$*, then* $\mathtt{method}(e) \in F$ *for every* $e \in G.\mathsf{E}$ *with* $\mathtt{sp}(e) = \mathtt{sp}(\alpha)$*. Then,* $M|_F \xrightarrow{\alpha}_{\mathsf{pRC11}} M'_F$ *implies that* $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$ *for some* M' *such that* $M'|_F = M'_F$.
(2) Let α *be an* pRC11 *transition label with* $\mathtt{method}(\alpha) \notin F$*. Suppose that if* $\alpha \in \mathsf{ProgLab}$ *and* $\mathtt{typ}(\alpha) \in \{\mathtt{W}, \mathtt{R}, \mathtt{RMW}\}$*, then* $\mathtt{method}(e) \notin F$ *for every* $e \in G.\mathsf{E}$ *with* $\mathtt{sp}(e) = \mathtt{sp}(\alpha)$*. Then,* $M|_{\overline{F}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M'_{\overline{F}}$ *implies that* $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$ *for some* M' *such that* $M'|_{\overline{F}} = M'_{\overline{F}}$.

7 THE LIBRARY ABSTRACTION THEOREM

In this section we state and prove the library abstraction theorem. This theorem assumes two libraries implementing the same set of methods, an *implementation* L and a *specification* $L^\#$ with $F = dom(L) = dom(L^\#)$, and two programs, a *client* Pr and a *most general client* MGC. The latter is representing the library's calling policy.

Example 7.1. For a library with no restrictions whatsoever on its clients (beyond the separation of variable spaces) one can use a most general client that repeatedly invokes arbitrary library methods with arbitrary stores. We denote this client by *MGC*free. On the right we present the code of the main method in each thread τ in *MGC*free for $dom(L) = \{f_1, ..., f_n\}$. We use `havoc` for arbitrarily modifying all registers.

```
BEGIN : havoc; goto f1 | ... | fn;
f1 : call(f1); goto BEGIN;
...
fn : call(fn); goto BEGIN;
```

Example 7.2. Alternatively, a policy that requires to call the library methods in a race-free fashion is captured by a most general client that uses for each thread the code on the right. Here, we hold a lock L while executing every method. We assume a standard lock implementation using release/acquire accesses (see §8).

```
BEGIN : havoc; goto f1 | ... | fn;
f1 : acquire(L); call(f1);
    release(L); goto BEGIN;
...
fn : acquire(L); call(fn);
    release(L); goto BEGIN;
```

Beyond syntactic separation of variable spaces between libraries and their clients (see Def. 6.1), the library abstraction theorem has two conditions:

- L should *refine* $L^\#$ w.r.t. *MGC*, denoted by $L \sqsubseteq_{MGC} L^\#$, and formally defined by:

$$L \sqsubseteq_{MGC} L^\# \quad \overset{\triangle}{\iff} \quad \mathsf{H}_F(MGC[L]) \subseteq \mathsf{H}_F(MGC[L^\#]) \land MGC[L] \text{ is not racy under pRC11}$$

- Pr should *adhere to MGC* w.r.t. $L^\#$, denoted by $Pr \sqsubseteq_{L^\#} MGC$, and formally defined by:

$$Pr \sqsubseteq_{L^\#} MGC \quad \overset{\triangle}{\iff} \quad \mathsf{H}_F(Pr[L^\#]) \subseteq \mathsf{H}_F(MGC[L^\#]) \land Pr[L^\#] \text{ is not racy under pRC11}$$

The first condition is an obligation of the library developer. It requires that all histories generated for the most general client using the implementation are also generated with the specification. Additionally, the implementation should not have data races when run by the most general client. Importantly, both conditions do not mention Pr: library developers have to be able to verify their implementations "once and for all" without access to a particular client program.

Dually, the second condition is an obligation of the client. It requires that the client adheres to the library policy (otherwise, the blame is on the client), which means that all histories generated by the client program should be also generated by the most general client, which expresses that policy. Additionally, the client program should not have data races. Importantly, both conditions do not mention L: clients should be able to apply the abstraction theorem without access to the library implementation. In fact, it suffices to assume that Pr uses $L^\#$ both when checking for adherence to the library's calling policy and for checking for data-race freedom (this can be important for both checks if the calling policy relies on the values returned by the library).

Now, what should the theorem guarantee? Intuitively, we want all client behaviors observable when using L to be observable when using $L^\#$. Thus, for every trace t generated by $Pr[L]$ that reaches a program state $\overline{p}$ and a memory state M, there should exist a corresponding trace $t^\#$ generated by $Pr[L^\#]$ that reaches a program state $\overline{p}^\#$ and a memory state $M^\#$, and a client should not be able to observe the difference between t and $t^\#$, $\overline{p}$ and $\overline{p}^\#$, and M and $M^\#$. This, however, does not mean that there is not any difference between these objects: L and $L^\#$ may perform different operations (leading to different traces), use different variables (leading to different memories), and internally use different registers (leading to different program states). We capture "client-equivalence" by requiring that: (1) the $\overline{F}$-parts of t and $t^\#$ coincide (using Def. 6.9); (2) the state $\overline{p}^\#$ is obtained from $\overline{p}$ by only changing the instruction sequence state for threads that are currently inside a method in F (using Def. 6.7); and (3) the $\overline{F}$-restriction of M and $M^\#$ coincide (using Def. 6.12). Finally, we also want to ensure that $Pr[L]$ is not racy.

All in all, we reach the following statement of the abstraction theorem.

Theorem 7.3 (Library Abstraction). *Let L and $L^\#$ be libraries implementing the same set F of methods. Let MGC and Pr be programs, such that both L and $L^\#$ are safe for both MGC and Pr. Suppose that $L \sqsubseteq_{MGC} L^\#$ and $Pr \sqsubseteq_{L^\#} MGC$. Then, the following hold:*

- *If $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L]\bowtie\mathrm{pRC11}} \langle \overline{p}, M \rangle$, then there exist $t^\#$ and $\langle \overline{p}^\#, M^\# \rangle$ such that the following hold: (1) $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t^\#}_{Pr[L^\#]\bowtie\mathrm{pRC11}} \langle \overline{p}^\#, M^\# \rangle$; (2) $t^\#|_{\overline{F}} = t|_{\overline{F}}$; (3) $\overline{p}^\# = \overline{p}[F \mapsto \overline{p}_{\mathsf{lib}}]$ for some $\overline{p}_{\mathsf{lib}}$ (in particular, $\overline{p}^\#(\tau) = \overline{p}(\tau)$ whenever $\overline{p}(\tau).\mathsf{f} \notin F$); and (4) $M^\#|_{\overline{F}} = M|_{\overline{F}}$.*
- *$Pr[L]$ is not racy under* pRC11.

In particular, the conditions of Thm. 7.3 ensure that $L \sqsubseteq_{Pr} L^\#$. The following property, which allows compositional verification of a library consisting of several (non-interacting) libraries, is obtained as a corollary of the abstraction theorem as in [Khyzha and Lahav 2021].

Corollary 7.4 (Compositionality). *Let $L_1, \dots, L_n$ be libraries implementing pairwise disjoint sets of methods, such that* $\mathsf{Space}(L_1), \dots, \mathsf{Space}(L_n), \mathsf{Space}(L_1^\#), \dots, \mathsf{Space}(L_n^\#)$, *and* $\mathsf{Space}(MGC \setminus dom(L_1 \uplus \dots \uplus L_n))$ *are pairwise disjoint. Suppose that for every $1 \leq i \leq n$, we have $L_i \sqsubseteq_{MGC_i} L_i^\#$ for $MGC_i = MGC[L_1^\# \uplus \dots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \dots \uplus L_n^\#]$. Then, $L_1 \uplus \dots \uplus L_n \sqsubseteq_{MGC} L_1^\# \uplus \dots \uplus L_n^\#$.*

The rest of this section is devoted to sketch the main steps in the proof of the abstraction theorem. First, we prove a general "composition lemma" which allows us to take a client's portion from one trace and glue it together with a library's portion from another trace, provided that the two traces induce the same history. The proof of this lemma is by induction on the length of the traces, where Lemmas 6.13 to 6.15 provide the main tools for the induction step.

Lemma 7.5 (Composition). *Let L and L' be libraries implementing the same set F of methods such that both are safe for a program Pr, and L is also safe for a program Pr'. Suppose that $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t_{\mathsf{cl}}}_{Pr[L']\bowtie\mathrm{pRC11}} \langle \overline{p}_{\mathsf{cl}}, M_{\mathsf{cl}} \rangle$ and $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t_{\mathsf{lib}}}_{Pr'[L]\bowtie\mathrm{pRC11}} \langle \overline{p}_{\mathsf{lib}}, M_{\mathsf{lib}} \rangle$, with $\mathsf{H}_F(t_{\mathsf{cl}}) = \mathsf{H}_F(t_{\mathsf{lib}})$. Then, $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L]\bowtie\mathrm{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M \rangle$ for some trace t and memory state M such that $t|_{\overline{F}} = t_{\mathsf{cl}}|_{\overline{F}}$, $t|_F = t_{\mathsf{lib}}|_F$, $M|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$, and $M|_F = M_{\mathsf{lib}}|_F$.*

Now, using the composition lemma, we are able to show the following key property.

Lemma 7.6. *Under the conditions of Thm. 7.3,* $\mathsf{H}_F(Pr[L]) \subseteq \mathsf{H}_F(MGC[L^\#])$.

Proof (outline). Assume otherwise, and let h be a shortest history in $\mathsf{H}_F(Pr[L]) \setminus \mathsf{H}_F(MGC[L^\#])$. Let t be a shortest trace of $Pr[L]\bowtie\mathrm{pRC11}$ with $\mathsf{H}_F(t) = h$. Since ε (the empty history) is clearly in $\mathsf{H}_F(MGC[L^\#])$, we know that t is non-empty. Consider the last transition label α in t, and let t' such that $t = t' \cdot \alpha$. The minimality of t ensures that α must be an element of HLab_F (i.e., call, return, call propagation or return propagation of some method in F). Let $h' = \mathsf{H}_F(t')$. The minimality of h further ensures that $h' \in \mathsf{H}_F(MGC[L^\#])$. Let $t'_\#$ and $\langle \overline{p}'_\#, M'_\# \rangle$ such that $\mathsf{H}_F(t'_\#) = h'$ and $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'_\#}_{MGC[L^\#]\bowtie\mathrm{pRC11}} \langle \overline{p}'_\#, M'_\# \rangle$. Let $\langle \overline{p}', M' \rangle$ be such that $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathrm{pRC11}} \langle \overline{p}', M' \rangle \xrightarrow{\alpha}_{Pr[L]\bowtie\mathrm{pRC11}} \langle \overline{p}, M \rangle$. We consider the following cases:

(1) $\alpha \in \mathsf{Call}_F \cup \mathsf{CP}_F$: Using Lemma 7.5 (applied with $L := L^\#$, $L' := L$, $Pr := Pr$, and $Pr' := MGC$), there exist $t''_\#$ and $M''_\#$ such that $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t''_\#}_{Pr[L^\#]\bowtie\mathrm{pRC11}} \langle \overline{p}'[F \mapsto \overline{p}'_\#], M''_\# \rangle$, $\mathsf{H}_F(t''_\#) = h'$, and $M''_\#|_{\overline{F}} = M'|_{\overline{F}}$. Then, using Lemmas 6.13 and 6.15, it is straightforward to show that α is enabled in $\langle \overline{p}'[F \mapsto \overline{p}'_\#], M''_\# \rangle$, and it follows that $h = h' \cdot \alpha = \mathsf{H}_F(t''_\#) \cdot \alpha \in \mathsf{H}_F(Pr[L^\#])$. Since $Pr \sqsubseteq_{L^\#} MGC$, we obtain $h \in \mathsf{H}_F(MGC[L^\#])$, which contradicts our assumption.

(2) $\alpha \in \mathsf{Ret}_F \cup \mathsf{RP}_F$: Using Lemma 7.5 (applied with $L := L$, $L' := L^\#$, $Pr := MGC$, and $Pr' := Pr$), there exist t'' and M'' such that $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t''}_{MGC[L]\bowtie \mathsf{pRC11}} \langle \overline{p}'_\#[F \mapsto \overline{p}'], M''\rangle$, $\mathsf{H}_F(t'') = h'$, and $M''|_F = M'|_F$. Then, using Lemmas 6.13 and 6.15, it is straightforward to show that α is enabled in $\langle \overline{p}'_\#[F \mapsto \overline{p}'], M''\rangle$, and it follows that $h = h' \cdot \alpha = \mathsf{H}_F(t'') \cdot \alpha \in \mathsf{H}_F(MGC[L])$. Since $L \sqsubseteq_{MGC} L^\#$, we obtain $h \in \mathsf{H}_F(MGC[L^\#])$, which contradicts our assumption. □

Then, to prove the abstraction theorem we need one additional lemma.

Lemma 7.7. *Suppose that $M_{\mathsf{Init}} \xrightarrow{t}_{\mathsf{pRC11}} \langle G, K\rangle$. Then, there exist a trace t' and a knowledge mapping K' for G such that the following hold:*

- $M_{\mathsf{Init}} \xrightarrow{t'}_{\mathsf{pRC11}} \langle G, K'\rangle$.
- $t'|_{\mathsf{ProgLab}} = t|_{\mathsf{ProgLab}}$.
- $K'(\tau, X) \cap (\mathsf{W}^{\mathtt{prel}}_X \cup \mathsf{W}^{\mathtt{rel}} \cup \mathsf{CR}) \subseteq dom(G.\mathsf{hb}^?_X \,;\, [\mathsf{E}^\tau])$ *for every* $\tau \in \mathsf{Tid}$ *and* $X \in \mathsf{Space}$.

The abstraction theorem is proved as follows.

Proof of Thm. 7.3 (sketch). By Lemma 7.6, we have $\mathsf{H}_F(Pr[L]) \subseteq \mathsf{H}_F(MGC[L^\#])$. Then, the first part of the claim directly follows using Lemma 7.5 by letting $L := L^\#$, $L' := L$, $Pr := Pr$, and $Pr' := MGC$. For the second part, suppose that $Pr[L]$ is racy under pRC11. Let $\langle \overline{p}, \langle G, K\rangle\rangle$ be a reachable state in $Pr[L]\bowtie\mathsf{pRC11}$ such that G is racy, and let e_1, e_2 be two events that form a race in G. Let $E = dom((G.\mathsf{po} \cup G.\mathsf{rf})^* \,;\, [e_1, e_2])$ and $G' = G|_E$. Clearly, e_1 and e_2 form a race in G'. It is also easy to see that there exist $\overline{p}'$ and K' such that $\langle \overline{p}', \langle G', K'\rangle\rangle$ is reachable in $Pr[L]\bowtie\mathsf{pRC11}$. By Lemma 7.7, there exists a knowledge mapping K'' for G' such that $\langle \overline{p}', \langle G', K''\rangle\rangle$ is reachable in $Pr[L]\bowtie\mathsf{pRC11}$ as well and $K''(\tau, X) \cap \mathsf{CR} \subseteq dom(G'.\mathsf{hb}^?_X \,;\, [\mathsf{E}^\tau])$ for every $\tau \in \mathsf{Tid}$ and $X \in \mathsf{Space}$. By Prop. 6.6, F is encapsulated in $\langle G', K''\rangle$. Thus, since $\mathsf{sp}(e_1) = \mathsf{sp}(e_2)$, either (i) $e_1 \in \mathsf{E}_{\overline{F}} \wedge e_2 \in \mathsf{E}_{\overline{F}}$ or (ii) $e_1 \in \mathsf{E}_F \wedge e_2 \in \mathsf{E}_F$. If (i) holds, we obtain a contradiction by obtaining a reachable state in $Pr[L^\#]\bowtie\mathsf{pRC11}$ in which e_1 and e_2 form a race. If (ii) holds, we obtain a contradiction by obtaining a reachable state in $MGC[L]\bowtie\mathsf{pRC11}$ in which e_1 and e_2 form a race. □

8 ILLUSTRATIVE APPLICATIONS

In this section we present three examples of applications of library abstraction of different kinds. To simplify the presentation, we use an extended programming language with standard loops and conditionals, mechanisms for passing arguments in method calls/returns, arrays, etc. While these are not directly supported in our development above, the intention should be clear and the translation to programs as defined in §3 is straightforward (but tedious). Each of the libraries below has its own private variable space denoted by $\mathsf{X_L}$. For simplicity, we use macros for lock operations:

$$\begin{aligned}
\mathtt{acquire}(x) &\triangleq \mathsf{LOCK} : \mathtt{l} := \mathtt{CAS}(\mathsf{X_L}, x, 0, 1, \mathtt{acq}, \mathtt{rlx}, \mathtt{rlx}); \mathtt{if}\ \mathtt{l} \neq 0\ \mathtt{goto}\ \mathsf{LOCK}\\
\mathtt{release}(x) &\triangleq \mathtt{store}(\mathsf{X_L}, x, 0, \mathtt{rel})\\
\mathtt{pacquire}(x) &\triangleq \mathsf{LOCK} : \mathtt{l} := \mathtt{CAS}(\mathsf{X_L}, x, 0, 1, \mathtt{pacq}, \mathtt{rlx}, \mathtt{rlx}); \mathtt{if}\ \mathtt{l} \neq 0\ \mathtt{goto}\ \mathsf{LOCK}\\
\mathtt{prelease}(x) &\triangleq \mathtt{store}(\mathsf{X_L}, x, 0, \mathtt{prel})
\end{aligned}$$

8.1 Read-Copy-Update Synchronization

Read-copy-update (RCU) is a synchronization mechanism, heavily used in the Linux kernel, that allows a (single) writer to safely manipulate a data structure while multiple readers are concurrently accessing it [Mckenney 2004]. For that matter, it provides *RCU critical sections* for the readers (delimited by `rcu_read_lock` and `rcu_read_unlock`) and a synchronization method (`synchronize_rcu`) for the writer that waits for all readers currently in critical sections to exit them. Readers should access the structure inside a critical section and the writer should not perform destructive updates before calling the synchronization method [Desnoyers et al. 2012].

Intuitively, every invocation of `synchronize_rcu` begins another 'grace period', and the implementation ensures that "read-side critical sections cannot span grace periods".[6] Following various long-lasting informal discussions among developers, Alglave et al. [2018] formalized this guarantee in the context of their proposed Linux Kernel memory model. They provided specialized ad hoc semantics to the RCU primitives in the form of declarative consistency constraints.

Next, we use our framework to demonstrate a simple specification of RCU under weak memory that is based on *standard locks*. We believe that our specification has the advantage of being more parsimonious and amenable to formal verification of client programs using RCU: it can be understood by relying solely on the semantics of locks, and allows verification using techniques and tools that already support reasoning about locks.

Our specification is given below. It assumes an MGC in which a particular thread (the writer) is repeatedly calling `synchronize_rcu`, and each other thread is a reader that interleaves invocations of `rcu_read_lock` and `rcu_read_unlock` with its own thread identifier, thus acquiring and releasing a per-reader lock $\mathtt{l}[\tau]$. The set *Readers* consist of all thread identifiers except for the writer's identifier, and the `foreach` loop iterates over this set in an arbitrary order (which may vary between invocations).

```
rcu_read_lock(τ) :
  acquire(l[τ]);
  return();
```

```
rcu_read_unlock(τ) :
  release(l[τ]);
  return();
```

```
synchronize_rcu :
  foreach  τ ∈ Readers
    acquire(l[τ]);
    release(l[τ]);
  return();
```

Each RCU critical section is protected by a per-reader lock. For synchronization, the writer acquires and immediately releases each of the reader locks. This ensures that RCU critical sections do not span grace periods. For example, using the specification, it is easy to conclude that the following behavior of the client program is disallowed (as usual, 0 is the initial value of all variables):

`store(X, x, 1, rlx);`
`synchronize_rcu();`
`store(X, y, 1, rlx);`
	`rcu_read_lock(T2);`
	`a := load(X, x, rlx); // 0`
	`b := load(X, y, rlx); // 1`
	`rcu_read_unlock(T2);`

Indeed, roughly speaking, since the read of x returns 0, we know that the writer must have acquired the reader's lock after the reader has released it. In turn, the value 1 has been written to y only after the reader exited its critical section.

Remark 4. Gotsman et al. [2013] introduced a verification technique *assuming SC* that is based on a simpler RCU specification. In their specification each reader thread τ sets a flag $\mathtt{rcs}[\tau]$ to 1 when entering the critical section, and resets it to 0 when exiting the critical section. The writer waits for all reader flags to be 0, which directly means that a critical section cannot span over more than one grace period. Using release/acquire accesses, their specification is as follows:

[6]See https://lwn.net/Articles/573497/ [Accessed July 2022].

```
rcu_read_lock(τ) :
  store(X_L, rcs[τ], 1, rel);
  return();
```

```
rcu_read_unlock(τ) :
  store(X_L, rcs[τ], 0, rel);
  return();
```

```
synchronize_rcu :
  foreach τ ∈ Readers
    repeat
      a := load(X_L, rcs[τ], acq)
    until a = 0;
  return();
```

Interestingly, this specification generates the same invocation-response sequences (i.e., same histories as employed in standard linearizability) as the one we propose. Nevertheless, it is *too weak* under weak memory concurrency, and, in fact, it allows the annotated behavior of the client above.

Following [Alglave et al. 2018; Desnoyers et al. 2012], we present a simplified implementation of the RCU library with weak memory constructs. It uses a global and per-reader 'phase bits', phase and rphase[τ], as well as per-reader flags, rcs[τ]. For correctness, it employs barriers (smp_mb in Linux or atomic_thread_fence(memory_order_seq_cst) in C11). While these are not included in the model we consider, they can be implemented here by acquire-release RMWs to an otherwise unused variable [Lahav et al. 2016]. Thus, fence() below is syntactic sugar for r := FADD($\mathsf{X_L}$, f, 0, acq, rel).

```
rcu_read_lock(τ) :
  a := load(X_L, phase, acq);
  store(X_L, rphase[τ], a, rel);
  store(X_L, rcs[τ], 1, rel);
  fence();
  return();
```

```
rcu_read_unlock(τ) :
  store(X_L, rcs[τ], 0, rel);
  return();
```

```
synchronize_rcu :
  fence();
  store(X_L, phase, 1, rel);
  foreach τ ∈ Readers
    repeat
      a := load(X_L, rcs[τ], acq);
      b := load(X_L, rphase[τ], acq)
    until a = 0 ∨ b = 1;
  store(X_L, phase, 0, rel);
  foreach τ ∈ Readers
    repeat
      a := load(X_L, rcs[τ], acq);
      b := load(X_L, rphase[τ], acq)
    until a = 0 ∨ b = 0;
  return();
```

In this implementation, for entering the critical section each reader τ stores phase in rphase[τ] and announces it enters the critical section by setting rcs[τ] to 1. When exiting the critical section, the reader resets rcs[τ] to 0. For the writer synchronization, the writer switches phase from 0 to 1, waits for each reader until it sees rcs[τ] = 0 or rphase[τ] = 1, switches phase back to 0, and waits again for each reader until it sees rcs[τ] = 0 or rphase[τ] = 0.

To verify refinement (i.e., to show $\mathsf{H}_F(MGC[L]) \subseteq \mathsf{H}_F(MGC[L^\#])$) we used the FDR refinement checker [Gibson-Robinson et al. 2014], which has been used before for automatic linearizability checking [Lowe 2017], and gives us guarantees up to a certain bound. Concretely, we managed to complete the automatic check for two readers with at most four unpropagated events between every two threads and for three readers with at most one unpropagated event. Nevertheless, we note that the length of the traces being checked is unbounded. A standard limitation here is the modeling gap between the paper definitions and FDR's encoding, which uses communicating sequential processes (CSP). Using the fact that the library specification and implementation employ solely rel/acq accesses, which implies that propagation has to follow the program order and reads only read from propagated writes (see Remark 3), we can model pRC11's memory state using FIFO buffers, which are supported by FDR. (But, using Thm. 7.3, refinement is guaranteed also when non-rel/acq accesses are used by clients as in the example above.) We leave a systematic FDR encoding, and possibly the verification of more efficient variants of the above implementation that use rlx accesses to future work.

Finally, we note that while every history of the implementation is also a history of the specification, the implementation has important advantages over the specification. For instance, consider a scenario where a reader is repeatedly entering critical sections and the writer invokes synchronize_rcu. With the specification, to complete its invocation, the writer has to be "lucky" and catch a free lock between two reader's sections. In turn, with the implementation, the writer is able to complete its invocation using the phase bit, without any particular "lucky" scheduling.

8.2 A Relaxed Concurrent Queue

Our second example continues the discussion in §2.2 and presents a specification and an implementation of a weak concurrent queue. The queue object is specified as follows:

```
enqueue(v) :
  pacquire(LT);
  tp := load(X_L, T, rlx);
  store(X_L, q[tp], v, rlx);
  store(X_L, T, tp + 1, rlx);
  prelease(LT);
  return();
```

```
dequeue :
 pacquire(LH);
 hp := load(X_L, H, rlx);
 hc := load(X_L, q[hp], rlx);
 if hc ≠ ⊥ then
  store(X_L, H, hp + 1, rlx);
 prelease(LH);
 return(hc);
```

The specification code uses an unbounded array q (represented by infinitely many variables q[0], q[1], ...) for the queue elements that initially contains only ⊥. The variables H and T store the index of the current head and tail of the queue. Then, the specification wraps a sequential implementation inside locks, making three choices (aiming to demonstrate the framework's flexibility):

- Except for the locks, all accesses are *relaxed*, so synchronization is only induced by the locks.
- The locks are accessed using *partial* acquire/release. This means that the queue does not expose its internal synchronization to the client, thus allowing all behaviors demonstrated in §2.2.
- The enqueue and dequeue methods use *two different locks*, which allows for certain "non-linearizable" behaviors. For example, after enqueue returns in one thread, a dequeue by another thread can retrieve ⊥ (that signifies an empty queue). Indeed, a relaxed read from q[k] may return either the value that was written to q[k] or the initial value. To avoid these behaviors, clients, if they want, may form their own synchronization between enqueues and dequeues (and analyze possible behaviors of the queue using the above specification).

A possible implementation that avoids the locks and uses only relaxed accesses is shown next.

```
enqueue(v) :
 BEGIN : tp := load(X_L, T, rlx);
 if tp ≠ 0 then
  tc := load(X_L, q[tp − 1], rlx);
  if tc = ⊥ goto BEGIN;
 tp' := CAS(X_L, T, tp, tp + 1, rlx, rlx, rlx);
 if tp' ≠ tp goto BEGIN;
 store(X_L, q[tp'], v, rlx);
 return();
```

```
dequeue :
 BEGIN : hp := load(X_L, H, rlx);
 hc := load(X_L, q[hp], rlx);
 if hc = ⊥ then
  return(⊥);
 else
  hp' := CAS(X_L, H, hp, hp + 1, rlx, rlx, rlx);
  if hp' ≠ hp goto BEGIN;
  return(hc);
```

This implementation uses CAS instructions to update H and T. It exploits the freedom allowed by the specification: no synchronization is ever induced between queue operations, and certain "non-linearizable" behaviors are allowed (when the read of q[hp] returns the overwritten ⊥ value). We note that it is important that `enqueue` waits for the previous cell being filled, which corresponds to all `enqueue`'s being totally ordered in the specification. In §A.5 we provide a sketch of the refinement proof, which generally follows a standard simulation argument albeit assuming non-SC memory and considering call/return propagation as observable transitions.

8.3 Local Data-Race-Freedom as an Instance of Library Abstraction

Our third application relies on the library policy component (*MGC*) of the abstraction theorem to derive a *local data-race-freedom guarantee* (LDRF, for short) for the underlying memory model (a fragment of RC11).[7] Local data-race-freedom guarantees, introduced in [Dolan et al. 2018] and further developed in [Cho et al. 2021], generalize the more well-known (global) data-race-freedom guarantee, by ensuring strong semantics for locations accessed by non-racy executions. Crucially, the premise of these guarantees (i.e., what locations are racy) is checked assuming the strong semantics for the specified locations, which makes LDRF particularly useful for modular reasoning [Cho et al. 2021].

In our case, in the role of the strong semantics we have release/acquire (RA, for short) semantics (thus we establish what is called LDRF-RA in [Cho et al. 2021]), and we want to show that if a program avoids races on a set S of variables, then it is safe to assume that the accesses to S have RA semantics (as if they have rel/acq access modes). Moreover, it suffices to establish the premise, namely the avoidance of races on S, assuming the accesses to S have RA semantics.

To formulate this intricate property in terms of refinement, consider a library implementing methods $\texttt{write}_x$ and $\texttt{read}_x$ for every variable $x \in S$. The library specification ($L^\#$) is as follows:

$\texttt{write}_x(v):\quad \texttt{store}(\mathsf{X_L}, x, v, \mathsf{rel}); \texttt{return}();$	$\texttt{read}_x:\quad \mathsf{a} := \texttt{load}(\mathsf{X_L}, x, \mathsf{acq}); \texttt{return}(\mathsf{a});$

The implementation (L) employs non-atomic accesses (it could use any access mode):

$\texttt{write}_x(v):\quad \texttt{store}(\mathsf{X_L}, x, v, \mathsf{na}); \texttt{return}();$	$\texttt{read}_x:\quad \mathsf{a} := \texttt{load}(\mathsf{X_L}, x, \mathsf{na}); \texttt{return}(\mathsf{a});$

Then, the assurance provided by LDRF is precisely (conditioned) contextual refinement between L and $L^\#$. Now, to express the data-race-freedom premise we take *MGC*drf to be a program that repeatedly and non-deterministically calls the library methods with arbitrary arguments, but avoids races by properly using standard per-variable readers-writer lock (which allows concurrent read operations but write operations require exclusive access). That is, before calling $\texttt{write}_x$ (respectively, $\texttt{read}_x$) *MGC*drf takes a lock on x with a write-mode (respectively, read-mode).

Adherence to the calling policy as specified by *MGC*drf (i.e., $Pr \sqsubseteq_{L^\#} MGC\mathsf{drf}$) is equivalent to the LDRF premise. In particular, $\mathsf{H}_{dom(L)}(Pr[L^\#]) \subseteq \mathsf{H}_{dom(L)}(MGC\mathsf{drf}[L^\#])$ means that in histories in $\mathsf{H}_{dom(L)}(Pr[L^\#])$ for every variable $x \in S$: (1) before calling $\texttt{write}_x$ by thread τ, the return markers of all previous invocations of $\texttt{write}_x$ and $\texttt{read}_x$ have propagated to thread τ; and (2) before calling $\texttt{read}_x$ by thread τ, the return markers of all previous invocations of $\texttt{write}_x$ have propagated to thread τ. In addition, adherence to the policy in our abstraction theorem assumes $L^\#$ (rather than L) just like LDRF's premise assuming RA semantics for accesses to S.

Accordingly, by applying the abstraction theorem we can establish LDRF-RA by proving our library correctness criterion $L \sqsubseteq_{MGC\mathsf{drf}} L^\#$ for the libraries above. This is straightforward: the conditions on propagation events of calls and returns that arise by $L^\#$ are already covered by *MGC*drf. For instance, if an invocation of $\texttt{read}_x$ reads from an invocation of $\texttt{write}_x$ in an execution of $MGC\mathsf{drf}[L^\#]$, then due to the rel/acq accesses by $L^\#$, the call $\texttt{write}_x$ has to propagate before $\texttt{read}_x$ returns, but this already holds in traces of $MGC\mathsf{drf}[L^\#]$ due to *MGC*drf.

[7] To the best of our knowledge, we are the first to observe that (local) data-race-freedom guarantee is an instance of library abstraction.

9 RELATED WORK

We have adopted several key instruments proposed in earlier work to achieve library abstraction. Concretely, Burckhardt et al. [2012] addressed this challenge for the x86-TSO architecture [Owens et al. 2009], and achieved library abstraction by including call/return markers in the thread-local store buffers, and exposing the propagation of these markers to the main memory in library histories. We adapt this idea to allow point-to-point communication and apply it in a much weaker memory model. (Moreover, [Burckhardt et al. 2012] does not consider libraries that rely on synchronization by the client or not meant to expose their internal synchronization to the client, since these issues do not arise on x86-TSO.) In turn, from [Khyzha and Lahav 2022], which studied library abstraction under non-volatile memory, we adopt the general formalism, the proof strategy of the abstraction theorem by relying on a general composition lemma (Lemma 7.5), and the modeling of a library's calling policy as a client program (*MGC*).

Another closely related work is [Batty et al. 2013], which provides the first library abstraction result for C11. For having simple specifications (which they want, like us, to be pieces of code) Batty et al. [2013] extended (a fragment of) C11 with specialized "atomic block" constructs, whose semantics is similar to the semantics of software transactions. In contrast, we use locks with "partial release/acquire" semantics, and have no need in including transactional features, which may be harder for client reasoning. Other crucial differences is that [Batty et al. 2013] does not support library's calling policies, and their correctness condition is based on partially ordered execution traces (with 'guarantee' and 'deny' relations), while we opt for considering only totally ordered histories, which, we believe, are easier to grasp. We also note that the relaxed atomics employed in [Batty et al. 2013] have the original C11 semantics (*without* the PO-RF constraint, and thus *with* "out-of-thin-air" behaviors), which, as they show, does not allow fully compositional reasoning.

Library abstraction under weak memory was also studied from a *declarative* point of view in [Dongol et al. 2018; Raad et al. 2019], using partial orders for exposing the externally visible synchronization induced by concurrent objects. Concretely, in the framework of [Raad et al. 2019], libraries are specified as collections of execution graphs, and the abstraction condition relates the graphs produced by the implementation to those in the specification. Their specification framework does not depend on the language constructs, and it is sufficiently expressive for having direct specifications of non-standard non-linearizable objects (e.g., queues with certain non-FIFO behaviors). We believe that there is a price for the generality: rich declarative specifications are often hard to understand and informally apply, and require non-standard methods for client reasoning. In addition, [Raad et al. 2019]'s 'well-formedness' requirement (which corresponds to our "policy adherence") is defined in terms of the implementation library (unlike in our abstraction theorem).

Smith et al. [2020] studied linearizability on a general hardware memory model, and related it to a certain refinement notion, which they call "object refinement". In contrast, we are aiming for standard contextual refinement between an implementation and its specification code. Other works developed correctness conditions for concurrent objects under weakly consistent memory, but were not formally related to a refinement notion (e.g., [Doherty et al. 2018]).

Weakly consistent objects were also studied by Emmi and Enea [2019] who proposed a specification framework based on extending standard histories with a *visibility* relation that is subject to varying constraints. Their approach was applied to non-linearizable Java concurrent objects, and allows verification by a generalization of the concept of linearization points [Krishna et al. 2020].

Other recent works propose *logical* approaches to library specification and verification under weak memory [Dalvandi and Dongol 2021a,b; Dang et al. 2022; Mével and Jourdan 2021]. In contrast to our work, their specifications are given as Hoare triples and refinement is understood from a program logic perspective. Specifically, for a fragment of the model we study (without non-atomics), Dalvandi and Dongol [2021a,b] propose an Owicki-Gries-style Hoare logic and specify libraries with view-based object semantics. Their approach is, however, limited to clients who synchronize only through the library operations. In turn, Mével and Jourdan [2021] and Dang et al. [2022] are based on the idea of *logical atomicity* [Birkedal et al. 2021; Jung 2019], and provide mechanized verification of certain library implementations in the Iris framework [Jung et al. 2015]. The weak memory model in [Dang et al. 2022] is similar to the one we study (RC11), while the model in [Mével and Jourdan 2021] is Multicore OCaml, which provides much stronger accesses than RC11. The library studied in [Mével and Jourdan 2021] is a queue that is stronger than our example in §8.2. It exposes synchronization to the client between from an enqueue of some element to the dequeue of that specific element, but still, unlike a lock-based queue, does not expose synchronization in all other cases (e.g., between two different enqueues).

Finally, the idea of having explicit steps for propagating knowledge between threads appeared before in semantics of relaxed memory concurrency, particularly in an operational model for the POWER architecture [Sarkar et al. 2011], but we are not aware of a similar existing model for RC11.

10 CONCLUSION

We established a library abstraction for (a fragment of) the RC11 model extended with specialized partial release/acquire accesses. The latter are used to have simple lock-based specifications of libraries that do not expose internal synchronization to their clients or libraries that rely on synchronization by the client. The condition for library abstraction is based on inclusion of sets of totally ordered library histories. To achieve this, we developed an operational version of the model with explicit steps for propagation of knowledge between threads, and included the propagation of method invocation and responses in library histories.

Our work has several limitations, which are interesting to lift and address in future work. In particular: (1) The fragment of RC11 that we consider excludes fences and SC-atomics; (2) We only consider partial correctness and finite histories ignoring liveness issues and thus our refinement notion is not termination preserving (see [Gotsman and Yang 2011] for a possible approach to lift this limitation); (3) We disallow nested method calls (in particular, recursion), as common in works relating linearizability to refinement (also under SC); and (4) We disallow libraries and their clients to transfer ownership of data structures among themselves or to run in a shared address space (see [Gotsman and Yang 2013] for a possible approach to lift this limitation).

Verification techniques for both library development and client reasoning are beyond the scope of the current work. In particular, it is interesting to see how our specification constructs can be supported by a program logic, which will pave the way to foundational mechanized refinement proofs as in [Dang et al. 2022], as well as by model checking tools, such as GenMC [Kokologiannakis et al. 2019] and C11Tester [Luo and Demsky 2021], that can be used for client reasoning.

Finally, we are interested in developing a view-based operational model (see, e.g., [Kaiser et al. 2017; Kang et al. 2017]) with explicit propagation steps that will be equivalent to pRC11, but possibly more intuitive. We also believe that pRC11 (in particular, its `rel`/`acq` fragment, see Remark 3) may be of independent interest for developing verification techniques, by relying on the analogy between pRC11's knowledge propagation and well-studied x86-TSO store propagation steps (see e.g., [Bouajjani et al. 2018; Enea and Farzan 2016]).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811) and the Israel Science Foundation (grant numbers 1566/18 and 814/22).

REFERENCES

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *ASPLOS*. ACM, New York, NY, USA, 405–418. https://doi.org/10.1145/3173162.3177156

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. In *POPL*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/2429069.2429099

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *ESOP*. Springer, Berlin, Heidelberg, 283–307. http://dx.doi.org/10.1007/978-3-662-46669-8_12

Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2021. Robustness Against Transactional Causal Consistency. *Logical Methods in Computer Science* Volume 17, Issue 1 (Feb. 2021). https://doi.org/10.23638/LMCS-17(1:12)2021

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for Free from Separation Logic Specifications. *Proc. ACM Program. Lang.* 5, ICFP, Article 81 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473586

Hans-Juergen Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *MSPC*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Tractable Refinement Checking for Concurrent Objects. In *POPL*. ACM, New York, NY, USA, 651–662. https://doi.org/10.1145/2676726.2677002

Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Serdar Tasiran. 2018. Reasoning About TSO Programs Using Reduction and Abstraction. In *CAV*. Springer International Publishing, Cham, 336–353. https://doi.org/10.1007/978-3-319-96142-2_21

Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Concurrent Library Correctness on the TSO Memory Model. In *ESOP*. Springer, Berlin, Heidelberg, 87–107. https://doi.org/10.1007/978-3-642-28869-2_5

Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Modular Data-Race-Freedom Guarantees in the Promising Semantics. In *PLDI*. ACM, New York, NY, USA, 867–882. https://doi.org/10.1145/3453483.3454082

Sadegh Dalvandi and Brijesh Dongol. 2021a. Verifying C11-Style Weak Memory Libraries. In *PPoPP*. ACM, New York, NY, USA, 451–453. https://doi.org/10.1145/3437801.3441619

Sadegh Dalvandi and Brijesh Dongol. 2021b. Verifying C11-Style Weak Memory Libraries via Refinement. *CoRR* abs/2108.06944 (2021). arXiv:2108.06944 https://arxiv.org/abs/2108.06944

Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic. In *PLDI*. ACM, New York, NY, USA, 792–808. https://doi.org/10.1145/3519939.3523451

Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Trans. Parallel Distrib. Syst.* 23, 2 (2012), 375–382. https://doi.org/10.1109/TPDS.2011.159

Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2018. Making Linearizability Compositional for Partially Ordered Executions. In *iFM*. Springer International Publishing, Cham, 110–129. https://doi.org/10.1007/978-3-319-98938-9_7

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *PLDI*. ACM, New York, NY, USA, 242–255. https://doi.org/10.1145/3192366.3192421

Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. 2018. On Abstraction and Compositionality for Weak-Memory Linearisability. In *VMCAI*. Springer International Publishing, Cham, 183–204. https://doi.org/10.1007/978-3-319-73721-8_9

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

Constantin Enea and Azadeh Farzan. 2016. On Atomicity in Presence of Non-atomic Writes. In *TACAS*. Springer, Berlin, Heidelberg, 497–514. https://doi.org/10.1007/978-3-662-49674-9_29

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theoretical Computer Science* 411, 51 (2010), 4379–4398. https://www.sciencedirect.com/science/article/pii/S0304397510005001

Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew W. Roscoe. 2014. FDR3 — A Modern Refinement Checker for CSP. In *TACAS*. Springer, Berlin, Heidelberg, 187–201. https://doi.org/10.1007/978-3-642-54862-8_13

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

Alexey Gotsman and Hongseok Yang. 2011. Liveness-Preserving Atomicity Abstraction. In *ICALP*. Springer, Berlin, Heidelberg, 453–465. https://doi.org/10.1007/978-3-642-22012-8_36

Alexey Gotsman and Hongseok Yang. 2013. Linearizability with Ownership Transfer. *Logical Methods in Computer Science* Volume 9, Issue 3 (Sept. 2013). https://lmcs.episciences.org/931

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

Ralf Jung. 2019. Logical atomicity in Iris: The good, the bad, and the ugly. In *Iris Workshop*. https://people.mpi-sws.org/~jung/iris/talk-iris2019.pdf

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *POPL*. ACM, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

Artem Khyzha and Ori Lahav. 2021. Taming x86-TSO Persistency. *Proc. ACM Program. Lang.* 5, POPL, Article 47 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434328

Artem Khyzha and Ori Lahav. 2022. Abstraction for Crash-Resilient Objects. In *ESOP*. Springer International Publishing, Cham, 262–289. https://doi.org/10.1007/978-3-030-99336-8_10

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158105

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *PLDI*. ACM, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

Siddharth Krishna, Michael Emmi, Constantin Enea, and Dejan Jovanović. 2020. Verifying Visibility-Based Weak Consistency. In *ESOP*. Springer International Publishing, Cham, 280–307. https://doi.org/10.1007/978-3-030-44914-8_11

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. In *POPL*. ACM, New York, NY, USA, 649–662. https://doi.org/10.1145/2837614.2837643

Ori Lahav and Roy Margalit. 2019. Robustness Against Release/Acquire Semantics. In *PLDI*. ACM, New York, NY, USA, 126–141. https://doi.org/10.1145/3314221.3314604

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

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

Gavin Lowe. 2017. Analysing Lock-Free Linearizable Datatypes Using CSP. In *Concurrency, Security, and Puzzles - Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*. Springer, 162–184. https://doi.org/10.1007/978-3-319-51046-0_9

Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. In *ASPLOS*. ACM, New York, NY, USA, 630–646. https://doi.org/10.1145/3445814.3446711

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

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

Peizhao Ou and Brian Demsky. 2018. Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 136 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276506

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs*. Springer, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019. On Library Correctness under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries under Declarative Consistency Models. *Proc. ACM Program. Lang.* 3, POPL, Article 68 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290381

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *PLDI*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/1993498.1993520

Graeme Smith, Kirsten Winter, and Robert J. Colvin. 2020. Linearizability on Hardware Weak Memory Models. *Form. Asp. Comput.* 32, 1 (Feb. 2020), 1–32. https://doi.org/10.1007/s00165-019-00499-8

A PROOFS

In this appendix we provide proof outlines that were not included in the main text. We skip proofs of propositions that straightforwardly follow from our definitions.

A.1 Proofs for Section 5

Lemma 5.9. *If $\langle G, K\rangle$ is reachable in* pRC11*, then G is* dRC11*-consistent.*

Proof. The proof is by induction on the length of the trace reaching $\langle G, K\rangle$. For the empty trace, the claim is obvious. Consider an pRC11-step transitioning from a reachable state $\langle G, K\rangle$ to $\langle G', K'\rangle$. By the induction hypothesis, G is dRC11-consistent. For the call/return and propagate steps of pRC11, the dRC11-consistency of G' easily follows from the fact that G is dRC11-consistent. Consider a write/read/rmw step: $\langle G, K\rangle \xrightarrow{\tau,l,f}_{\mathsf{pRC11}} \langle G', K'\rangle$ with $\mathtt{typ}(l) \in \{\mathtt{W}, \mathtt{R}, \mathtt{RMW}\}$. Let $e = \mathsf{NextEvent}(G.\mathsf{E}, \tau, l, f)$, and let $w \in G.\mathsf{W}_{\mathtt{sp}(e),\mathtt{loc}(e)}$ such that the following hold:

- $G' = \mathsf{Add}(G, e, w)$.
- $w \notin dom(G.\mathtt{mo} \,;G.\mathtt{rf}^? \,; [K(\tau)(\mathtt{sp}(e))])$.
- $e \in \mathsf{W} \implies w \notin dom(G.\mathtt{rf} \,; [\mathsf{RMW}])$.
- $K' = K' = K[\tau \mapsto k']$, where $k' = \lambda Y.\, K(\tau)(Y) \cup \{e\}$.

By Prop. 5.8, we also know that K' is a well-formed knowledge mapping for G'. We prove each of the consistency conditions for G':

- read coherence: Let $X \in \mathsf{Space}$. We show that $[\mathsf{R}_X]\,; G'.\mathtt{fr}\,; G'.\mathtt{rf}^?\,; G'.\mathtt{hb}_X$ is irreflexive. Since e is $G'.\mathsf{po} \cup G'.\mathtt{rf}$-maximal, it suffices to show that $\langle e, e\rangle \notin [\mathsf{R}_X]\,;G'.\mathtt{fr}\,;G'.\mathtt{rf}^?\,;G'.\mathtt{hb}_X$. Suppose otherwise, and let $e' \in G'.\mathsf{E}$ such that $\langle e, e'\rangle \in [\mathsf{R}_X]\,; G'.\mathtt{fr}\,; G'.\mathtt{rf}^?$ and $\langle e', e\rangle \in G'.\mathtt{hb}_X$. In this case, we have $e \in \mathsf{R}_X$. Then, the fact that $\langle e, e'\rangle \in G'.\mathtt{fr};G'.\mathtt{rf}^?$ implies that $\langle w, e'\rangle \in G.\mathtt{mo};G.\mathtt{rf}^?$. In addition, since $\langle e', e\rangle \in G'.\mathtt{hb}_X$, the fact that K' is a well-formed knowledge mapping for G' implies that $e' \in K'(\tau)(X)$, which entails that $e' \in K(\tau)(X)$ (since $e \neq e'$). This contradicts the fact that $w \notin dom(G.\mathtt{mo}\,; G.\mathtt{rf}^?\,; [K(\tau)(X)])$.
- write coherence: Let $X \in \mathsf{Space}$. We show that $[\mathsf{W}_X];G'.\mathtt{mo};G'.\mathtt{rf}^?;G'.\mathtt{hb}_X$ is irreflexive. Since e is $G'.\mathsf{po} \cup G'.\mathtt{rf}$-maximal, it suffices to show that $\langle e, e\rangle \notin [\mathsf{W}_X]\,;G'.\mathtt{mo};G'.\mathtt{rf}^?;G'.\mathtt{hb}_X$. Suppose otherwise, and let $e' \in G'.\mathsf{E}$ such that $\langle e, e'\rangle \in [\mathsf{W}_X]\,; G'.\mathtt{mo}\,; G'.\mathtt{rf}^?$ and $\langle e', e\rangle \in G'.\mathtt{hb}_X$. In this case, we have $e \in \mathsf{W}_X$. Then, the fact that $\langle e, e'\rangle \in G'.\mathtt{mo};G'.\mathtt{rf}^?$ implies that $\langle w, e'\rangle \in G.\mathtt{mo};G.\mathtt{rf}^?$. In addition, since $\langle e', e\rangle \in G'.\mathtt{hb}_X$, the fact that K' is a well-formed knowledge mapping for G' implies that $e' \in K'(\tau)(X)$, which entails that $e' \in K(\tau)(X)$ (since $e \neq e'$). This contradicts the fact that $w \notin dom(G.\mathtt{mo}\,; G.\mathtt{rf}^?\,; [K(\tau)(X)])$.
- rmw1: We show that $G'.\mathtt{mo}\,; G'.\mathtt{rf}$ is irreflexive. Suppose otherwise, and let $e_1, e_2 \in G'.\mathsf{E}$ such that $\langle e_1, e_2\rangle \in G'.\mathtt{mo}$ and $\langle e_2, e_1\rangle \in G'.\mathtt{rf}$. This implies that $e_1 \in \mathsf{RMW}$. Since e has no outgoing $\mathtt{rf}$-edges in G', we have $e_2 \neq e$. Now, if $e_1 \neq e$, then we have $\langle e_1, e_2\rangle \in G.\mathtt{mo}$ and $\langle e_2, e_1\rangle \in G.\mathtt{rf}$, which contradicts the fact that G satisfies rmw1. On the other hand, we cannot have $e_1 = e$ as well, since when adding the $\mathtt{rf}$-edge to an RMW event, we make its source to be the $\mathtt{mo}$-immediate predecessor of the added event (see definition of $\mathsf{Add}(G, e, w)$).
- rmw2: We show that $G'.\mathtt{fr}\,; G'.\mathtt{mo}$ is irreflexive. Suppose otherwise, and let $e_1, e_2 \in G'.\mathsf{E}$ such that $\langle e_1, e_2\rangle \in G'.\mathtt{fr}$ and $\langle e_2, e_1\rangle \in G'.\mathtt{mo}$. This implies that $e_1 \in \mathsf{RMW}$. Let e_3 such that $\langle e_3, e_1\rangle \in G'.\mathtt{rf}$ and $\langle e_3, e_2\rangle \in G'.\mathtt{mo}$. Since e has no outgoing $\mathtt{rf}$-edges in G', we have $e_3 \neq e$. In addition, we have $e_1 \neq e$ as well, since when adding the $\mathtt{rf}$-edge to an RMW event, we make its source to be the $\mathtt{mo}$-immediate predecessor of the added event. Now, if $e_2 \neq e$, we obtain $\langle e_1, e_2\rangle \in G.\mathtt{fr}$ and $\langle e_2, e_1\rangle \in G.\mathtt{mo}$, which contradicts the fact that G satisfies rmw2. Otherwise, we have $e_2 = e$. Then, $\langle e_2, e_1\rangle \in G'.\mathtt{mo}$ implies that $\langle w, e_1\rangle \in G.\mathtt{mo}$, and $\langle e_3, e\rangle \in G'.\mathtt{mo}$ implies that $\langle e_3, w\rangle \in G.\mathtt{mo}^?$. Now, if $e_3 \neq w$, we obtain $\langle e_1, w\rangle \in G.\mathtt{fr}$ and $\langle w, e_1\rangle \in G.\mathtt{mo}$, which again contradicts the fact that G

satisfies RMW2. On the other hand, we cannot have $e_3 = w$, since this contradicts the fact that $w \notin dom(G.\mathtt{rf} ; [\mathsf{RMW}])$ when $e \in \mathsf{W}$.

- PO-RF: Easily follows by the facts that G satisfies PO-RF and that e is $G'.\mathsf{po} \cup G'.\mathtt{rf}$-maximal. □

Theorem 5.6 (Equivalence of the Models). *A program state $\overline{p}$ is reachable for a program Pr under* dRC11 *iff it is reachable for Pr under* pRC11. *Furthermore, Pr is racy under* dRC11 *iff it is racy under* pRC11.

Proof. It suffices to show that for every program state $\overline{p}$ and execution graph G the following are equivalent:

(a) $\langle \overline{p}, G\rangle$ is reachable in $Pr \bowtie \mathsf{FG}$ and G is dRC11-consistent.
(b) $\langle \overline{p}, \langle G, K\rangle\rangle$ is reachable in $Pr \bowtie \mathsf{pRC11}$ for some knowledge mapping K.

Given Lemma 5.9, $(b) \implies (a)$ is easy: Since for every $\alpha \in \mathsf{ProgLab}$, $\langle G, K\rangle \xrightarrow{\alpha}_{\mathsf{pRC11}} \langle G', K'\rangle$ implies $G \xrightarrow{\alpha}_{\mathsf{FG}} G'$, it suffices to consider an pRC11 trace and show that the execution graph component in the final state of that run is dRC11-consistent, which follows by Lemma 5.9.

Next, we prove that $(a) \implies (b)$. Suppose that $\langle \overline{p}, G\rangle$ is reachable in $Pr \bowtie \mathsf{FG}$ and G is dRC11-consistent. Relying on commutativity of operations of different threads in the program semantics (i.e., in the LTS induced by Pr), it suffices to demonstrate an enumeration $e_1, \dots, e_n$ of $G.\mathsf{E} \setminus \mathsf{Init}$ that respects $G.\mathsf{po}$, such that

$$\langle \mathtt{tid}(e_1), \mathtt{lab}(e_1), \mathtt{method}(e_1)\rangle, \dots, \langle \mathtt{tid}(e_n), \mathtt{lab}(e_n), \mathtt{method}(e_n)\rangle$$

is obtained as a restriction of some trace of pRC11 to ProgLab. We construct this enumeration by taking an arbitrary total order extending several relations as defined next:

$$\begin{aligned}
P &\triangleq \{p \in \mathsf{PLab} \mid \mathsf{E}(p) \in G.\mathsf{E}\} \\
R_{\mathsf{prop}} &\triangleq \{\langle e, p\rangle \in G.\mathsf{E} \times P \mid \mathsf{E}(p) = e\} \\
T &\triangleq \left\{\langle p, p'\rangle \in P \times P \;\middle|\; \begin{array}{l} \mathtt{ptid}(p) = \mathtt{ptid}(p') \land \mathsf{psp}(p) = \mathsf{psp}(p') \land \\ \langle \mathsf{E}(p), \mathsf{E}(p')\rangle \in [\mathsf{E}_{\mathsf{psp}(p)} \cup \mathsf{CR}] ; G.\mathsf{hb}_{\mathsf{psp}(p)} ; [\mathsf{W}^{\mathsf{prel}}_{\mathsf{psp}(p)} \cup \mathsf{W}^{\mathsf{rel}} \cup \mathsf{CR}] \end{array}\right\} \\
R_{\mathtt{rfp}} &\triangleq \{\langle p, e\rangle \in P \times G.\mathsf{E} \mid \langle \mathsf{E}(p), e\rangle \in G.\mathtt{rf}^+ \land \mathtt{ptid}(p) = \mathtt{tid}(e) \land e \in \mathsf{R}^{\mathsf{pacq}}_{\mathsf{psp}(p)}\} \\
R_{\mathtt{rf}} &\triangleq \{\langle p, e\rangle \in P \times G.\mathsf{E} \mid \langle \mathsf{E}(p), e\rangle \in G.\mathtt{rf}^+ \land \mathtt{ptid}(p) = \mathtt{tid}(e) \land e \in \mathsf{R}^{\mathsf{acq}}\} \\
R_{\mathtt{fr}} &\triangleq \{\langle r, p\rangle \in G.\mathsf{E} \times P \mid \langle r, \mathsf{E}(p)\rangle \in G.\mathtt{fr} ; G.\mathtt{rf}^? \land \mathtt{ptid}(p) = \mathtt{tid}(r) \land \mathsf{psp}(p) = \mathsf{sp}(r)\} \\
R_{\mathsf{mo}} &\triangleq \{\langle w, p\rangle \in G.\mathsf{E} \times P \mid \langle w, \mathsf{E}(p)\rangle \in G.\mathsf{mo} ; G.\mathtt{rf}^? \land \mathtt{ptid}(p) = \mathtt{tid}(w) \land \mathsf{psp}(p) = \mathsf{sp}(w)\} \\
R &\triangleq G.\mathsf{po} \cup G.\mathtt{rf} \cup R_{\mathsf{prop}} \cup T \cup R_{\mathtt{rfp}} \cup R_{\mathtt{rf}} \cup R_{\mathtt{fr}} \cup R_{\mathsf{mo}}
\end{aligned}$$

Claim A.0.1: T is transitive.
Claim A.0.2: $R_{\mathsf{prop}} ; (R_{\mathtt{rfp}} \cup R_{\mathtt{rf}}) \subseteq \bigcup_X G.\mathtt{rf}^+$.
Claim A.0.3: $R_{\mathsf{prop}} ; T^+ ; (R_{\mathtt{rfp}} \cup R_{\mathtt{rf}}) \subseteq \bigcup_X G.\mathsf{hb}_X$.
Claim A.0.4: $(R_{\mathtt{fr}} \cup R_{\mathsf{mo}}) ; (R_{\mathtt{rfp}} \cup R_{\mathtt{rf}}) \subseteq \bigcup_X [\mathsf{E}_X] ; (G.\mathtt{fr} \cup G.\mathsf{mo}) ; G.\mathtt{rf}^+$.
Claim A.0.5: $(R_{\mathtt{fr}} \cup R_{\mathsf{mo}}) ; T^+ ; (R_{\mathtt{rfp}} \cup R_{\mathtt{rf}}) \subseteq \bigcup_X [\mathsf{E}_X] ; (G.\mathtt{fr} \cup G.\mathsf{mo}) ; G.\mathtt{rf}^? ; G.\mathsf{hb}_X$.
Claim A.0.6: $(R_{\mathtt{fr}} \cup R_{\mathsf{mo}}) ; T^* ; (R_{\mathtt{rfp}} \cup R_{\mathtt{rf}}) \subseteq G.\mathsf{po}$.

Proof. Let $\langle e_1, e_2\rangle \in (R_{\mathtt{fr}} \cup R_{\mathsf{mo}}) ; T^* ; (R_{\mathtt{rfp}} \cup R_{\mathtt{rf}})$. First, note that we must have $\mathtt{tid}(e_1) = \mathtt{tid}(e_2)$, and so we either have $\langle e_1, e_2\rangle \in G.\mathsf{po}$ or $\langle e_2, e_1\rangle \in G.\mathsf{po}$. Assume, by way of contradiction, that $\langle e_2, e_1\rangle \in G.\mathsf{po}$. By the previous two claims, we have $\langle e_1, e_2\rangle \in \bigcup_X [\mathsf{E}_X] ; (G.\mathtt{fr} \cup G.\mathsf{mo}) ; (G.\mathtt{rf}^? ; G.\mathsf{hb}_X \cup G.\mathtt{rf}^+)$. Hence, we obtain a loop in the relation $[\mathsf{E}_X];(G.\mathtt{fr} \cup G.\mathsf{mo});(G.\mathtt{rf}^?;G.\mathsf{hb}_X \cup G.\mathtt{rf}^+); G.\mathsf{po}$, which entails a loop in $[\mathsf{E}_X] ; (G.\mathtt{fr} \cup G.\mathsf{mo}) ; G.\mathtt{rf}^? ; G.\mathsf{hb}_X$ or in $(G.\mathtt{fr} \cup G.\mathsf{mo}) ; G.\mathtt{rf}^+ ; G.\mathsf{po}$, while both are forbidden by dRC11-consistency. ◃

Claim A.0.7: $[\mathsf{E}] ; R^{+} ; [\mathsf{E}] \subseteq (G.\mathsf{po} \cup G.\mathtt{rf})^{+}$.

Proof. It is easy to see that $[\mathsf{E}] ; R^{+} ; [\mathsf{E}] \subseteq (G.\mathsf{po} \cup G.\mathtt{rf} \cup (R_{\mathsf{prop}} \cup R_{\mathtt{fr}} \cup R_{\mathsf{mo}}) ; T^{*} ; (R_{\mathtt{rfp}} \cup R_{\mathtt{rf}}))^{+}$. Then, the claim easily follows from the previous claims. ◁

Claim A.0.8: R is acyclic.

Proof. R-cycles between propagation actions are T-loops (since T is transitive), whereas R-cycles between events are $G.\mathsf{po} \cup G.\mathtt{rf}$-cycles (using the previous claim). Both are forbidden by the PO-RF constraint in dRC11-consistency. ◁

Now, let $\alpha_1, \dots, \alpha_n$ be an enumeration of $(G.\mathsf{E} \setminus \mathsf{Init}) \cup P$ that respects R. We prove that $\beta_1, \dots, \beta_n$ is a trace of pRC11, where:

$$\beta_i = \begin{cases} \langle \mathtt{tid}(\alpha_i), \mathtt{lab}(\alpha_i), \mathtt{method}(\alpha_i) \rangle & \alpha_i \in G.\mathsf{E} \\ \alpha_i & \alpha_i \in P \end{cases}$$

For that matter, let $G^0 = G_{\mathsf{Init}}$ and $K^0 = K_{\mathsf{Init}}$, and for every $1 \leq i \leq n$, let: $E_i = \mathsf{E} \cap \{\alpha_1, \dots, \alpha_i\}$, $G^i = G|_{E_i}$, and $K^i = \lambda\tau.\, \lambda X.\, \mathsf{Init} \cup G.\mathsf{E}^{\tau} \cup \{e \in \mathsf{E} \mid \exists k \leq i.\, \alpha_k = \mathsf{EP}(e, \tau, X)\}$. We prove by induction on i that $\langle G^i, K^i \rangle \xrightarrow{\beta_i}_{\mathsf{pRC11}} \langle G^{i+1}, K^{i+1} \rangle$ for every $0 \leq i \leq n-1$:

- Case 1—$\alpha_i \in G.\mathsf{E}$ and $\mathtt{typ}(\alpha_i) \in \{\mathtt{W}, \mathtt{R}, \mathtt{RMW}\}$: Let $X = \mathtt{sp}(\alpha_i)$, $\tau = \mathtt{tid}(\alpha_i)$, and $k = K^i(\tau)$. Let w be the immediate $G^{i+1}.\mathsf{mo}$-predecessor of α_i if $\alpha_i \in \mathsf{W}$, or the $G^{i+1}.\mathtt{rf}$-source of α_i if $\alpha_i \in \mathsf{R}$. Then, we have $\alpha_i = \mathsf{NextEvent}(G^i.\mathsf{E}, \tau, \mathtt{lab}(\alpha_i), \mathtt{method}(\alpha_i))$, $w \in G^i.\mathsf{W}_{X, \mathtt{loc}(\alpha_i)}$, $\alpha_i \in \mathsf{R} \implies \mathtt{val}_{\mathsf{W}}(w) = \mathtt{val}_{\mathsf{R}}(\alpha_i)$, $G^{i+1} = \mathsf{Add}(G^i, \alpha_i, w)$, and $K^{i+1} = K^i[\tau \mapsto k \sqcup (\lambda Y.\, \{\alpha_i\})]$. Hence, $\langle G^i, K^i \rangle \xrightarrow{\beta_i}_{\mathsf{pRC11}} \langle G^{i+1}, K^{i+1} \rangle$ follows from the following claims:
  - $w \notin \mathit{dom}(G^i.\mathsf{mo} ; G^i.\mathtt{rf}^{?} ; [k(X)])$: To see this, suppose otherwise. Let $e \in k(X)$ such that $\langle w, e \rangle \in G^i.\mathsf{mo} ; G^i.\mathtt{rf}^{?}$. Then, we have $\langle \alpha_i, e \rangle \in (G^i.\mathsf{mo} \cup G^i.\mathtt{fr}) ; G^i.\mathtt{rf}^{?}$, and so we also have $\langle \alpha_i, e \rangle \in (G.\mathsf{mo} \cup G.\mathtt{fr}) ; G.\mathtt{rf}^{?}$. Let $p = \mathsf{EP}(e, \tau, X)$, and let $k \leq i$ such that $\alpha_k = p$. It follows that $\langle \alpha_i, p \rangle \in R_{\mathsf{mo}} \cup R_{\mathtt{fr}}$. Since the enumeration respects $R_{\mathsf{mo}} \cup R_{\mathtt{fr}}$, this contradicts the fact that $k \leq i$.
  - If $\alpha_i \in \mathsf{W}$, then $w \notin \mathit{dom}(G^i.\mathtt{rf} ; [\mathsf{RMW}])$: To see this, suppose otherwise. Let $u \in \mathsf{RMW}$ such that $\langle w, u \rangle \in G^i.\mathtt{rf}$. Hence, we also have $\langle w, u \rangle \in G.\mathtt{rf}$. However, our construction ensures that $\langle w, \alpha_i \rangle \in G.\mathsf{mo}$ and $\langle \alpha_i, u \rangle \in G.\mathsf{mo}$, which contradicts the fact that G satisfies the RMW2 condition.
  - If $\alpha_i \in \mathsf{R}^{\mathsf{pacq}}$, then $\mathit{dom}(G^i.\mathtt{rf}^{*} ; [w]) \subseteq k(X)$: Suppose that $\alpha_i \in \mathsf{R}^{\mathsf{pacq}}$, and let w' such that $\langle w', w \rangle \in G^i.\mathtt{rf}^{*}$. Then, $\langle w', \alpha_i \rangle \in G.\mathtt{rf}^{+}$. Let $p = \mathsf{EP}(w', \tau, X)$. Then, we have $\langle p, \alpha_i \rangle \in R_{\mathtt{rfp}}$. Since the enumeration respects $R_{\mathtt{rfp}}$, we have $\alpha_k = p$ for some $k < i$. The latter ensures that $w' \in k(X)$.
  - If $\alpha_i \in \mathsf{R}^{\mathsf{acq}}$, then $\forall Y.\, \mathit{dom}(G^i.\mathtt{rf}^{*} ; [w]) \subseteq k(Y)$: Let $Y \in \mathsf{Space}$. Suppose that $\alpha_i \in \mathsf{R}^{\mathsf{acq}}$, and let w' such that $\langle w', w \rangle \in G^i.\mathtt{rf}^{*}$. Then, $\langle w', \alpha_i \rangle \in G.\mathtt{rf}^{+}$. Let $p = \mathsf{EP}(w', \tau, Y)$. Then, we have $\langle p, \alpha_i \rangle \in R_{\mathtt{rf}}$. Since the enumeration respects $R_{\mathtt{rf}}$, we have $\alpha_k = p$ for some $k < i$. The latter ensures that $w' \in k(Y)$.
- Case 2 — $\alpha_i \in G.\mathsf{E}$ and $\mathtt{typ}(\alpha_i) \in \{\mathsf{CALL}, \mathsf{RET}\}$: Let $\tau = \mathtt{tid}(\alpha_i)$ and $k = K^i(\tau)$. Then, we have $\alpha_i = \mathsf{NextEvent}(G^i.\mathsf{E}, \tau, \mathtt{lab}(\alpha_i), \mathtt{method}(\alpha_i))$, $G^{i+1} = \mathsf{Add}(G^i, \alpha_i)$, and $K^{i+1} = K^i[\tau \mapsto k \sqcup (\lambda Y.\, \{\alpha_i\})]$. Hence, $\langle G^i, K^i \rangle \xrightarrow{\beta_i}_{\mathsf{pRC11}} \langle G^{i+1}, K^{i+1} \rangle$.
- Case 3 — $\alpha_i = \mathsf{EP}(e, \tau, X) \in P$: In this case, $\langle G^i, K^i \rangle \xrightarrow{\beta_i}_{\mathsf{pRC11}} \langle G^{i+1}, K^{i+1} \rangle$ follows from the following:
  - $e \in G^i.\mathsf{E} \setminus K^i(\tau)(X)$: Since the enumeration respects R_{prop}, we have that $\alpha_k = e$ for some $k < i$, and so $e \in G^i.\mathsf{E}$. Since $\alpha_i = \mathsf{EP}(e, \tau, X)$, we cannot have $\alpha_j = \mathsf{EP}(e, \tau, X)$ for some $j < i$, and so $e \notin K^i(\tau)(X)$.

– $e \in \mathsf{W}^{\mathtt{prel}}_X \cup \mathsf{W}^{\mathtt{rel}} \cup \mathsf{CR} \implies (\mathsf{E}_X \cup \mathsf{CR}) \cap dom(G^i.\mathtt{hb}_X \,;[e]) \subseteq K^i(\tau)(X)$: Suppose that $e \in \mathsf{W}^{\mathtt{prel}}_X \cup \mathsf{W}^{\mathtt{rel}} \cup \mathsf{CR}$, and let $e' \in \mathsf{E}_X \cup \mathsf{CR}$ such that $\langle e', e\rangle \in G^i.\mathtt{hb}_X$. Let $p' = \mathsf{EP}(e', \tau, X)$. Then, we have $\langle p', \alpha_i\rangle \in T$. Since the enumeration respects T, we have $\alpha_k = p'$ for some $k < i$. The latter ensures that $e' \in K^i(\tau)(X)$. □

A.2 Properties of Program Semantics and Program State Composition

In this appendix we state and prove several claims on program semantics (§3) and the program state composition (Def. 6.7). Henceforth, we let $F \subseteq \mathsf{F}$ be an arbitrary set of methods.

Proposition A.1. *Let L be a library that is safe for programs Pr and Pr'. Suppose that $\overline{p}' \xrightarrow{\alpha}_{Pr'[L]} \overline{p}$ with* $\mathtt{method}(\alpha) \in dom(L)$. *Then, $\overline{p}' \xrightarrow{\alpha}_{Pr[L]} \overline{p}$.*

Proposition A.2. *Let L and L' be libraries with the same domain that are safe for a program Pr. Suppose that $\overline{p}' \xrightarrow{\alpha}_{Pr[L']} \overline{p}$ with* $\mathtt{method}(\alpha) \notin dom(L)$. *Then, $\overline{p}' \xrightarrow{\alpha}_{Pr[L]} \overline{p}$.*

Proposition A.3. *Suppose that $\overline{p}'_{\mathsf{lib}} \xrightarrow{\alpha}_{Pr} \overline{p}_{\mathsf{lib}}$. Let $\overline{p}_{\mathsf{cl}}$ be a program state with $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{f} \notin F$. Then, $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}] = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}]$.*

Proof. Let $\tau = \mathtt{tid}(\alpha)$. We prove the claim by establishing $\forall \pi \in \mathsf{Tid}. \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi) = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\pi)$. Let $\pi \in \mathsf{Tid}$. If $\overline{p}_{\mathsf{cl}}(\pi).\mathsf{f} \notin F$, then $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi) = \overline{p}_{\mathsf{cl}}(\pi) = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\pi)$. Otherwise, if $\overline{p}_{\mathsf{cl}}(\pi).\mathsf{f} \in F$, we have $\pi \neq \tau$, and hence $\overline{p}_{\mathsf{lib}}(\pi) = \overline{p}'_{\mathsf{lib}}(\pi)$. Therefore, we have $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi) = \langle \overline{p}_{\mathsf{lib}}(\pi).\mathsf{pc}, \overline{p}_{\mathsf{lib}}(\pi).\phi, \overline{p}_{\mathsf{cl}}(\pi).\mathsf{pc}_{\mathsf{s}}, \overline{p}_{\mathsf{cl}}(\pi).\mathsf{f}\rangle = \langle \overline{p}'_{\mathsf{lib}}(\pi).\mathsf{pc}, \overline{p}'_{\mathsf{lib}}(\pi).\phi, \overline{p}_{\mathsf{cl}}(\pi).\mathsf{pc}_{\mathsf{s}}, \overline{p}_{\mathsf{cl}}(\pi).\mathsf{f}\rangle = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\pi)$. □

Proposition A.4. *Suppose that $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha}_{Pr} \overline{p}_{\mathsf{cl}}$ with* $\mathtt{method}(\alpha) \in F$ *and* $\mathtt{typ}(\alpha) \neq \mathsf{RET}$. *Let $\overline{p}_{\mathsf{lib}}$ be a program state. Then, $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}] = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$.*

Proof. Let $\tau = \mathtt{tid}(\alpha)$. We prove the claim by establishing $\forall \pi \in \mathsf{Tid}. \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi) = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\pi)$. Let $\pi \in \mathsf{Tid}$. If $\pi \neq \tau$, then $\overline{p}_{\mathsf{cl}}(\pi) = \overline{p}'_{\mathsf{cl}}(\pi)$ and it is easy to see (irrespective of whether $\overline{p}_{\mathsf{cl}}(\pi).\mathsf{f} \in F$ or $\overline{p}_{\mathsf{cl}}(\pi).\mathsf{f} \notin F$) that $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi) = \overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi)$. When $\pi = \tau$, we have $\overline{p}'_{\mathsf{cl}}(\tau).\mathsf{f} \in F$, and since $\overline{p}'_{\mathsf{cl}}(\tau).\mathsf{f} \in F$ and $\mathtt{typ}(\alpha) \neq \mathsf{RET}$, we have $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc}_{\mathsf{s}} = \overline{p}'_{\mathsf{cl}}(\tau).\mathsf{pc}_{\mathsf{s}}$ and $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{f} = \overline{p}'_{\mathsf{cl}}(\tau).\mathsf{f}$. Therefore, $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau) = \langle \overline{p}_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}_{\mathsf{lib}}(\tau).\phi, \overline{p}'_{\mathsf{cl}}(\tau).\mathsf{pc}_{\mathsf{s}}, \overline{p}'_{\mathsf{cl}}(\tau).\mathsf{f}\rangle = \langle \overline{p}_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}_{\mathsf{lib}}(\tau).\phi, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc}_{\mathsf{s}}, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{f}\rangle = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau)$. □

Proposition A.5. *Suppose that $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha}_{Pr} \overline{p}_{\mathsf{cl}}$ with* $\mathtt{method}(\alpha) \notin F$ *and $\alpha \notin \mathsf{Call}_F$. Let $\overline{p}_{\mathsf{lib}}$ be a program state. Then, $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}] \xrightarrow{\alpha}_{Pr} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$.*

Proof. Let $\alpha = \langle \tau, l_\epsilon, f\rangle$. For every $\pi \neq \tau$, we have $\overline{p}'_{\mathsf{cl}}(\pi) = \overline{p}_{\mathsf{cl}}(\pi)$, and so $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi) = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi)$. It remains to show that $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau) \xrightarrow{l_\epsilon, f}_{Pr(\tau)} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau)$. Since $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha}_{Pr} \overline{p}_{\mathsf{cl}}$, we have $\overline{p}'_{\mathsf{cl}}(\tau).\mathsf{f} = f$ and $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{f} \notin F$. Hence, $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau) = \overline{p}'_{\mathsf{cl}}(\tau)$ and $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau) = \overline{p}_{\mathsf{cl}}(\tau)$, and so $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau) \xrightarrow{l_\epsilon, f}_{Pr(\tau)} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau)$. □

Proposition A.6. *Suppose that $\overline{p}'_{\mathsf{lib}} \xrightarrow{\tau, l_\epsilon, f}_{Pr} \overline{p}_{\mathsf{lib}}$ with $f \in F$ and* $\mathtt{typ}(l_\epsilon) \neq \mathsf{RET}$. *Let $\overline{p}_{\mathsf{cl}}$ be a program state with $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{f} = f$. Then, $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}] \xrightarrow{\tau, l_\epsilon, f}_{Pr} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$.*

Proof. For every $\pi \neq \tau$, we have $\overline{p}'_{\mathsf{lib}}(\pi) = \overline{p}_{\mathsf{lib}}(\pi)$, and so $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\pi) = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi)$. It remains to show that $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\tau) \xrightarrow{l_\epsilon, f}_{Pr(\tau)} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau)$. Since $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{f} = f \in F$, we have $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\tau) = \langle \overline{p}'_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}'_{\mathsf{lib}}(\tau).\phi, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc}_{\mathsf{s}}, f\rangle$ and $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau) = \langle \overline{p}_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}_{\mathsf{lib}}(\tau).\phi, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc}_{\mathsf{s}}, f\rangle$. Since $\overline{p}'_{\mathsf{lib}} \xrightarrow{\tau, l_\epsilon, f}_{Pr} \overline{p}_{\mathsf{lib}}$, we have $\overline{p}'_{\mathsf{lib}}(\tau) \xrightarrow{l_\epsilon, f}_{Pr(\tau)} \overline{p}_{\mathsf{lib}}(\tau)$. Since $f \in$

F and $\mathtt{typ}(l_\epsilon) \neq \mathsf{RET}$, it must be the case that $\langle \overline{p}'_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}'_{\mathsf{lib}}(\tau).\phi\rangle \xrightarrow{l_\epsilon}_{Pr(\tau)(f)} \langle \overline{p}_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}_{\mathsf{lib}}(\tau).\phi\rangle$. Then, by definition, we have $\langle \overline{p}'_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}'_{\mathsf{lib}}(\tau).\phi, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc_s}, f\rangle \xrightarrow{l_\epsilon, f}_{Pr(\tau)} \langle \overline{p}_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}_{\mathsf{lib}}(\tau).\phi, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc_s}, f\rangle$. □

Proposition A.7. *Let $f \in F$. Suppose that $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha}_{Pr} \overline{p}_{\mathsf{cl}}$ and $\overline{p}'_{\mathsf{lib}} \xrightarrow{\alpha}_{Pr'} \overline{p}_{\mathsf{lib}}$ with $\alpha = \langle \tau, \mathsf{CALL}(f, \phi), \mathtt{main}\rangle$. Then, $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}] \xrightarrow{\alpha}_{Pr} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$.*

Proof. For every $\pi \neq \tau$, we have $\overline{p}'_{\mathsf{cl}}(\pi) = \overline{p}_{\mathsf{cl}}(\pi)$ and $\overline{p}'_{\mathsf{lib}}(\pi) = \overline{p}_{\mathsf{lib}}(\pi)$, and so $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\pi) = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi)$. It remains to show that $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\tau) \xrightarrow{l, \mathtt{main}}_{Pr(\tau)} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau)$ where $l = \mathsf{CALL}(f, \phi)$. Since $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha}_{Pr} \overline{p}_{\mathsf{cl}}$, we have $Pr(\tau)(\overline{p}'_{\mathsf{cl}}.\mathsf{pc}) = \mathtt{call}(f)$, $\overline{p}'_{\mathsf{cl}}(\tau).\phi = \phi$, $\overline{p}'_{\mathsf{cl}}(\tau).\mathsf{pc_s} = \perp$, $\overline{p}'_{\mathsf{cl}}(\tau).\mathsf{f} = \mathtt{main}$, $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc_s} = \overline{p}'_{\mathsf{cl}}(\tau).\mathsf{pc} + 1$, and $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{f} = f$. Since $\overline{p}'_{\mathsf{lib}} \xrightarrow{\alpha}_{Pr'} \overline{p}_{\mathsf{lib}}$, we have $\overline{p}_{\mathsf{lib}}(\tau).\mathsf{pc} = 0$ and $\overline{p}_{\mathsf{lib}}(\tau).\phi = \phi$. Hence, $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\tau) = \overline{p}'_{\mathsf{cl}}(\tau) = \langle \overline{p}'_{\mathsf{cl}}(\tau).\mathsf{pc}, \phi, \perp, \mathtt{main}\rangle$ and $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau) = \langle 0, \phi, \overline{p}'_{\mathsf{cl}}(\tau).\mathsf{pc} + 1, f\rangle$. By definition, we have $\langle \overline{p}'_{\mathsf{cl}}(\tau).\mathsf{pc}, \phi, \perp, \mathtt{main}\rangle \xrightarrow{l, \mathtt{main}}_{Pr(\tau)} \langle 0, \phi, \overline{p}'_{\mathsf{cl}}(\tau).\mathsf{pc} + 1, f\rangle$. □

Proposition A.8. *Let $f \in F$. Suppose that $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha}_{Pr} \overline{p}_{\mathsf{cl}}$ and $\overline{p}'_{\mathsf{lib}} \xrightarrow{\alpha}_{Pr'} \overline{p}_{\mathsf{lib}}$ with $\alpha = \langle \tau, \mathsf{RET}(\phi), f\rangle$. Then, $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}] \xrightarrow{\alpha}_{Pr'} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$.*

Proof. For every $\pi \neq \tau$, we have $\overline{p}'_{\mathsf{cl}}(\pi) = \overline{p}_{\mathsf{cl}}(\pi)$ and $\overline{p}'_{\mathsf{lib}}(\pi) = \overline{p}_{\mathsf{lib}}(\pi)$, and so $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\pi) = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\pi)$. It remains to show that $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\tau) \xrightarrow{l, f}_{Pr'(\tau)} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau)$ where $l = \mathsf{RET}(\phi)$. Since $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha}_{Pr} \overline{p}_{\mathsf{cl}}$, we have $\overline{p}'_{\mathsf{cl}}(\tau).\mathsf{pc_s} = \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc}$, $\overline{p}'_{\mathsf{cl}}(\tau).\mathsf{f} = f$, $\overline{p}_{\mathsf{cl}}(\tau).\phi = \phi$, $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc_s} = \perp$, and $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{f} = \mathtt{main}$. Since $\overline{p}'_{\mathsf{lib}} \xrightarrow{\alpha}_{Pr'} \overline{p}_{\mathsf{lib}}$, we have $Pr'(\tau)(f)(\overline{p}'_{\mathsf{lib}}(\tau).\mathsf{pc}) = \mathtt{return}$ and $\overline{p}'_{\mathsf{lib}}(\tau).\phi = \phi$. Hence, $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\tau) = \langle \overline{p}'_{\mathsf{lib}}(\tau).\mathsf{pc}, \phi, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc}, f\rangle$ and $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau) = \overline{p}_{\mathsf{cl}}(\tau) = \langle \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc}, \phi, \perp, \mathtt{main}\rangle$. By definition, we have $\langle \overline{p}'_{\mathsf{lib}}(\tau).\mathsf{pc}, \phi, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc}, f\rangle \xrightarrow{l, f}_{Pr'(\tau)} \langle \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc}, \phi, \perp, \mathtt{main}$ □

Proposition A.9. *Suppose that α is enabled in $\overline{p}'_{\mathsf{cl}}$ in Pr. Let $\overline{p}_{\mathsf{lib}}$ be a program state. If $\mathtt{method}(\alpha) \notin F$, then α is enabled in $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$ in Pr.*

Proof. Let $\alpha = \langle \tau, l_\epsilon, f\rangle$. Since α is enabled in $\overline{p}'_{\mathsf{cl}}$ in Pr, we have $\overline{p}'_{\mathsf{cl}}(\tau).\mathsf{f} = f \notin F$. Hence, $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}](\tau) = \overline{p}'_{\mathsf{cl}}(\tau)$, from which the claim follows. □

Proposition A.10. *Suppose that α is enabled in $\overline{p}'_{\mathsf{lib}}$ in Pr. Let $\overline{p}_{\mathsf{cl}}$ be a program state with $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{f} = \mathtt{method}(\alpha) \in F$. Then, α is enabled in $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}]$ in Pr.*

Proof. Let $\alpha = \langle \tau, l_\epsilon, f\rangle$. Since $\overline{p}_{\mathsf{cl}}(\tau).\mathsf{f} \in F$, we have $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}](\tau) = \langle \overline{p}'_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}'_{\mathsf{lib}}(\tau).\phi, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc_s}, f\rangle$. Since α is enabled in $\overline{p}'_{\mathsf{lib}}$ in Pr, $\langle l_\epsilon, f\rangle$ is enabled in $\overline{p}'_{\mathsf{lib}}(\tau)$ in $Pr(\tau)$. Then, either *(i)* l_ϵ is enabled in $\langle \overline{p}'_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}'_{\mathsf{lib}}(\tau).\phi\rangle$ in $Pr(\tau)(f)$; or *(ii)* $l_\epsilon = \mathsf{RET}(\phi)$, $Pr(\tau)(f)(\overline{p}'_{\mathsf{lib}}(\tau).\mathsf{pc}) = \mathtt{return}$ and $\overline{p}'_{\mathsf{lib}}(\tau).\phi = \phi$. In both cases, by definition, α is enabled in $\langle \overline{p}'_{\mathsf{lib}}(\tau).\mathsf{pc}, \overline{p}'_{\mathsf{lib}}(\tau).\phi, \overline{p}_{\mathsf{cl}}(\tau).\mathsf{pc_s}, f\rangle$. □

Proposition A.11. *Suppose that $\langle \overline{p}, M\rangle \xrightarrow{t_1}_{Pr \bowtie \mathsf{pRC11}} \langle \overline{p}_1, M_1\rangle$ and $\langle \overline{p}, M\rangle \xrightarrow{t_2}_{Pr' \bowtie \mathsf{pRC11}} \langle \overline{p}_2, M_2\rangle$. If $\mathsf{H}_F(t_1) = \mathsf{H}_F(t_2)$, then the following hold for every $\tau \in \mathsf{Tid}$:*

- *If $\overline{p}_1(\tau).\mathsf{f} \in F$, then $\overline{p}_2(\tau).\mathsf{f} = \overline{p}_1(\tau).\mathsf{f}$.*
- *If $\overline{p}_1(\tau).\mathsf{f} \notin F$, then $\overline{p}_2(\tau).\mathsf{f} \notin F$.*

A.3 Proofs for Section 6

Proposition A.12. *Let G be an execution graph, $F \subseteq \mathsf{F}$, $\tau \in \mathsf{Tid}$, $l \in \mathsf{Lab}$, and $f \in \mathsf{F}$.*

- *If $f \in F$ or ($\mathtt{typ}(l) = \mathsf{CALL}$ and $\mathtt{callee}(l) \in F$), then $\mathsf{NextEvent}(G|_{\mathsf{E}_F}.\mathsf{E}, \tau, l, f) = \mathsf{NextEvent}(G.\mathsf{E}, \tau, l, f)$.*
- *If $f \notin F$ or $\mathtt{typ}(l) = \mathsf{RET}$, then $\mathsf{NextEvent}(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{E}, \tau, l, f) = \mathsf{NextEvent}(G.\mathsf{E}, \tau, l, f)$.*

Proposition A.13. *Suppose that a set $F \subseteq \mathsf{F}$ is encapsulated in an execution graph G. Then, $G.\mathtt{sw}_{\mathrm{base}} = G|_{\mathsf{E}_F}.\mathtt{sw}_{\mathrm{base}} \cup G|_{\mathsf{E}_{\overline{F}}}.\mathtt{sw}_{\mathrm{base}}$ and $G.\mathtt{sw}_X = G|_{\mathsf{E}_F}.\mathtt{sw}_X \cup G|_{\mathsf{E}_{\overline{F}}}.\mathtt{sw}_X$ for every $X \in \mathsf{Space}$.*

Proposition A.14. *Suppose that $F \subseteq \mathsf{F}$ is encapsulated in a well-formed execution graph G. Then, for every $X \in \mathsf{Space}$:*

- $G.\mathtt{hb}_X \,;[\{e \in \mathsf{E} \mid \mathtt{method}(e) \in F\}] \subseteq (G.\mathtt{hb}_X \,;[\mathsf{CR}_F])^? \,; G|_{\mathsf{E}_F}.\mathtt{hb}_X$.
- $G.\mathtt{hb}_X \,;[\{e \in \mathsf{E} \mid \mathtt{method}(e) \notin F\}] \subseteq (G.\mathtt{hb}_X \,;[\mathsf{CR}_F])^? \,; G|_{\mathsf{E}_{\overline{F}}}.\mathtt{hb}_X$.
- $[\{e \in \mathsf{E} \mid \mathtt{method}(e) \in F\}] \,; G.\mathtt{hb}_X \subseteq G|_{\mathsf{E}_F}.\mathtt{hb}_X \,;([\mathsf{CR}_F] \,; G.\mathtt{hb}_X)^?$.
- $[\{e \in \mathsf{E} \mid \mathtt{method}(e) \notin F\}] \,; G.\mathtt{hb}_X \subseteq G|_{\mathsf{E}_{\overline{F}}}.\mathtt{hb}_X \,;([\mathsf{CR}_F] \,; G.\mathtt{hb}_X)^?$.

Lemma 6.15 (Merge). *Suppose that $F \subseteq \mathsf{F}$ is encapsulated in a well-formed memory state $M = \langle G, K\rangle$. Then, the following hold:*

(1) Let α be an pRC11 transition label with $\mathtt{method}(\alpha) \in F$. Suppose that if $\alpha \in \mathsf{ProgLab}$ and $\mathtt{typ}(\alpha) \in \{\mathtt{W}, \mathtt{R}, \mathtt{RMW}\}$, then $\mathtt{method}(e) \in F$ for every $e \in G.\mathsf{E}$ with $\mathtt{sp}(e) = \mathtt{sp}(\alpha)$. Then, $M|_F \xrightarrow{\alpha}_{\mathtt{pRC11}} M'_F$ implies that $M \xrightarrow{\alpha}_{\mathtt{pRC11}} M'$ for some M' such that $M'|_F = M'_F$.

(2) Let α be an pRC11 transition label with $\mathtt{method}(\alpha) \notin F$. Suppose that if $\alpha \in \mathsf{ProgLab}$ and $\mathtt{typ}(\alpha) \in \{\mathtt{W}, \mathtt{R}, \mathtt{RMW}\}$, then $\mathtt{method}(e) \notin F$ for every $e \in G.\mathsf{E}$ with $\mathtt{sp}(e) = \mathtt{sp}(\alpha)$. Then, $M|_{\overline{F}} \xrightarrow{\alpha}_{\mathtt{pRC11}} M'_{\overline{F}}$ implies that $M \xrightarrow{\alpha}_{\mathtt{pRC11}} M'$ for some M' such that $M'|_{\overline{F}} = M'_{\overline{F}}$.

Proof. We prove each of the claims:

(1) Suppose that $M|_F \xrightarrow{\alpha}_{\mathtt{pRC11}} M'_F$. Let $M'_F = \langle G'_F, K'_F\rangle$. We consider the three possible pRC11-steps:
- $\alpha = \langle \tau, l, f\rangle$ and $\mathtt{typ}(l) \in \{\mathtt{W}, \mathtt{R}, \mathtt{RMW}\}$: Let $e = \mathsf{NextEvent}(G|_{\mathsf{E}_F}.\mathsf{E}, \tau, l, f)$ and $X = \mathtt{sp}(e)$. Then, there exists an event $w \in G|_{\mathsf{E}_F}.\mathsf{W}_{X,\mathtt{loc}(e)}$ such that the following hold:
  - $e \in \mathsf{R} \implies \mathtt{val}_{\mathtt{W}}(w) = \mathtt{val}_{\mathtt{R}}(e)$.
  - $w \notin dom(G|_{\mathsf{E}_F}.\mathtt{mo} \,; G|_{\mathsf{E}_F}.\mathtt{rf}^? \,; [K|_{\mathsf{E}_F}(\tau)(X)])$.
  - $e \in \mathsf{W} \implies w \notin dom(G|_{\mathsf{E}_F}.\mathtt{rf} \,; [\mathsf{RMW}])$.
  - $e \in \mathsf{R}^{\mathtt{pacq}} \implies dom(G|_{\mathsf{E}_F}.\mathtt{rf}^* \,; [w]) \subseteq K|_{\mathsf{E}_F}(\tau)(X)$.
  - $e \in \mathsf{R}^{\mathtt{acq}} \implies \forall Y.\, dom(G|_{\mathsf{E}_F}.\mathtt{rf}^* \,; [w]) \subseteq K|_{\mathsf{E}_F}(\tau)(Y)$.
  - $G'_F = \mathsf{Add}(G|_{\mathsf{E}_F}, e, w)$.
  - $K'_F = K|_{\mathsf{E}_F}[\tau \mapsto k'_F]$ where $k'_F = \lambda Y.\, K|_{\mathsf{E}_F}(\tau)(Y) \cup \{e\}$.

  In addition, the assumption on α ensures that $f \in F$ and $\mathtt{method}(w) \in F$ (since $\mathtt{sp}(e) = \mathtt{sp}(w)$). Let $G' = \mathsf{Add}(G, e, w)$ and $K' = K[\tau \mapsto k']$, where $k' = \lambda Y.\, K(\tau)(Y) \cup \{e\}$. Let $M' = \langle G', K'\rangle$. It is easy to see that $G'|_{\mathsf{E}_F} = G'_F$ (in particular, $e = \mathsf{NextEvent}(G.\mathsf{E}, \tau, l, f)$ by Prop. A.12) and $K'|_{\mathsf{E}_F} = K'_F$. Then, $M \xrightarrow{\alpha}_{\mathtt{pRC11}} M'$ follows from the following:
  - $w \notin dom(G.\mathtt{mo} \,; G.\mathtt{rf}^? \,; [K(\tau)(X)])$: Suppose that $\langle w, a\rangle \in G.\mathtt{mo}$ and $\langle a, b\rangle \in G.\mathtt{rf}^?$ for some $a \in G.\mathsf{E}$ and $b \in K(\tau)(X)$. Then, by definition, we have $\mathtt{sp}(w) = \mathtt{sp}(a) = \mathtt{sp}(b)$. Since F is encapsulated in G and $\mathtt{method}(w) \in F$, we must have $\mathtt{method}(a) \in F$ and $\mathtt{method}(b) \in F$. Hence, we have $\langle w, a\rangle \in G|_{\mathsf{E}_F}.\mathtt{mo}$, $\langle a, b\rangle \in G|_{\mathsf{E}_F}.\mathtt{rf}^?$, and $a \in K|_{\mathsf{E}_F}(\tau)(X)$. This contradicts the fact that $w \notin dom(G|_{\mathsf{E}_F}.\mathtt{mo} \,; G|_{\mathsf{E}_F}.\mathtt{rf}^? \,; [K|_{\mathsf{E}_F}(\tau)(X)])$.
  - $e \in \mathsf{W} \implies w \notin dom(G.\mathtt{rf} \,; [\mathsf{RMW}])$: Suppose that $e \in \mathsf{W}$ but $\langle w, a\rangle \in G.\mathtt{rf}$ for some $a \in \mathsf{RMW}$. Then, by definition, we have $\mathtt{sp}(w) = \mathtt{sp}(a)$. Since F is encapsulated in G and $\mathtt{method}(w) \in F$, we must have $\mathtt{method}(a) \in F$. Hence, we have $\langle w, a\rangle \in G|_{\mathsf{E}_F}.\mathtt{rf}$. This contradicts the fact that $e \in \mathsf{W} \implies w \notin dom(G|_{\mathsf{E}_F}.\mathtt{rf} \,; [\mathsf{RMW}])$.
  - $e \in \mathsf{R}^{\mathtt{pacq}} \implies dom(G.\mathtt{rf}^* \,; [w]) \subseteq K(\tau)(X)$: Suppose that $e \in \mathsf{R}^{\mathtt{pacq}}$ but $\langle a_1, a_2\rangle, \dots, \langle a_{n-1}, a_n\rangle \in G.\mathtt{rf}$ and $\langle a_n, w\rangle \in G.\mathtt{rf}^?$ for some $n \geq 1$ and $a_1, \dots, a_n$ such that $a_1 \notin K(\tau)(X)$. Then, by definition, we have $\mathtt{sp}(a_1) = \dots = \mathtt{sp}(a_n) = \mathtt{sp}(w)$. Since F is encapsulated in G and $\mathtt{method}(w) \in F$, we must have $\mathtt{method}(a_i) \in F$ for every $1 \leq i \leq n$. Hence, we have $\langle a_1, w\rangle \in G|_{\mathsf{E}_F}.\mathtt{rf}^*$. This contradicts the fact that $e \in \mathsf{R}^{\mathtt{pacq}} \implies dom(G|_{\mathsf{E}_F}.\mathtt{rf}^* \,; [w]) \subseteq K|_{\mathsf{E}_F}(\tau)(X)$.

- $e \in \mathsf{R}^{\mathsf{acq}} \implies \forall Y.\, dom(G.\mathsf{rf}^* \,;\, [w]) \subseteq K(\tau)(Y)$: Suppose that $e \in \mathsf{R}^{\mathsf{acq}}$ but there exist $Y \in \mathsf{Space}$ and $a_1 \notin K(\tau)(Y)$ such that $\langle a_1, w\rangle \in G.\mathsf{rf}^*$. Let $n \geq 1$ and $a_2, \ldots, a_n$ such that $\langle a_1, a_2\rangle, \ldots, \langle a_{n-1}, a_n\rangle \in G.\mathsf{rf}$ and $\langle a_n, w\rangle \in G.\mathsf{rf}^?$. Then, by definition, we have $\mathtt{sp}(a_1) = \ldots = \mathtt{sp}(a_n) = \mathtt{sp}(w)$. Since F is encapsulated in G and $\mathtt{method}(w) \in F$, we must have $\mathtt{method}(a_i) \in F$ for every $1 \leq i \leq n$. Hence, we have $\langle a_1, w\rangle \in G|_{\mathsf{E}_F}.\mathsf{rf}^*$. This contradicts the fact that $e \in \mathsf{R}^{\mathsf{acq}} \implies \forall Y.\, dom(G|_{\mathsf{E}_F}.\mathsf{rf}^* \,;\, [w]) \subseteq K|_{\mathsf{E}_F}(\tau)(Y)$.

- $\alpha = \langle \tau, l, f\rangle$ and $\mathtt{typ}(l) \in \{\mathsf{CALL}, \mathsf{RET}\}$: Let $e = \mathsf{NextEvent}(G|_{\mathsf{E}_F}.\mathsf{E}, \tau, l, f)$. Then, we have $G'_F = \mathsf{Add}(G|_{\mathsf{E}_F}, e)$ and $K'_F = K|_{\mathsf{E}_F}[\tau \mapsto k'_F]$, where $k'_F = \lambda Y.\, K|_{\mathsf{E}_F}(\tau)(Y) \cup \{e\}$. Let $G' = \mathsf{Add}(G, e)$ and $K' = K[\tau \mapsto k']$, where $k' = \lambda Y.\, K(\tau)(Y) \cup \{e\}$. Let $M' = \langle G', K'\rangle$. It is easy to see that $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$, $G'|_{\mathsf{E}_F} = G'_F$ (in particular, $e = \mathsf{NextEvent}(G.\mathsf{E}, \tau, l, f)$ by Prop. A.12), and $K'|_{\mathsf{E}_F} = K'_F$.
- $\alpha = \mathsf{EP}(e, \tau, X) \in \mathsf{PLab}$: Then, the following hold:
  - $e \in G|_{\mathsf{E}_F}.\mathsf{E} \setminus K|_{\mathsf{E}_F}(\tau)(X)$.
  - $e \in \mathsf{W}_X^{\mathsf{prel}} \cup \mathsf{W}^{\mathsf{rel}} \cup \mathsf{CR} \implies (\mathsf{E}_X \cup \mathsf{CR}) \cap dom(G|_{\mathsf{E}_F}.\mathsf{hb}_X \,;\, [e]) \subseteq K|_{\mathsf{E}_F}(\tau)(X)$.
  - $G'_F = G|_{\mathsf{E}_F}$ and $K'_F = K|_{\mathsf{E}_F}[\tau \mapsto k'_F]$, where $k'_F = K|_{\mathsf{E}_F}(\tau)[X \mapsto K|_{\mathsf{E}_F}(\tau)(X) \cup \{e\}]$.

  Since $\mathtt{method}(\alpha) \in F$, we clearly have that $e \in G.\mathsf{E} \setminus K(\tau)(X)$. In addition the following holds:
  Claim A.14.1: $e \in \mathsf{W}_X^{\mathsf{prel}} \cup \mathsf{W}^{\mathsf{rel}} \cup \mathsf{CR} \implies (\mathsf{E}_X \cup \mathsf{CR}) \cap dom(G.\mathsf{hb}_X \,;\, [e]) \subseteq K(\tau)(X)$.

  Proof. Suppose that $e \in \mathsf{W}_X^{\mathsf{prel}} \cup \mathsf{W}^{\mathsf{rel}} \cup \mathsf{CR}$ (and so, $(\mathsf{E}_X \cup \mathsf{CR}) \cap dom(G|_{\mathsf{E}_F}.\mathsf{hb}_X \,;\, [e]) \subseteq K|_{\mathsf{E}_F}(\tau)(X)$), and let $e' \in \mathsf{E}_X \cup \mathsf{CR}$ such that $\langle e', e\rangle \in G.\mathsf{hb}_X$. By Prop. A.14, $\langle e', e\rangle \in G.\mathsf{hb}_X$ implies that $\langle e', e\rangle \in (G.\mathsf{hb}_X \,;\, [\mathsf{CR}_F])^? \,;\, G|_{\mathsf{E}_F}.\mathsf{hb}_X$. If $\langle e', e\rangle \in G|_{\mathsf{E}_F}.\mathsf{hb}_X$, then we have $e' \in K|_{\mathsf{E}_F}(\tau)(X)$, and so $e' \in K(\tau)(X)$. Otherwise, let $e'' \in \mathsf{CR}_F$ such that $\langle e', e''\rangle \in G.\mathsf{hb}_X$ and $\langle e'', e\rangle \in G|_{\mathsf{E}_F}.\mathsf{hb}_X$. Then, we have $e'' \in K|_{\mathsf{E}_F}(\tau)(X)$, and so $e'' \in K(\tau)(X)$. Since K is well-formed for G, $e'' \in K(\tau)(X)$ and $\langle e', e''\rangle \in G.\mathsf{hb}_X$ together imply that $e' \in K(\tau)(X)$. ◁

  Now, let $M' = \langle G', K'\rangle$, where $G' = G$ and $K' = K[\tau \mapsto k']$ for $k' = K(\tau)[X \mapsto K(\tau)(X) \cup \{e\}]$. Then, by definition we have $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$, and it is also easy to see that $G'|_{\mathsf{E}_F} = G'_F$ and $K'|_{\mathsf{E}_F} = K'_F$.

(2) Suppose that $M|_{\overline{F}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M'_{\overline{F}}$. Let $M'_{\overline{F}} = \langle G'_{\overline{F}}, K'_{\overline{F}}\rangle$. We consider the three possible pRC11-steps:
- $\alpha = \langle \tau, l, f\rangle$ and $\mathtt{typ}(l) \in \{\mathsf{W}, \mathsf{R}, \mathsf{RMW}\}$: Let $e = \mathsf{NextEvent}(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{E}, \tau, l, f)$ and $X = \mathtt{sp}(e)$. Then, there exists an event $w \in G|_{\mathsf{E}_{\overline{F}}}.\mathsf{W}_{X,\mathtt{loc}(e)}$ such that the following hold:
  - $e \in \mathsf{R} \implies \mathtt{val}_{\mathsf{W}}(w) = \mathtt{val}_{\mathsf{R}}(e)$.
  - $w \notin dom(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{mo} \,;\, G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}^? \,;\, [K|_{\mathsf{E}_{\overline{F}}}(\tau)(X)])$.
  - $e \in \mathsf{W} \implies w \notin dom(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf} \,;\, [\mathsf{RMW}])$.
  - $e \in \mathsf{R}^{\mathsf{pacq}} \implies dom(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}^* \,;\, [w]) \subseteq K|_{\mathsf{E}_{\overline{F}}}(\tau)(X)$.
  - $e \in \mathsf{R}^{\mathsf{acq}} \implies \forall Y.\, dom(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}^* \,;\, [w]) \subseteq K|_{\mathsf{E}_{\overline{F}}}(\tau)(Y)$.
  - $G'_{\overline{F}} = \mathsf{Add}(G|_{\mathsf{E}_{\overline{F}}}, e, w)$.
  - $K'_{\overline{F}} = K|_{\mathsf{E}_{\overline{F}}}[\tau \mapsto k'_{\overline{F}}]$ where $k'_{\overline{F}} = \lambda Y.\, K|_{\mathsf{E}_{\overline{F}}}(\tau)(Y) \cup \{e\}$.

  In addition, the assumption on α ensures that $f \notin F$ and $\mathtt{method}(w) \notin F$ (since $\mathtt{sp}(e) = \mathtt{sp}(w)$). Let $G' = \mathsf{Add}(G, e, w)$ and $K' = K[\tau \mapsto k']$, where $k' = \lambda Y.\, K(\tau)(Y) \cup \{e\}$. Let $M' = \langle G', K'\rangle$. It is easy to see that $G'|_{\mathsf{E}_{\overline{F}}} = G'_{\overline{F}}$ (in particular, $e = \mathsf{NextEvent}(G.\mathsf{E}, \tau, l, f)$ by Prop. A.12) and $K'|_{\mathsf{E}_{\overline{F}}} = K'_{\overline{F}}$. Then, $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$ follows from the following:
  - $w \notin dom(G.\mathsf{mo} \,;\, G.\mathsf{rf}^? \,;\, [K(\tau)(X)])$: Suppose that $\langle w, a\rangle \in G.\mathsf{mo}$ and $\langle a, b\rangle \in G.\mathsf{rf}^?$ for some $a \in G.\mathsf{E}$ and $b \in K(\tau)(X)$. Then, by definition, we have $\mathtt{sp}(w) = \mathtt{sp}(a) = \mathtt{sp}(b)$. Since F is encapsulated in G and $\mathtt{method}(w) \notin F$, we must have $\mathtt{method}(a) \notin F$ and $\mathtt{method}(b) \notin F$. Hence, we have $\langle w, a\rangle \in G|_{\mathsf{E}_{\overline{F}}}.\mathsf{mo}$, $\langle a, b\rangle \in G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}^?$, and $a \in K|_{\mathsf{E}_{\overline{F}}}(\tau)(X)$. This contradicts the fact that $w \notin dom(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{mo} \,;\, G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}^? \,;\, [K|_{\mathsf{E}_{\overline{F}}}(\tau)(X)])$.

- $e \in \mathsf{W} \implies w \notin dom(G.\mathsf{rf}\,;[\mathsf{RMW}])$: Suppose that $e \in \mathsf{W}$ but $\langle w, a\rangle \in G.\mathsf{rf}$ for some $a \in \mathsf{RMW}$. Then, by definition, we have $\mathsf{sp}(w) = \mathsf{sp}(a)$. Since F is encapsulated in G and $\mathtt{method}(w) \notin F$, we must have $\mathtt{method}(a) \notin F$. Hence, we have $\langle w, a\rangle \in G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}$. This contradicts the fact that $e \in \mathsf{W} \implies w \notin dom(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}\,;[\mathsf{RMW}])$.
- $e \in \mathsf{R}^{\mathsf{pacq}} \implies dom(G.\mathsf{rf}^*\,;[w]) \subseteq K(\tau)(X)$: Suppose that $e \in \mathsf{R}^{\mathsf{pacq}}$ but $\langle a_1, a_2\rangle, \dots, \langle a_{n-1}, a_n\rangle \in G.\mathsf{rf}$ and $\langle a_n, w\rangle \in G.\mathsf{rf}^?$ for some $n \geq 1$ and $a_1, \dots, a_n$ such that $a_1 \notin K(\tau)(X)$. Then, by definition, we have $\mathsf{sp}(a_1) = \dots = \mathsf{sp}(a_n) = \mathsf{sp}(w)$. Since F is encapsulated in G and $\mathtt{method}(w) \notin F$, we must have $\mathtt{method}(a_i) \notin F$ for every $1 \leq i \leq n$. Hence, we have $\langle a_1, w\rangle \in G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}^*$. This contradicts the fact that $e \in \mathsf{R}^{\mathsf{pacq}} \implies dom(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}^*\,;[w]) \subseteq K|_{\mathsf{E}_{\overline{F}}}(\tau)(X)$.
- $e \in \mathsf{R}^{\mathsf{acq}} \implies \forall Y.\, dom(G.\mathsf{rf}^*\,;[w]) \subseteq K(\tau)(Y)$: Suppose that $e \in \mathsf{R}^{\mathsf{acq}}$ but there exist $Y \in \mathsf{Space}$ and $a_1 \notin K(\tau)(Y)$ such that $\langle a_1, w\rangle \in G.\mathsf{rf}^*$. Let $n \geq 1$ and $a_2, \dots, a_n$ such that $\langle a_1, a_2\rangle, \dots, \langle a_{n-1}, a_n\rangle \in G.\mathsf{rf}$ and $\langle a_n, w\rangle \in G.\mathsf{rf}^?$. Then, by definition, we have $\mathsf{sp}(a_1) = \dots = \mathsf{sp}(a_n) = \mathsf{sp}(w)$. Since F is encapsulated in G and $\mathtt{method}(w) \notin F$, we must have $\mathtt{method}(a_i) \notin F$ for every $1 \leq i \leq n$. Hence, we have $\langle a_1, w\rangle \in G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}^*$. This contradicts the fact that $e \in \mathsf{R}^{\mathsf{acq}} \implies \forall Y.\, dom(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{rf}^*\,;[w]) \subseteq K|_{\mathsf{E}_{\overline{F}}}(\tau)(Y)$.

- $\alpha = \langle \tau, l, f\rangle$ and $\mathtt{typ}(l) \in \{\mathsf{CALL}, \mathsf{RET}\}$: Let $e = \mathsf{NextEvent}(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{E}, \tau, l, f)$. Then, we have $G'_{\overline{F}} = \mathsf{Add}(G|_{\mathsf{E}_{\overline{F}}}, e)$ and $K'_{\overline{F}} = K|_{\mathsf{E}_{\overline{F}}}[\tau \mapsto k'_{\overline{F}}]$, where $k'_{\overline{F}} = \lambda Y.\, K|_{\mathsf{E}_{\overline{F}}}(\tau)(Y) \cup \{e\}$. Let $G' = \mathsf{Add}(G, e)$ and $K' = K[\tau \mapsto k']$, where $k' = \lambda Y.\, K(\tau)(Y) \cup \{e\}$. Let $M' = \langle G', K'\rangle$. It is easy to see that $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$, $G'|_{\mathsf{E}_{\overline{F}}} = G'_{\overline{F}}$ (in particular, $e = \mathsf{NextEvent}(G.\mathsf{E}, \tau, l, f)$ by Prop. A.12), and $K'|_{\mathsf{E}_{\overline{F}}} = K'_{\overline{F}}$.
- $\alpha = \mathsf{EP}(e, \tau, X) \in \mathsf{PLab}$: Then, the following hold:
  - $e \in G|_{\mathsf{E}_{\overline{F}}}.\mathsf{E} \setminus K|_{\mathsf{E}_{\overline{F}}}(\tau)(X)$.
  - $e \in \mathsf{W}_X^{\mathsf{prel}} \cup \mathsf{W}^{\mathsf{rel}} \cup \mathsf{CR} \implies (\mathsf{E}_X \cup \mathsf{CR}) \cap dom(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X\,;[e]) \subseteq K|_{\mathsf{E}_{\overline{F}}}(\tau)(X)$.
  - $G'_{\overline{F}} = G|_{\mathsf{E}_{\overline{F}}}$ and $K'_{\overline{F}} = K|_{\mathsf{E}_{\overline{F}}}[\tau \mapsto k'_{\overline{F}}]$, where $k'_{\overline{F}} = K|_{\mathsf{E}_{\overline{F}}}(\tau)[X \mapsto K|_{\mathsf{E}_{\overline{F}}}(\tau)(X) \cup \{e\}]$.

  Since $\mathtt{method}(\alpha) \notin F$, we clearly have that $e \in G.\mathsf{E} \setminus K(\tau)(X)$. In addition the following holds:
  Claim A.14.2: $e \in \mathsf{W}_X^{\mathsf{prel}} \cup \mathsf{W}^{\mathsf{rel}} \cup \mathsf{CR} \implies (\mathsf{E}_X \cup \mathsf{CR}) \cap dom(G.\mathsf{hb}_X\,;[e]) \subseteq K(\tau)(X)$.

  Proof. Suppose that $e \in \mathsf{W}_X^{\mathsf{prel}} \cup \mathsf{W}^{\mathsf{rel}} \cup \mathsf{CR}$ (and so, $(\mathsf{E}_X \cup \mathsf{CR}) \cap dom(G|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X\,;[e]) \subseteq K|_{\mathsf{E}_{\overline{F}}}(\tau)(X)$), and let $e' \in \mathsf{E}_X \cup \mathsf{CR}$ such that $\langle e', e\rangle \in G.\mathsf{hb}_X$. By Prop. A.14, $\langle e', e\rangle \in G.\mathsf{hb}_X$ implies that $\langle e', e\rangle \in (G.\mathsf{hb}_X\,;[\mathsf{CR}_F])^?\,; G|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X$. If $\langle e', e\rangle \in G|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X$, then we have $e' \in K|_{\mathsf{E}_{\overline{F}}}(\tau)(X)$, and so $e' \in K(\tau)(X)$. Otherwise, let $e'' \in \mathsf{CR}_F$ such that $\langle e', e''\rangle \in G.\mathsf{hb}_X$ and $\langle e'', e\rangle \in G|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X$. Then, we have $e'' \in K|_{\mathsf{E}_{\overline{F}}}(\tau)(X)$, and so $e'' \in K(\tau)(X)$. Since K is well-formed for G, $e'' \in K(\tau)(X)$ and $\langle e', e''\rangle \in G.\mathsf{hb}_X$ together imply that $e' \in K(\tau)(X)$. ◁

  Now, let $M' = \langle G', K'\rangle$, where $G' = G$ and $K' = K[\tau \mapsto k']$ for $k' = K(\tau)[X \mapsto K(\tau)(X) \cup \{e\}]$. Then, by definition we have $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$, and it is also easy to see that $G'|_{\mathsf{E}_{\overline{F}}} = G'_{\overline{F}}$ and $K'|_{\mathsf{E}_{\overline{F}}} = K'_{\overline{F}}$. □

Proposition A.15 (Determinism). *If ($\alpha \in \mathsf{ProgLab}$ and $\mathtt{typ}(\mathtt{lab}(\alpha)) \in \{\mathsf{CALL}, \mathsf{RET}\}$) or $\alpha \in \mathsf{PLab}$, then $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$ and $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M''$ imply that $M' = M''$.*

Proposition A.16. *Suppose that $F \subseteq \mathsf{F}$ is encapsulated in a well-formed memory state M. If $M|_F \xrightarrow{\alpha}_{\mathsf{pRC11}} M'_F$ and $M|_{\overline{F}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M'_{\overline{F}}$ for $\alpha \in \mathsf{HLab}_F$, then $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$ for some M' such that $M'|_F = M'_F$ and $M'|_{\overline{F}} = M'_{\overline{F}}$.*

Proof. Suppose first that $\mathtt{method}(\alpha) \in F$. Then, by Lemma 6.15 (Item 1), we have $M \xrightarrow{\alpha}_{\mathsf{pRC11}} M'$ for some M'. By Lemma 6.13, we have $M|_F \xrightarrow{\alpha}_{\mathsf{pRC11}} M'|_F$ and $M|_{\overline{F}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M'|_{\overline{F}}$. By Prop. A.15, it follows that $M'|_F = M'_F$ and $M'|_{\overline{F}} = M'_{\overline{F}}$. The case that $\mathtt{method}(\alpha) \notin F$ is similar using Item 2 in Lemma 6.15. ◻

A.4 Proof for Section 7

Lemma 7.5 (Composition). *Let L and L' be libraries implementing the same set F of methods such that both are safe for a program Pr, and L is also safe for a program Pr'. Suppose that $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t_{\mathsf{cl}}}_{Pr[L']\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}, M_{\mathsf{cl}}\rangle$ and $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t_{\mathsf{lib}}}_{Pr'[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{lib}}, M_{\mathsf{lib}}\rangle$, with $\mathsf{H}_F(t_{\mathsf{cl}}) = \mathsf{H}_F(t_{\mathsf{lib}})$. Then, $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M\rangle$ for some trace t and memory state M such that $t|_{\overline{F}} = t_{\mathsf{cl}}|_{\overline{F}}$, $t|_F = t_{\mathsf{lib}}|_F$, $M|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$, and $M|_F = M_{\mathsf{lib}}|_F$.*

Proof. We prove the claim by induction on the sum of lengths of t_{cl} and t_{lib}. In the base case, $|t_{\mathsf{cl}}| + |t_{\mathsf{lib}}| = 0$, and we can simply take $t = \epsilon$ and $M = M_{\mathsf{Init}}$. Suppose now that the claim holds for every t'_{cl} and t'_{lib} with $|t'_{\mathsf{cl}}| + |t'_{\mathsf{lib}}| < |t_{\mathsf{cl}}| + |t_{\mathsf{lib}}|$. We split the rest of the proof into the following three cases (which exhaust all the possibilities for t_{cl} and t_{lib}):

(1) $t_{\mathsf{cl}} = t'_{\mathsf{cl}} \cdot \alpha_{\mathsf{cl}}$ for $\alpha_{\mathsf{cl}} \notin \mathsf{HLab}_F$: Consider a state $\langle \overline{p}'_{\mathsf{cl}}, M'_{\mathsf{cl}}\rangle$ for which the following holds:

$$\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t'_{\mathsf{cl}}}_{Pr[L']\bowtie\mathsf{pRC11}} \langle \overline{p}'_{\mathsf{cl}}, M'_{\mathsf{cl}}\rangle \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L']\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}, M_{\mathsf{cl}}\rangle$$

By the induction hypothesis, $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M'\rangle$ for some t' and M', such that $t'|_{\overline{F}} = t'_{\mathsf{cl}}|_{\overline{F}}$, $t'|_F = t_{\mathsf{lib}}|_F$, $M'|_{\overline{F}} = M'_{\mathsf{cl}}|_{\overline{F}}$, and $M'|_F = M_{\mathsf{lib}}|_F$. We consider two cases:

(a) $\mathtt{method}(\alpha_{\mathsf{cl}}) \in F$: Let $t \triangleq t'$ and $M \triangleq M'$. Then, the claim follows from the following:
- $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}] = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$: If $\alpha_{\mathsf{cl}} \in \mathsf{PLab}$ (i.e., memory internal step), then $\overline{p}'_{\mathsf{cl}} = \overline{p}_{\mathsf{cl}}$, and we are done. Otherwise, $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L']} \overline{p}_{\mathsf{cl}}$, and the claim follows by Prop. A.4.
- $M|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$: Since $M = M'$ and $M'|_{\overline{F}} = M'_{\mathsf{cl}}|_{\overline{F}}$, it suffices to show that $M'_{\mathsf{cl}}|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$. If $\mathtt{lab}(\alpha_{\mathsf{cl}}) = \epsilon$ (i.e., program internal step), then $M'_{\mathsf{cl}} = M_{\mathsf{cl}}$, and we are done. Otherwise, $M'_{\mathsf{cl}} \xrightarrow{\alpha_{\mathsf{cl}}}_{\mathsf{pRC11}} M_{\mathsf{cl}}$, and by Lemma 6.13 we have $M'_{\mathsf{cl}}|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$.

(b) $\mathtt{method}(\alpha_{\mathsf{cl}}) \notin F$: Let $t \triangleq t' \cdot \alpha_{\mathsf{cl}}$. It is easy to see that $t|_{\overline{F}} = t_{\mathsf{cl}}|_{\overline{F}}$ and $t|_F = t_{\mathsf{lib}}|_F$. It suffices to show that $\langle \overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M'\rangle \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M\rangle$ for some M such that $M|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$ and $M|_F = M_{\mathsf{lib}}|_F$. Consider the following cases:
- When $\mathtt{lab}(\alpha_{\mathsf{cl}}) = \epsilon$ (i.e., program internal step), we have $M'_{\mathsf{cl}} = M_{\mathsf{cl}}$ and $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L']} \overline{p}_{\mathsf{cl}}$. Let $M \triangleq M'$. Then, we clearly have $M|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$ and $M|_F = M_{\mathsf{lib}}|_F$. Since $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L']} \overline{p}_{\mathsf{cl}}$, using Propositions A.2 and A.5 we have $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}] \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L]} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$. Hence, we also have $\langle \overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M'\rangle \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M\rangle$.
- Otherwise, we have $M'_{\mathsf{cl}} \xrightarrow{\alpha_{\mathsf{cl}}}_{\mathsf{pRC11}} M_{\mathsf{cl}}$. By Lemma 6.13, we have $M'_{\mathsf{cl}}|_{\overline{F}} \xrightarrow{\alpha_{\mathsf{cl}}}_{\mathsf{pRC11}} M_{\mathsf{cl}}|_{\overline{F}}$. Since $M'|_{\overline{F}} = M'_{\mathsf{cl}}|_{\overline{F}}$, we have $M'|_{\overline{F}} \xrightarrow{\alpha_{\mathsf{cl}}}_{\mathsf{pRC11}} M_{\mathsf{cl}}|_{\overline{F}}$. In addition, since $M'|_{\overline{F}} \xrightarrow{\alpha_{\mathsf{cl}}}_{\mathsf{pRC11}} M_{\mathsf{cl}}|_{\overline{F}}$, using Lemma 6.15-Item 2 (and establishing its precondition using Prop. 6.2), we have $M' \xrightarrow{\alpha_{\mathsf{cl}}}_{\mathsf{pRC11}} M$ for some M such that $M|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$. By Lemma 6.14, $M' \xrightarrow{\alpha_{\mathsf{cl}}}_{\mathsf{pRC11}} M$ implies that $M|_F = M'|_F$, and so we also have $M|_F = M_{\mathsf{lib}}|_F$. To show that $\langle \overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M'\rangle \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M\rangle$, consider each case:
  - $\alpha_{\mathsf{cl}} \in \mathsf{PLab}$ (i.e., memory internal step): In this case $\overline{p}'_{\mathsf{cl}} = \overline{p}_{\mathsf{cl}}$, and the claim easily follows.
  - Otherwise, we have $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L']} \overline{p}_{\mathsf{cl}}$. By Propositions A.2 and A.5 we have $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}] \xrightarrow{\alpha_{\mathsf{cl}}}_{Pr[L]} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$, from which the claim follows.

(2) $t_{\mathsf{lib}} = t'_{\mathsf{lib}} \cdot \alpha_{\mathsf{lib}}$ for $\alpha_{\mathsf{lib}} \notin \mathsf{HLab}_F$: Consider a state $\langle \overline{p}'_{\mathsf{lib}}, M'_{\mathsf{lib}}\rangle$ for which the following holds:

$$\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}}\rangle \xrightarrow{t'_{\mathsf{lib}}}_{Pr'[L]\bowtie\mathsf{pRC11}} \langle \overline{p}'_{\mathsf{lib}}, M'_{\mathsf{lib}}\rangle \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr'[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{lib}}, M_{\mathsf{lib}}\rangle$$

By the induction hypothesis, $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}], M' \rangle$ for some t' and M', such that $t'|_{\overline{F}} = t_{\mathsf{cl}}|_{\overline{F}}$, $t'|_F = t'_{\mathsf{lib}}|_F$, $M'|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$, and $M'|_F = M'_{\mathsf{lib}}|_F$. We consider two cases:

(a) $\mathtt{method}(\alpha_{\mathsf{lib}}) \notin F$: Let $t \triangleq t'$ and $M \triangleq M'$. Then, the claim follows from the following:
   - $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}] = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$: If $\alpha_{\mathsf{lib}} \in \mathsf{PLab}$ (i.e., memory internal step), then $\overline{p}'_{\mathsf{lib}} = \overline{p}_{\mathsf{lib}}$, and we are done. Otherwise, $\overline{p}'_{\mathsf{lib}} \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr'[L]} \overline{p}_{\mathsf{lib}}$. Let $\tau = \mathtt{tid}(\alpha)$. Since $\mathtt{method}(\alpha_{\mathsf{lib}}) \notin F$, we have $\overline{p}'_{\mathsf{lib}}(\tau) \notin F$. By Prop. A.11, we also have $\overline{p}_{\mathsf{cl}}(\tau) \notin F$. Hence, the claim follows by Prop. A.3.
   - $M|_F = M_{\mathsf{lib}}|_F$: Since $M = M'$ and $M'|_F = M'_{\mathsf{lib}}|_F$, it suffices to show that $M'_{\mathsf{lib}}|_F = M_{\mathsf{lib}}|_F$. If $\mathtt{lab}(\alpha_{\mathsf{lib}}) = \epsilon$ (i.e., program internal step), then $M'_{\mathsf{lib}} = M_{\mathsf{lib}}$, and we are done. Otherwise, $M'_{\mathsf{lib}} \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{pRC11}} M_{\mathsf{lib}}$, and by Lemma 6.13 we have $M'_{\mathsf{lib}}|_F = M_{\mathsf{lib}}|_F$.

(b) $\mathtt{method}(\alpha_{\mathsf{lib}}) \in F$: Let $t \triangleq t' \cdot \alpha_{\mathsf{lib}}$. It is easy to see that $t|_{\overline{F}} = t_{\mathsf{cl}}|_{\overline{F}}$ and $t|_F = t_{\mathsf{lib}}|_F$. It suffices to show that $\langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}], M' \rangle \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M \rangle$ for some M such that $M|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$ and $M|_F = M_{\mathsf{lib}}|_F$, Consider the following cases:
   - When $\mathtt{lab}(\alpha_{\mathsf{lib}}) = \epsilon$ (i.e., program internal step), we have $M'_{\mathsf{lib}} = M_{\mathsf{lib}}$ and $\overline{p}'_{\mathsf{lib}} \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr'[L]} \overline{p}_{\mathsf{lib}}$. Let $M \triangleq M'$. Then, we clearly have $M|_{\overline{F}} = M_{\mathsf{cl}}|_F$ and $M|_F = M_{\mathsf{lib}}|_F$. Since $\overline{p}'_{\mathsf{lib}} \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr'[L]} \overline{p}_{\mathsf{lib}}$, using Propositions A.1, A.6 and A.11 we have $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}] \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr[L]} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$. Hence, we also have $\langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}], M' \rangle \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M \rangle$.
   - Otherwise, we have $M'_{\mathsf{lib}} \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{pRC11}} M_{\mathsf{lib}}$. By Lemma 6.13, we have $M'_{\mathsf{lib}}|_F \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{pRC11}} M_{\mathsf{lib}}|_F$. Since $M'|_F = M'_{\mathsf{lib}}|_F$, we have $M'|_F \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{pRC11}} M_{\mathsf{lib}}|_F$. In addition, since $M'|_F \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{pRC11}} M_{\mathsf{lib}}|_F$, using Lemma 6.15-Item 1 (and establishing its precondition using Prop. 6.2), we have $M' \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{pRC11}} M$, for some M such that $M|_F = M_{\mathsf{lib}}|_F$. By Lemma 6.14, $M' \xrightarrow{\alpha_{\mathsf{lib}}}_{\mathsf{pRC11}} M$ implies that $M|_{\overline{F}} = M'|_{\overline{F}}$, and so we also have $M|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$. To show that $\langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}], M' \rangle \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M \rangle$, consider each case:
     - $\alpha_{\mathsf{cl}} \in \mathsf{PLab}$ (i.e., memory internal step): In this case $\overline{p}'_{\mathsf{lib}} = \overline{p}_{\mathsf{lib}}$, and the claim easily follows.
     - Otherwise, we have $\overline{p}'_{\mathsf{lib}} \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr'[L]} \overline{p}_{\mathsf{lib}}$. By Propositions A.1, A.6 and A.11 we have $\overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}] \xrightarrow{\alpha_{\mathsf{lib}}}_{Pr[L]} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$, from which the claim follows.

(3) $t_{\mathsf{cl}} = t'_{\mathsf{cl}} \cdot \alpha_{\mathsf{cl}}$ for $\alpha_{\mathsf{cl}} \in \mathsf{HLab}_F$ and $t_{\mathsf{lib}} = t'_{\mathsf{lib}} \cdot \alpha_{\mathsf{lib}}$ for $\alpha_{\mathsf{lib}} \in \mathsf{HLab}_F$: In this case we have $\alpha_{\mathsf{cl}} = \alpha_{\mathsf{lib}}$. Let $\alpha = \alpha_{\mathsf{cl}}$, and let $\langle \overline{p}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle$ and $\langle \overline{p}'_{\mathsf{lib}}, M'_{\mathsf{lib}} \rangle$ be states for which the following hold:

$$\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'_{\mathsf{cl}}}_{Pr[L']\bowtie\mathsf{pRC11}} \langle \overline{p}'_{\mathsf{cl}}, M'_{\mathsf{cl}} \rangle \xrightarrow{\alpha}_{Pr[L']\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}, M_{\mathsf{cl}} \rangle$$

$$\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'_{\mathsf{lib}}}_{Pr'[L]\bowtie\mathsf{pRC11}} \langle \overline{p}'_{\mathsf{lib}}, M'_{\mathsf{lib}} \rangle \xrightarrow{\alpha}_{Pr'[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{lib}}, M_{\mathsf{lib}} \rangle$$

By the induction hypothesis, $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}], M' \rangle$ for some t' and M', such that $t'|_{\overline{F}} = t'_{\mathsf{cl}}|_{\overline{F}}$, $t'|_F = t'_{\mathsf{lib}}|_F$, $M'|_{\overline{F}} = M'_{\mathsf{cl}}|_{\overline{F}}$, and $M'|_F = M'_{\mathsf{lib}}|_F$. Let $t \triangleq t' \cdot \alpha$. It is easy to see that $t|_{\overline{F}} = t_{\mathsf{cl}}|_{\overline{F}}$ and $t|_F = t_{\mathsf{lib}}|_F$. Since $\alpha \in \mathsf{HLab}_F$, we have $\mathtt{lab}(\alpha) \neq \epsilon$ and so $M'_{\mathsf{cl}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M_{\mathsf{cl}}$ and $M'_{\mathsf{lib}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M_{\mathsf{lib}}$. Now, since $M'_{\mathsf{cl}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M_{\mathsf{cl}}$, using Lemma 6.13, we have $M'_{\mathsf{cl}}|_{\overline{F}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M_{\mathsf{cl}}|_{\overline{F}}$, and so $M'|_{\overline{F}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M_{\mathsf{cl}}|_{\overline{F}}$. Similarly, since $M'_{\mathsf{lib}} \xrightarrow{\alpha}_{\mathsf{pRC11}} M_{\mathsf{lib}}$, using Lemma 6.13, we have $M'_{\mathsf{lib}}|_F \xrightarrow{\alpha}_{\mathsf{pRC11}} M_{\mathsf{lib}}|_F$, and so $M'|_F \xrightarrow{\alpha}_{\mathsf{pRC11}} M_{\mathsf{lib}}|_F$. Therefore, using Prop. A.16, we obtain that $M' \xrightarrow{\alpha}_{\mathsf{pRC11}} M$ for some M such that $M|_{\overline{F}} = M_{\mathsf{cl}}|_{\overline{F}}$ and $M|_F = M_{\mathsf{lib}}|_F$. To show that $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M \rangle$, it remains to show that $\langle \overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}], M' \rangle \xrightarrow{\alpha}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}], M \rangle$. If $\alpha \in \mathsf{PLab}$, then $\overline{p}'_{\mathsf{lib}} = \overline{p}_{\mathsf{lib}}$ and $\overline{p}'_{\mathsf{cl}} = \overline{p}_{\mathsf{cl}}$, and so $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}] = \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$ and we are done using the fact that $M' \xrightarrow{\alpha}_{\mathsf{pRC11}} M$. Otherwise, we have $\overline{p}'_{\mathsf{cl}} \xrightarrow{\alpha}_{Pr[L']} \overline{p}_{\mathsf{cl}}$ and

$\overline{p}'_{\mathsf{lib}} \xrightarrow{\alpha}_{Pr'[L]} \overline{p}_{\mathsf{lib}}$. By Propositions A.1, A.2, A.7 and A.8, we obtain that $\overline{p}'_{\mathsf{cl}}[F \mapsto \overline{p}'_{\mathsf{lib}}] \xrightarrow{\alpha}_{Pr[L]} \overline{p}_{\mathsf{cl}}[F \mapsto \overline{p}_{\mathsf{lib}}]$, from which the claim follows. □

Lemma 7.6. *Under the conditions of Thm. 7.3,* $\mathsf{H}_F(Pr[L]) \subseteq \mathsf{H}_F(MGC[L^\#])$.

Proof. Assume otherwise, and let h be a shortest history in $\mathsf{H}_F(Pr[L]) \setminus \mathsf{H}_F(MGC[L^\#])$. Let t be a shortest trace of $Pr[L]\bowtie\mathsf{pRC11}$ with $\mathsf{H}_F(t) = h$. Since ε (the empty history) is clearly in $\mathsf{H}_F(MGC[L^\#])$, we know that t is non-empty. Consider the last transition label α in t, and let t' such that $t = t' \cdot \alpha$. The minimality of t ensures that α must be an element of HLab_F (i.e., call, return, call propagation or return propagation of some method in F). The minimality of h further ensures that $\mathsf{H}_F(t') \in \mathsf{H}_F(MGC[L^\#])$. Let $t'_\#$ and $\langle \overline{p}'_\#, M'_\# \rangle$ such that $\mathsf{H}_F(t'_\#) = \mathsf{H}_F(t')$ and $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'_\#}_{MGC[L^\#]\bowtie\mathsf{pRC11}} \langle \overline{p}'_\#, M'_\# \rangle$. Let $\langle \overline{p}', M' \rangle$ be such that $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t'}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}', M' \rangle \xrightarrow{\alpha}_{Pr[L]\bowtie\mathsf{pRC11}} \langle \overline{p}, M \rangle$. We consider the following cases:

(1) $\alpha \in \mathsf{Call}_F \cup \mathsf{CP}_F$: Using Lemma 7.5 (applied with $L := L^\#$, $L' := L$, $Pr := Pr$, and $Pr' := MGC$), there exist $t''_\#$ and $M''_\#$ such that $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t''_\#}_{Pr[L^\#]\bowtie\mathsf{pRC11}} \langle \overline{p}'[F \mapsto \overline{p}'_\#], M''_\# \rangle$, $\mathsf{H}_F(t''_\#) = \mathsf{H}_F(t')$, and $M''_\#|_{\overline{F}} = M'|_{\overline{F}}$.
<u>Claim A.16.1:</u> α is enabled in $M''_\#$.
Proof. If $\alpha \in \mathsf{Call}_F$, then the claim is trivial. Otherwise, since α is enabled in M', by Lemma 6.13, α is also enabled in $M'|_{\overline{F}} = M''_\#|_{\overline{F}}$. By Lemma 6.15 (Item 2), α is also enabled in $M''_\#$. ◁
<u>Claim A.16.2:</u> If $\alpha \in \mathsf{Call}_F$, then α is enabled in $\overline{p}'[F \mapsto \overline{p}'_\#]$ (in the LTS $Pr[L^\#]$).
Proof. Since $\overline{p}' \xrightarrow{\alpha}_{Pr[L]} \overline{p}$, the claim follows from Propositions A.2 and A.9. ◁
It follows that $h = \mathsf{H}_F(t') \cdot \alpha = \mathsf{H}_F(t''_\#) \cdot \alpha \in \mathsf{H}_F(Pr[L^\#])$. Since $Pr \sqsubseteq_{L^\#} MGC$, we obtain $h \in \mathsf{H}_F(MGC[L^\#])$, which contradicts our assumption.

(2) $\alpha \in \mathsf{Ret}_F \cup \mathsf{RP}_F$: Using Lemma 7.5 (applied with $L := L$, $L' := L^\#$, $Pr := MGC$, and $Pr' := Pr$), there exist t'' and M'' such that $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t''}_{MGC[L]\bowtie\mathsf{pRC11}} \langle \overline{p}'_\#[F \mapsto \overline{p}'], M'' \rangle$, $\mathsf{H}_F(t'') = \mathsf{H}_F(t')$, and $M''|_F = M'|_F$.
<u>Claim A.16.3:</u> α is enabled in M''.
Proof. If $\alpha \in \mathsf{Ret}_F$, then the claim is trivial. Otherwise, since α is enabled in M', by Lemma 6.13, α is also enabled in $M'|_F = M''|_F$. By Lemma 6.15 (Item 1), α is also enabled in M''. ◁
<u>Claim A.16.4:</u> If $\alpha \in \mathsf{Ret}_F$, then α is enabled in $\overline{p}'_\#[F \mapsto \overline{p}']$ (in the LTS $MGC[L]$).
Proof. Let $\tau = \mathsf{Tid}(\alpha)$. Since $\alpha \in \mathsf{Ret}_F$, we have $\mathtt{method}(\alpha) \in F$, and since $\overline{p}' \xrightarrow{\alpha}_{Pr[L]} \overline{p}$, we also have $\overline{p}'(\tau).\mathtt{f} = \mathtt{method}(\alpha)$. Since $\mathsf{H}_F(t') = \mathsf{H}_F(t'_\#)$, by Prop. A.11, we have $\overline{p}'_\#(\tau).\mathtt{f} = \mathtt{method}(\alpha)$ as well. Thus, since $\overline{p}' \xrightarrow{\alpha}_{Pr[L]} \overline{p}$, the claim follows from Propositions A.1 and A.10. ◁
It follows that $h = \mathsf{H}_F(t') \cdot \alpha = \mathsf{H}_F(t'') \cdot \alpha \in \mathsf{H}_F(MGC[L])$. Since $L \sqsubseteq_{MGC} L^\#$, we obtain $h \in \mathsf{H}_F(MGC[L^\#])$, which contradicts our assumption. □

Lemma 7.7. *Suppose that* $M_{\mathsf{Init}} \xrightarrow{t}_{\mathsf{pRC11}} \langle G, K \rangle$. *Then, there exist a trace t' and a knowledge mapping K' for G such that the following hold:*

- $M_{\mathsf{Init}} \xrightarrow{t'}_{\mathsf{pRC11}} \langle G, K' \rangle$.
- $t'|_{\mathsf{ProgLab}} = t|_{\mathsf{ProgLab}}$.
- $K'(\tau, X) \cap (\mathsf{W}^{\mathsf{prel}}_X \cup \mathsf{W}^{\mathsf{rel}} \cup \mathsf{CR}) \subseteq dom(G.\mathsf{hb}^?_X \,;\, [\mathsf{E}^\tau])$ *for every* $\tau \in \mathsf{Tid}$ *and* $X \in \mathsf{Space}$.

Proof (sketch). Let T be the set of all traces t_0 such that (1) $M_{\mathsf{Init}} \xrightarrow{t_0}_{\mathsf{pRC11}} \langle G, K_0 \rangle$ for some knowledge mapping K_0 for G; and (2) $t_0|_{\mathsf{ProgLab}} = t|_{\mathsf{ProgLab}}$. Then, $t \in T$, and so T is not empty. Let t' be a trace in T of minimal length, and let K' such that $M_{\mathsf{Init}} \xrightarrow{t'}_{\mathsf{pRC11}} \langle G, K' \rangle$. Suppose (by way of contradiction) that $K'(\tau, X) \cap (\mathsf{W}^{\mathsf{prel}}_X \cup \mathsf{W}^{\mathsf{rel}} \cup \mathsf{CR}) \not\subseteq dom(G.\mathsf{hb}^?_X \,;\, [\mathsf{E}^\tau])$ for some $\tau \in \mathsf{Tid}$ and

$X \in \text{Space}$. Let t'' be the trace obtained from t' by omitting the last transition label in t' of the form $\mathsf{EP}(e, \tau, X)$ with $e \in \mathsf{W}^{\mathtt{prel}}_X \cup \mathsf{W}^{\mathtt{rel}} \cup \mathsf{CR}$ and $e \notin dom(G.\mathsf{hb}^?_X \,;\, [\mathsf{E}^\tau])$. Then, it can be shown that $t'' \in T$, which contradicts the minimality of t'. ◻

Theorem 7.3 (Library Abstraction). *Let L and $L^\#$ be libraries implementing the same set F of methods. Let MGC and Pr be programs, such that both L and $L^\#$ are safe for both MGC and Pr. Suppose that $L \sqsubseteq_{MGC} L^\#$ and $Pr \sqsubseteq_{L^\#} MGC$. Then, the following hold:*

- *If $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t}_{Pr[L] \bowtie \mathsf{pRC11}} \langle \overline{p}, M \rangle$, then there exist $t^\#$ and $\langle \overline{p}^\#, M^\# \rangle$ such that the following hold: (1) $\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t^\#}_{Pr[L^\#] \bowtie \mathsf{pRC11}} \langle \overline{p}^\#, M^\# \rangle$; (2) $t^\#|_{\overline{F}} = t|_{\overline{F}}$; (3) $\overline{p}^\# = \overline{p}[F \mapsto \overline{p}_{\mathsf{lib}}]$ for some $\overline{p}_{\mathsf{lib}}$ (in particular, $\overline{p}^\#(\tau) = \overline{p}(\tau)$ whenever $\overline{p}(\tau).\mathsf{f} \notin F$); and (4) $M^\#|_{\overline{F}} = M|_{\overline{F}}$.*
- *$Pr[L]$ is not racy under* pRC11.

Proof. By Lemma 7.6, we have $\mathsf{H}_F(Pr[L]) \subseteq \mathsf{H}_F(MGC[L^\#])$. Then, the first part of the claim directly follows using Lemma 7.5 by letting $L := L^\#$, $L' := L$, $Pr := Pr$, and $Pr' := MGC$.

For the second part, suppose that $Pr[L]$ is racy under pRC11. Let $\langle \overline{p}, \langle G, K \rangle \rangle$ be a reachable state in $Pr[L] \bowtie \mathsf{pRC11}$ such that G is racy. Let e_1, e_2 be two events that form a race in G. Let $E = dom((G.\mathsf{po} \cup G.\mathsf{rf})^* \,;\, [e_1, e_2])$ and $G' = G|_E$. Clearly, e_1 and e_2 form a race in G'. It is also easy to see that there exist $\overline{p}'$ and K' such that $\langle \overline{p}', \langle G', K' \rangle \rangle$ is reachable in $Pr[L] \bowtie \mathsf{pRC11}$. By Lemma 7.7, there exists a knowledge mapping K'' for G' such that $\langle \overline{p}', \langle G', K'' \rangle \rangle$ is reachable in $Pr[L] \bowtie \mathsf{pRC11}$ as well and $K''(\tau, X) \cap \mathsf{CR} \subseteq dom(G'.\mathsf{hb}^?_X \,;\, [\mathsf{E}^\tau])$ for every $\tau \in \mathsf{Tid}$ and $X \in \text{Space}$. Let $M = \langle G', K'' \rangle$ and $X = \mathtt{sp}(e_1) = \mathtt{sp}(e_2)$. By Prop. 6.6, F is encapsulated in M. Thus, since $\mathtt{sp}(e_1) = \mathtt{sp}(e_2)$, it follows that one of the following holds:

(1) $e_1 \in \mathsf{E}_{\overline{F}}$ and $e_2 \in \mathsf{E}_{\overline{F}}$: Since $\mathsf{H}_F(Pr[L]) \subseteq \mathsf{H}_F(MGC[L^\#])$, by applying Lemma 7.5 with $L := L^\#$, $L' := L$, $Pr := Pr$, and $Pr' := MGC$, we obtain a program state $\overline{p}_\#$ and a memory state $M_\# = \langle G_\#, K_\# \rangle$ such that $\langle \overline{p}'[F \mapsto \overline{p}_\#], M_\# \rangle$ is reachable in $Pr[L^\#] \bowtie \mathsf{pRC11}$ and $M_\#|_{\overline{F}} = M|_{\overline{F}}$. Since $Pr[L^\#]$ is not racy under pRC11, we have $\langle e_1, e_2 \rangle \in G_\#.\mathsf{hb}_X \cup G_\#.\mathsf{hb}_X^{-1}$. Suppose w.l.o.g. that $\langle e_1, e_2 \rangle \in G_\#.\mathsf{hb}_X$. Let $\tau = \mathtt{tid}(e_2)$.

Claim A.16.1: $\langle e_1, e_2 \rangle \in G_\#|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X \,;\, [\mathsf{CR}_F] \,;\, G_\#.\mathsf{hb}_X$.

Proof. By Prop. A.14, we have $\langle e_1, e_2 \rangle \in G_\#|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X \,;\, ([\mathsf{CR}_F] \,;\, G_\#.\mathsf{hb}_X)^?$. Since $G_\#|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X = G'|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X \subseteq G'.\mathsf{hb}_X$, and e_1, e_2 form a race in G', we cannot have $\langle e_1, e_2 \rangle \in G_\#|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X$. ◁

Let $f \in \mathsf{CR}_F$ such that $\langle e_1, f \rangle \in G_\#|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X$ and $\langle f, e_2 \rangle \in G_\#.\mathsf{hb}_X$.

Claim A.16.2: $\langle e_1, f \rangle \in G'.\mathsf{hb}_X$.

Proof. Immediately follows using the fact that $G_\#|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X = G'|_{\mathsf{E}_{\overline{F}}}.\mathsf{hb}_X \subseteq G'.\mathsf{hb}_X$. ◁

Claim A.16.3: $f \in dom(G'.\mathsf{hb}^?_X \,;\, [\mathsf{E}^\tau])$.

Proof. Using Prop. 5.8, $\langle f, e_2 \rangle \in G_\#.\mathsf{hb}_X$ implies that $f \in K_\#(\tau, X)$. Since $M_\#|_{\overline{F}} = M|_{\overline{F}}$, we have $f \in K''(\tau, X)$ as well. Since $K''(\tau, X) \cap \mathsf{CR} \subseteq dom(G'.\mathsf{hb}^?_X \,;\, [\mathsf{E}^\tau])$, it follows that $f \in dom(G'.\mathsf{hb}^?_X \,;\, [\mathsf{E}^\tau])$. ◁

From the two last claims, we obtain that $e_1 \in dom(G'.\mathsf{hb}_X \,;\, [\mathsf{E}^\tau])$.

Claim A.16.4: e_2 is $G'.\mathsf{po}$-maximal.

Proof. Suppose otherwise, and let e be a $G'.\mathsf{po}$-maximal event with $\mathtt{tid}(e) = \tau$. Then, $e_1 \in dom(G'.\mathsf{hb}_X \,;\, [\mathsf{E}^\tau])$ implies that $\langle e_1, e \rangle \in G'.\mathsf{hb}_X$. It follows that $\langle e_1, e \rangle \in (G'.\mathsf{po} \cup G'.\mathsf{rf})^+$. However, since $e \in G'.\mathsf{E}$, we either have $\langle e, e_1 \rangle \in (G'.\mathsf{po} \cup G'.\mathsf{rf})^+$ or $\langle e, e_2 \rangle \in (G'.\mathsf{po} \cup G'.\mathsf{rf})^+$, and both contradict the fact that G' is dRC11-consistent by Lemma 5.9 (as they violate po-rf). ◁

Using the last claim, the fact that $e_1 \in dom(G'.\mathsf{hb}_X \,;\, [\mathsf{E}^\tau])$ implies that $\langle e_1, e_2 \rangle \in G'.\mathsf{hb}_X$, which contradicts the fact that e_1 and e_2 form a race in G'.

(2) $e_1 \in \mathsf{E}_F$ and $e_2 \in \mathsf{E}_F$: Since $\mathsf{H}_F(Pr[L]) \subseteq \mathsf{H}_F(MGC[L^\#])$, by applying Lemma 7.5 with $L := L$, $L' := L^\#$, $Pr := MGC$, and $Pr' := Pr$, we obtain a program state $\overline{p}^*$ and a memory state $M^* = \langle G^*, K^* \rangle$ such that $\langle \overline{p}^*[F \mapsto \overline{p}'], M^* \rangle$ is reachable in $MGC[L] \bowtie \mathrm{pRC11}$ and $M^*|_F = M|_F$. Since $MGC[L]$ is not racy under pRC11 we have $\langle e_1, e_2 \rangle \in G'.\mathsf{hb}_X \cup G'.\mathsf{hb}_X^{-1}$. Suppose w.l.o.g. that $\langle e_1, e_2 \rangle \in G'.\mathsf{hb}_X$. Let $\tau = \mathsf{Tid}(e_2)$.

Claim A.16.5: $\langle e_1, e_2 \rangle \in G^*|_{\mathsf{E}_F}.\mathsf{hb}_X \,;\, [\mathsf{CR}_F] \,;\, G^*.\mathsf{hb}_X$.

Proof. By Prop. A.14, we have $\langle e_1, e_2 \rangle \in G^*|_{\mathsf{E}_F}.\mathsf{hb}_X \,;\, ([\mathsf{CR}_F] \,;\, G^*.\mathsf{hb}_X)^?$. Since $G^*|_{\mathsf{E}_F}.\mathsf{hb}_X = G'|_{\mathsf{E}_F}.\mathsf{hb}_X \subseteq G'.\mathsf{hb}_X$, and e_1, e_2 form a race in G', we cannot have $\langle e_1, e_2 \rangle \in G^*|_{\mathsf{E}_F}.\mathsf{hb}_X$. ◁

Let $f \in \mathsf{CR}_F$ such that $\langle e_1, f \rangle \in G^*|_{\mathsf{E}_F}.\mathsf{hb}_X$ and $\langle f, e_2 \rangle \in G^*.\mathsf{hb}_X$.

Claim A.16.6: $\langle e_1, f \rangle \in G'.\mathsf{hb}_X$.

Proof. Immediately follows using the fact that $G^*|_{\mathsf{E}_F}.\mathsf{hb}_X = G'|_{\mathsf{E}_F}.\mathsf{hb}_X \subseteq G'.\mathsf{hb}_X$. ◁

Claim A.16.7: $f \in dom(G'.\mathsf{hb}_X^? \,;\, [\mathsf{E}^\tau])$.

Proof. Using Prop. 5.8, $\langle f, e_2 \rangle \in G^*.\mathsf{hb}_X$ implies that $f \in K^*(\tau, X)$. Since $M^*|_F = M|_F$, we have $f \in K''(\tau, X)$ as well. Since $K''(\tau, X) \cap \mathsf{CR} \subseteq dom(G'.\mathsf{hb}_X^? \,;\, [\mathsf{E}^\tau])$, it follows that $f \in dom(G'.\mathsf{hb}_X^? \,;\, [\mathsf{E}^\tau])$. ◁

From the two last claims, we obtain that $e_1 \in dom(G'.\mathsf{hb}_X \,;\, [\mathsf{E}^\tau])$.

Claim A.16.8: e_2 is $G'.\mathsf{po}$-maximal.

Proof. Similar to the proof of Claim A.16.4. ◁

Using the last claim, the fact that $e_1 \in dom(G'.\mathsf{hb}_X \,;\, [\mathsf{E}^\tau])$ implies that $\langle e_1, e_2 \rangle \in G'.\mathsf{hb}_X$, which contradicts the fact that e_1 and e_2 form a race in G'. □

Corollary 7.4 (Compositionality). *Let $L_1, \dots, L_n$ be libraries implementing pairwise disjoint sets of methods, such that $\mathsf{Space}(L_1), \dots, \mathsf{Space}(L_n), \mathsf{Space}(L_1^\#), \dots, \mathsf{Space}(L_n^\#)$, and $\mathsf{Space}(MGC \setminus dom(L_1 \uplus \dots \uplus L_n))$ are pairwise disjoint. Suppose that for every $1 \leq i \leq n$, we have $L_i \sqsubseteq_{MGC_i} L_i^\#$ for $MGC_i = MGC[L_1^\# \uplus \dots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \dots \uplus L_n^\#]$. Then, $L_1 \uplus \dots \uplus L_n \sqsubseteq_{MGC} L_1^\# \uplus \dots \uplus L_n^\#$.*

Proof. We prove the claim by induction on n. For $n = 1$, the claim trivially holds. For the induction step, let $L_1, \dots, L_n, L_1^\#, \dots, L_n^\#$ be libraries, and let MGC be a program satisfying the required conditions. For $MGC' = MGC[L_n^\#]$, we have that $\mathsf{Space}(L_1), \dots, \mathsf{Space}(L_{n-1}), \mathsf{Space}(L_1^\#), \dots, \mathsf{Space}(L_{n-1}^\#)$, and $\mathsf{Space}(MGC' \setminus dom(L_1 \uplus \dots \uplus L_{n-1}))$ are pairwise disjoint. In addition, for every $1 \leq i \leq n-1$,

$$MGC'[L_1^\# \uplus \dots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \dots \uplus L_{n-1}^\#] = MGC[L_n^\#][L_1^\# \uplus \dots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \dots \uplus L_{n-1}^\#] = MGC[L_1^\# \uplus \dots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \dots \uplus L_n^\#].$$

Hence, for every $1 \leq i \leq n-1$, we have $L_i \sqsubseteq_{MGC_i} L_i^\#$ for $MGC_i = MGC'[L_1^\# \uplus \dots \uplus L_{i-1}^\# \uplus L_{i+1}^\# \uplus \dots \uplus L_{n-1}^\#]$. By the induction hypothesis, it follows that $L \sqsubseteq_{MGC'} L^\#$ for $L = L_1 \uplus \dots \uplus L_{n-1}$ and $L^\# = L_1^\# \uplus \dots \uplus L_{n-1}^\#$, which implies that $MGC[L] \sqsubseteq_{L_n^\#} MGC[L^\#]$. In addition, by assumption we have $L_n \sqsubseteq_{MGC_n} L_n^\#$ for $MGC_n = MGC[L^\#]$. Hence, the abstraction theorem (Thm. 7.3) ensures that $L_n \sqsubseteq_{MGC[L]} L_n^\#$. This implies that $L \uplus L_n \sqsubseteq_{MGC} L \uplus L_n^\#$. In addition, $L \sqsubseteq_{MGC'} L^\#$ also implies that $L \uplus L_n^\# \sqsubseteq_{MGC} L^\# \uplus L_n^\#$. Finally, $L \uplus L_n \sqsubseteq_{MGC} L^\# \uplus L_n^\#$ follows from $L \uplus L_n \sqsubseteq_{MGC} L \uplus L_n^\#$ and $L \uplus L_n^\# \sqsubseteq_{MGC} L^\# \uplus L_n^\#$. □

A.5 Proof Sketch of Queue Library Correctness (§8.2)

Theorem A.17. *For the queue implementation L_{queue} and specification $L_{\mathsf{queue}}^\#$ as given in §8.2, we have $L_{\mathsf{queue}} \sqsubseteq_{MGC\mathsf{free}} L_{\mathsf{queue}}^\#$.*

Proof. To show refinement we consider histories of $MGC\mathsf{free}[L_{\mathsf{queue}}]$ and $MGC\mathsf{free}[L_{\mathsf{queue}}^\#]$ which consists of labels corresponding to call, return, and the propagation of the call and return of the enqueue and dequeue methods. Since the local registers are not used for passing values to (and from) the functions, in the following discussion we ignore the bookkeeping of the register variables in the history labels. We observe the following, *Remark 5.* For a given trace $t \in \text{traces}(\mathit{MGC}\text{free}[L_{\text{queue}}]\bowtie\text{pRC11})$ of the queue implementation, no two different Ws appearing on the trace t can write to the same location in the queue. It is true because a write to the location $q[n]$ happens only after n is obtained by the most recent preceding RMW in trace t with $\texttt{loc}(\texttt{RMW}) = \texttt{T}$. Since, due the release acquire semantics, no two RMWs with $\texttt{loc}(\texttt{RMW}) = \texttt{T}$ can read from the same write, we will always get a different values for n. Therefore, any location in the queue can be written at most once. Let $v[n]$ represents the value written at the location $q[n]$ by some write on the trace t. Also, by looking at the structure of enqueue method (in specification program), it easy to see that no two different writes Ws appearing on a given trace $t^{\#} \in \text{traces}(\mathit{MGC}\text{free}[L^{\#}_{\text{queue}}]\bowtie\text{pRC11})$ can write to the same location in the queue. Let $v^{\#}[n]$ represents the value written at the location $q[n]$ by some write on the trace $t^{\#}$.

Remark 6. For a given trace $t \in \text{traces}(\mathit{MGC}\text{free}[L_{\text{queue}}]\bowtie\text{pRC11})$ of the implementation program, the value read by an RMW, with $\texttt{loc}(\texttt{RMW}) = \texttt{H}$ appearing in the trace t is the value written by the most recent previous RMW with $\texttt{loc}(\texttt{RMW}) = \texttt{H}$. On the other hand for a given trace $t \in \text{traces}(\mathit{MGC}\text{free}[L^{\#}_{\text{queue}}]\bowtie\text{pRC11})$ of the specification program it is easy to see that the value read by a R with $\texttt{loc}(\texttt{R}) = \texttt{H}$ is same as the value written by the most recent previous write W with $\texttt{loc}(\texttt{W}) = \texttt{H}$.

Remark 7. A read R by thread τ, with $\texttt{loc}(\texttt{R}) = q[n]$, appearing on a given trace $t \in \text{traces}(\mathit{MGC}\text{free}[L_{\text{queue}}]\bowtie\text{pRC11})$ can either return $v[n]$ or $\perp$, if the prefix of t, containing this R as the last label, does not contain a propagation level of the form $\text{EP}(e, \tau, X_L)$ where e is an event corresponding to return of an enqueue method. Otherwise, such a read can only return $v[n]$. Similarly, a read R with $\texttt{loc}(\texttt{R}) = q[n]$ appearing on a given trace $t \in \text{traces}(\mathit{MGC}\text{free}[L^{\#}_{\text{queue}}]\bowtie\text{pRC11})$ can either return $v^{\#}[n]$ or $\perp$ when the prefix of t, containing this R as the last label, does not contain a propagation level of the form $\text{EP}(e, \tau, X_L)$ where e is an event corresponding to return of an enqueue method. Otherwise, such a read can only return $v^{\#}[n]$.

Next we define the notion of *similar traces* to state and prove our main result. Let $L_W := \{\langle\tau, l\rangle \mid \tau \in \text{Tid} \wedge l \in \{\texttt{W}(\texttt{X}_\texttt{L}, \texttt{T}, ,), \texttt{W}(\texttt{X}_\texttt{L}, \texttt{H}, ,)\}\}$ be the collection of labels corresponding to the execution of instructions $\texttt{store}(\texttt{X}_\texttt{L}, \texttt{T}, \texttt{tp}+1, \texttt{rlx})$ and $\texttt{store}(\texttt{X}_\texttt{L}, \texttt{H}, \texttt{hp}+1, \texttt{rlx})$ from the specification program of the queue object. Similarly, let $L_U := \{\langle\tau, l\rangle \mid \tau \in \text{Tid} \wedge l \in \{\texttt{RMW}(\texttt{X}_\texttt{L}, \texttt{T}, , ,), \texttt{RMW}(\texttt{X}_\texttt{L}, \texttt{H}, , ,)\}\}$ be the collection of RMWs labels corresponding to the successful execution of CAS instruction inside the enqueue and dequeue methods of the queue implementation. For $l_U \in L_U$ and $l_W \in L_W$, we say $l_U \approx l_W$ iff $\texttt{tid}(l_U) = \texttt{tid}(l_W) \wedge \texttt{loc}(l_U) = \texttt{loc}(l_W) \wedge \texttt{val}_\texttt{W}(l_U) = \texttt{val}_\texttt{W}(l_W)$. For the traces $t \in \text{traces}(\mathit{MGC}\text{free}[L_{\text{queue}}]\bowtie\text{pRC11})$ and $t^{\#} \in \text{traces}(\mathit{MGC}\text{free}[L^{\#}_{\text{queue}}]\bowtie\text{pRC11})$, we say that *$t$ is similar to $t^{\#}$*, denoted as $t \simeq t^{\#}$, if $\forall i, t|_{\text{HLab}_F \cup L_U}[i] \in \text{HLab}_F \implies t|_{\text{HLab}_F \cup L_U}[i] = t^{\#}|_{\text{HLab}_F \cup L_W}[i]$, $\forall i, t|_{\text{HLab}_F \cup L_U}[i] \in L_U \implies t|_{\text{HLab}_F \cup L_U}[i] \approx t^{\#}|_{\text{HLab}_F \cup L_W}[i]$, and $\forall n, v[n] \neq \perp \implies v[n] = v^{\#}[n]$.

Intuitively, if trace t is similar to trace $t^{\#}$, then their entries corresponding to history labels are exactly same and the entries from L_U in t is replaced with corresponding entries from L_W in $t^{\#}$. It is easy to see that $\text{H}_F(t) = \text{H}_F(t^{\#})$ whenever $t \simeq t^{\#}$. Therefore, our main result follows from the following claim,

- For every $t \in \text{traces}(\mathit{MGC}\text{free}[L_{\text{queue}}]\bowtie\text{pRC11})$ where $\langle\overline{p}_{\text{Init}}, M_{\text{Init}}\rangle \xrightarrow{t}_{\mathit{MGC}\text{free}[L_{\text{queue}}]\bowtie\text{pRC11}} \langle\overline{p}, M\rangle$, there exists $t^{\#} \in \text{traces}(\mathit{MGC}\text{free}[L^{\#}_{\text{queue}}]\bowtie\text{pRC11})$ such that $\langle\overline{p}_{\text{Init}}, M_{\text{Init}}\rangle \xrightarrow{t^{\#}}_{\mathit{MGC}\text{free}[L^{\#}_{\text{queue}}]\bowtie\text{pRC11}} \langle\overline{p}^{\#}, M^{\#}\rangle$, $t \simeq t^{\#}$, and locks are open in $M^{\#}$.

We prove the above claim using induction on the length of trace t. Consider the last transition label α in t, and let t' such that $t = t' \cdot \alpha$. Let $\langle\overline{p}_{\text{Init}}, M_{\text{Init}}\rangle \xrightarrow{t'}_{\mathit{MGC}\text{free}[L_{\text{queue}}]\bowtie\text{pRC11}} \langle\overline{p}', M'\rangle \xrightarrow{\alpha}_{\mathit{MGC}\text{free}[L_{\text{queue}}]\bowtie\text{pRC11}} \langle\overline{p}, M\rangle$. By induction hypothesis, there exists $t'' \in \text{traces}(\mathit{MGC}\text{free}[L^{\#}_{\text{queue}}]\bowtie\text{pRC11})$ such that

$\langle \overline{p}_{\mathsf{Init}}, M_{\mathsf{Init}} \rangle \xrightarrow{t''}_{MGC\mathsf{free}[L^{\#}_{\mathsf{queue}}] \bowtie \mathsf{pRC11}} \langle \overline{p}'', M'' \rangle$ and $t' \simeq t''$. We consider the following four cases and show the existence of $t^{\#}$ such that $t \simeq t^{\#}$ in each of these cases. For all other possible values of α we let $t^{\#} := t''$ and it is easy to see that $t \simeq t''$.

(1) $\alpha = \mathsf{EP}(e, \tau, X_L)$ and $\mathtt{typ}(e) \in \{\mathtt{CALL}, \mathtt{RET}\}$: Let $\mathtt{tid}(\alpha) = \tau$, then α represents the propagation of call (or return) of an enqueue (or dequeue) method invoked by thread τ. If $\mathtt{typ}(e) = \mathtt{CALL}$, then it is easy to see that M'' can also take this step. On the other hand if $\mathtt{typ}(e) = \mathtt{RET}$, then $MGC\mathsf{free}[L^{\#}_{\mathsf{queue}}] \bowtie \mathsf{pRC11}$ can take the α step after propagating all the events on thread τ between the call and return of the method whose return is currently being propagated. Let $t^{\#}$ be the resulting trace of $MGC\mathsf{free}[L^{\#}_{\mathsf{queue}}] \bowtie \mathsf{pRC11}$, it is easy to see that $t \simeq t^{\#}$ and the locks are still open in $M^{\#}$.

(2) $\mathtt{typ}(\alpha) = \mathtt{CALL}$: Let $\mathtt{tid}(\alpha) = \tau$, then α signifies that the last executed instruction is either a call to enqueue or dequeue by thread τ. It is easy to see that thread τ of $MGC\mathsf{free}[L^{\#}_{\mathsf{queue}}] \bowtie \mathsf{pRC11}$ can also take an α step and $t \simeq t'' \cdot \alpha$ while the locks are still open in $M^{\#}$.

(3) $\mathtt{typ}(\alpha) = \mathtt{RMW}$: Let $\mathtt{tid}(\alpha) = \tau$, then α signifies that the last executed instruction is either a $\mathtt{tp}' := \mathtt{CAS}(\mathsf{X_L}, \mathsf{T}, \mathtt{tp}, \mathtt{tp}+1, \mathtt{rlx}, \mathtt{rlx}, \mathtt{rlx})$ or a $\mathsf{hp}' := \mathtt{CAS}(\mathsf{X_L}, \mathsf{H}, \mathsf{hp}, \mathsf{hp}+1, \mathtt{rlx}, \mathtt{rlx}, \mathtt{rlx})$ by thread τ. We consider the $\mathtt{tp}' := \mathtt{CAS}(\mathsf{X_L}, \mathsf{T}, \mathtt{tp}, \mathtt{tp}+1, \mathtt{rlx}, \mathtt{rlx}, \mathtt{rlx})$ step taken by the enqueue implementation. Note that due to induction hypothesis t'' already has a label corresponding to the call of the enqueue method by thread τ in $MGC\mathsf{free}[L^{\#}_{\mathsf{queue}}] \bowtie \mathsf{pRC11}$. We obtain $t^{\#}$ from t'' by appending in sequential order the labels corresponding to instruction between $\mathtt{pacquire}(\mathtt{LT})$ and $\mathtt{prelease}(\mathtt{LT})$ (including the labels for the $\mathtt{pacquire}$ and $\mathtt{prelease}$ instructions). It is easy to see that $t \simeq t^{\#}$ and the locks remain open in $M^{\#}$. We can give a similar argument when α corresponds to the $\mathsf{hp}' := \mathtt{CAS}(\mathsf{X_L}, \mathsf{H}, \mathsf{hp}, \mathsf{hp}+1, \mathtt{rlx}, \mathtt{rlx}, \mathtt{rlx})$ step taken by the dequeue method of the implementation program.

(4) $\mathtt{typ}(\alpha) = \mathtt{RET}$: Let $\mathtt{tid}(\alpha) = \tau$, then α signifies that the last executed instruction is a return of enqueue (or of dequeue) by thread τ. It is easy to see that thread τ of $MGC\mathsf{free}[L^{\#}_{\mathsf{queue}}] \bowtie \mathsf{pRC11}$ can also take this α step (i.e., $t \simeq t'' \cdot \alpha$), and the locks are open in $M^{\#}$. If α is a dequeue we claim that the value returned will be same. If the value returned, by the dequeue of the implemented library L_{queue}, is $\perp$ then due to Remark 7 and induction hypothesis, the dequeue of the specification $L^{\#}_{\mathsf{queue}}$ can return $\perp$. On the other hand, if the value retuned is $v[n] \neq \perp$, then n is the value returned by the most recent RMW on the location H. In this case, the dequeue of implementation would have executed the instructions $\mathsf{hp} := \mathtt{load}(\mathsf{X_L}, \mathsf{H}, \mathtt{rlx})$ and $\mathsf{hc} := \mathtt{load}(\mathsf{X_L}, \mathsf{q}[\mathsf{hp}], \mathtt{rlx})$. Therefore, due to Remark 5, Remark 6, Remark 7, and the induction hypothesis the dequeue of specification can return the value $v^{\#}[n]$. In the present case, when $v[n] \neq \perp$, by induction hypothesis we know that $v[n] = v^{\#}[n]$ which completes our proof. $\square$

Received 2022-07-07; accepted 2022-11-07