# Parameterised Linearisability

Andrea Cerone[1], Alexey Gotsman[1], and Hongseok Yang[2]

[1] IMDEA Software Institute
[2] University of Oxford

**Abstract** Many concurrent libraries are parameterised, meaning that they implement generic algorithms that take another library as a parameter. In such cases, the standard way of stating the correctness of concurrent libraries via linearisability is inapplicable. We generalise linearisability to parameterised libraries and investigate subtle trade-offs between the assumptions that such libraries can make about their environment and the conditions that linearisability has to impose on different types of interactions with it. We prove that the resulting parameterised linearisability is closed under instantiating parameter libraries and composing several non-interacting libraries, and furthermore implies observational refinement. These results allow modularising the reasoning about concurrent programs using parameterised libraries and confirm the appropriateness of the proposed definitions. We illustrate the applicability of our results by proving the correctness of a parameterised library implementing flat combining.

## 1 Introduction

Concurrent libraries encapsulate high-performance concurrent algorithms and data structures behind a well-defined interface, providing a set of methods for clients to call. Many such libraries [6,7,13] are *parameterised*, meaning that they implement generic algorithms that take another library as a parameter and use it to implement more complex functionality (we give a concrete example in §2). Reasoning about the correctness of parameterised libraries is challenging, as it requires considering all possible libraries that they can take as parameters.

Correctness of concurrent libraries is usually stated using *linearisability* [8], which fixes a certain correspondence between the *concrete* library implementation and a (possibly simpler) *abstract* library, whose behaviour the concrete one is supposed to simulate. For example, a high-performance concurrent stack that allows multiple push and pop operations to access the data structure at the same time may be specified by an abstract library where each operation takes effect atomically. However, linearisability considers only *ground* libraries, where all of the library implementation is given, and is thus inapplicable to parameterised ones. In this paper we propose a notion of *parameterised linearisability* (§3 and §4) that lifts this limitation. The key idea is to take into account not only interactions of a library with its client, but also with its parameter library, with the two types of interactions being subject to different conditions.

A challenge we have to deal with while generalising linearisability in this way is that parameterised libraries are often correct only under some assumptions about the context in which they are used. Thus, a parameterised library may assume that the library it takes as a parameter is *encapsulated*, meaning that clients cannot call its methods directly. A parameterised library may also accept as a parameter only libraries satisfying certain properties. For this reason, we actually present three notions of parameterised

linearisability, appropriate for different situations: a general one, which does not make any assumptions about the client or the parameter library, a notion appropriate for the case when the parameter library is encapsulated, and *up-to linearisability*, which allows making assumptions about the parameter library. These notions differ in subtle ways: we find that there is a trade-off between the assumptions that parameterised libraries make about their environment and the conditions that a notion of linearisability has to impose on different types of interactions with it.

We prove that the proposed notions of parameterised linearisability are *contextual* (§5), i.e., closed under parameter instantiation. This includes the case when the parameter library is itself parameterised. On the other hand, when the parameter is an ordinary ground library, this result allows us to derive the classical linearisability of the instantiated library from our notion for the parameterised one. We also prove that parameterised linearisability is *compositional* (§5): if several non-interacting libraries are linearisable, so is their composition. Finally, we show that parameterised linearisability implies *observational refinement* (§6): the behaviours of any complete program using a concrete parameterised library can be reproduced if the program uses a corresponding abstract one instead. All these results allow modularising the reasoning about concurrent programs using parameterised libraries: contextuality and compositionality break the reasoning about complex parameterised libraries into that about individual libraries from which they are constructed; observational refinement then lifts this to complete programs, including clients. The properties of parameterised linearisability we establish also serve to confirm the appropriateness of the proposed definitions.

We illustrate the applicability of our results by proving the up-to linearisability of flat combining [6] (§4), a generic algorithm for converting hard-to-parallelise sequential data structures into concurrent ones.

Due to space constraints, we defer the proofs of most theorems to [1, §B].

## 2 Parameterised Libraries

We consider ***parameterised libraries*** (or simply libraries) $L$, which provide some ***public methods*** to their ***clients***. The latter are multi-threaded programs that can call the methods in parallel. In §4 and §6 we introduce a particular syntax for libraries and clients; for now it suffices to treat them abstractly. Our libraries are called parameterised because we allow their method implementations to call ***abstract methods***, whose implementation is left unspecified. Abstract methods are meant to be implemented by another library provided by $L$'s client, which we call the ***parameter library*** of $L$.

We identify methods by names from a set $\mathcal{M}$, ranged over by $m$, and threads by identifiers from a set $\mathcal{T}$, ranged over by $t$. For the sake of simplicity, we assume that methods take a single integer as a parameter and always return an integer. We annotate libraries with types as in $L : M \to M'$, where $M, M' \subseteq \mathcal{M}$ give the sets of abstract and public methods of $L$, respectively. If $M = \emptyset$ we call $L$ a ***ground library***. The sets $M$ and $M'$ do not have to be disjoint: methods in $M \cap M'$ may be called by $L$'s clients, but their implementation is inherited from the one given by the parameter library.

**Example: Flat Combining.** Flat combining [6] is a recent synchronisation paradigm, which can be viewed [14] as a parameterised library $\mathsf{FC} : \{m_i\}_{i=1}^n \to \{\mathsf{do\_}m_i\}_{i=1}^n$ for a given set of methods $\{m_i\}_{i=1}^n$. In Figure 1 we show a pseudocode of its implementation, which simplifies the original one in ways orthogonal to our goals. FC takes a library,

whose methods $m_i$ are meant to be executed sequentially, and efficiently turns it into a library with methods do_$m_i$ that can be called concurrently.

As usual, this is achieved by means of mutual exclusion, implemented using a lock, but in a way that is more sophisticated than just acquiring it before calling a method $m_i$. A thread executing do_$m_i$ first publishes the operation it would like to execute and its parameter in its entry of the requests array. It then spins, trying to acquire the global lock. Having acquired a lock, the thread becomes a *combiner*: it performs the operations requested by all threads, stored in requests, by calling methods $m_i$ of the parameter library and writing the values returned into the retval field of the corresponding entries in requests. Each spinning thread periodically checks this field and stops if some other thread has performed the operation it requested (for simplicity, we assume that nil is a special value that is never returned by any method). This algorithm benefits from cache locality when the combiner executes several operations in sequence, and thus yields good performance even for hard-to-parallelise data structures, such as stacks and queues.

```
LOCK lock;
struct{op,param,retval} requests[NThread];

do_m_i(int z):
  requests[mytid()].op = i;
  requests[mytid()].param = z;
  requests[mytid()].retval = nil;
  do:
    if (lock.tryacquire()):
      for (t = 0; t < NThread; t++):
        if (requests[t].retval == nil):
          int j = requests[t].op;
          int w = requests[t].param;
          requests[t].retval = m_j(w);
      lock.release();
  while (requests[mytid()].retval == nil);
  return requests[mytid()].retval;
```

**Figure 1.** Flat combining: implementation FC.

In this paper, we develop a framework for specifying and verifying parameterised concurrent libraries. For flat combining, our framework suggests using an ***abstract*** library $\mathsf{FC}^\sharp : \{m_i\}_{i=1}^n \to \{\mathsf{do}\_m_i\}_{i=1}^n$ in Figure 2 as a specification for the ***concrete*** library in Figure 1. $\mathsf{FC}^\sharp$ specifies the expected behaviour of flat combining by using the naive mutual exclusion. Showing that the implementation satisfies this specification in our framework amounts to proving that it is related to $\mathsf{FC}^\sharp$ by *parameterised linearisability*, which we present next.

```
LOCK lock;
do_m_i(int z):
  lock.acquire();
  int retval = m_i(z);
  lock.release();
  return retval;
```

**Figure 2.** Flat combining: specification $\mathsf{FC}^\sharp$.

## 3 Histories and Parameterised Linearisability

**Histories.** Informally, for a concrete library (such as the one in Figure 1) to be correct with respect to an abstract one (such as the one in Figure 2), the two should interact with their environment—the client and the parameter library—in similar ways. In this paper, we assume that different libraries and their clients access disjoint portions of memory, and thus interactions between them are limited to passing parameters and return values at method calls and returns. This is a standard assumption [8], which we believe can be relaxed using existing techniques [5]; see §7 for discussion. We record interactions of a parameterised library $L : M \to M'$ with its environment using *histories* (Definition 1 below), which are certain sequences of ***actions*** of the form

$$\mathsf{Act} ::= (t, \mathsf{call?}\, m'(z)) \mid (t, \mathsf{ret!}\, m'(z)) \mid (t, \mathsf{call!}\, m(z)) \mid (t, \mathsf{ret?}\, m(z)),$$

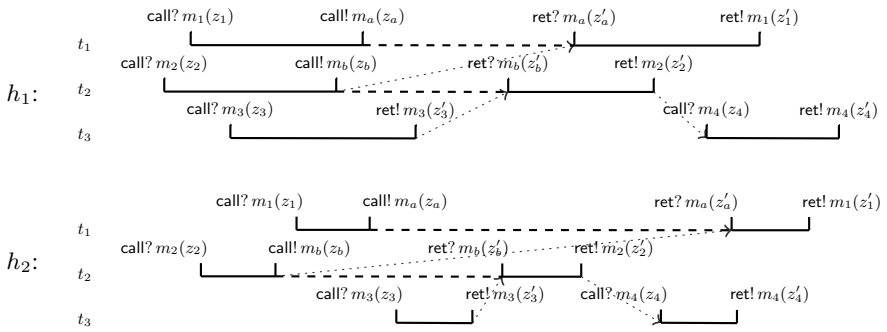


**Figure 4.** Illustration of histories and parameterised linearisability. A solid line represents a thread executing the code of the parameterised library, and a dashed one, the parameter library.

where $t \in \mathcal{T}$ is the thread performing the action, $m' \in M'$ or $m \in M$ is the method involved, and $z \in \mathbb{Z}$ is the method parameter or a return value.

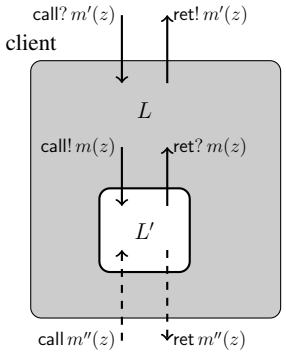


**Figure 3.** Interactions of a library $L$ with its client and parameter library $L'$.

We illustrate the meaning of the actions in Figure 3: call? and ret! describe the client invoking public methods $m'$ of the parameterised library $L$, and call! and ret? the library $L$ invoking implementations of abstract methods $m$ provided by a parameter library $L'$. We denote the sets of actions corresponding to interactions with these two entities by ClAct and AbsAct, respectively. In the spirit of the opponent-proponent distinction in game semantics [9,11], we annotate actions by ! or ? depending on whether the action was initiated by $L$ or by an external entity, and we denote the corresponding sets of actions by Act! and Act?. We also use sets ActCall?, ActRet!, ActCall! and ActRet? with the expected meaning. Clients can also call methods $m'' \in M \cap M'$ directly, as represented by the dashed lines in the figure. Since such interactions do not involve the library $L$, we do not include them into Act. Histories are finite sequences of actions with invocations of abstract methods properly nested inside those of public ones.

DEFINITION 1 (Histories) *A **history** $h : M \to M'$ is a finite sequence of actions such that for every $t$, the projection of $h$ to $t$'s actions is a prefix of a sequence generated by the grammar* SHist *below, where $m \in M$ and $m' \in M'$:*

$$\begin{array}{lll} \mathsf{SHist} & ::= \varepsilon \mid (t, \mathsf{call?}\, m'(z))\, \mathsf{IntSHist}\, (t, \mathsf{ret!}\, m'(z')) \mid \mathsf{SHist}\ \mathsf{SHist} \\ \mathsf{IntSHist} & ::= \varepsilon \mid (t, \mathsf{call!}\, m(z))\, (t, \mathsf{ret?}\, m(z')) \mid \mathsf{IntSHist}\ \mathsf{IntSHist} \end{array}$$

We denote the set of histories by Hist. See Figure 4 for examples. In this paper, we focus on safety properties of libraries and thus let histories be finite. This assumption is also taken by the classical notion of linearisability [8] and can be relaxed as described in [4] (§7). For a history $h$ and $A \subseteq \mathsf{Act}$, we let $h|_A$ be the projection of $h$ onto actions in $A$ and we denote the $i$-th action in $h$ by $h(i)$.

**Parameterised Linearisability.** We would like the notion of correctness of a concrete library $L : M \to M'$ with respect to an abstract one $L^\sharp : M \to M'$ to imply *observational refinement*. Informally, this property means that $L^\sharp$ can be used to replace $L$ in any program (consisting of a client, the library and an instantiation of the parameter library) while keeping its observable behaviours reproducible; a formal definition is given in §6. While this notion is intuitive, establishing it between two libraries directly is challenging because of the quantification over all possible programs they can be used by. We therefore set out to find a correctness criterion that compares the concrete and abstract libraries in isolation and thus avoids this quantification. For ground libraries, ***linearisability*** [8] formulates such a criterion by matching a history $h_1$ of $L$ with a history $h_2$ of $L^\sharp$ that yields the same client-observable behaviour. The following definition generalises it to parameterised libraries.

DEFINITION 2 (Parameterised linearisability: general case) *A history $h_1 : M \to M'$ is **linearised** by another one $h_2 : M \to M'$, written $h_1 \sqsubseteq h_2$, if there exists a permutation $\pi : \mathbb{N} \to \mathbb{N}$ such that*

$$\forall i.\, h_1(i) = h_2(\pi(i)) \wedge (\forall j.\, i < j \wedge ((\exists t.\, h_1(i) = (t, -) \wedge h_2(j) = (t, -)) \vee (h_1(i) \in \mathsf{Act!} \wedge h_1(j) \in \mathsf{Act?})) \implies \pi(i) < \pi(j)).$$

*For sets of histories $H_1, H_2$ we let $H_1 \sqsubseteq H_2 \iff \forall h_1 \in H_1.\, \exists h_2 \in H_2.\, h_1 \sqsubseteq h_2$.*

In §4 we show how to generate all histories of a library in a particular language and define linearisability on libraries by the $\sqsubseteq$ relation on their sets of histories. For now we explain the above abstract definition. According to it, a history $h_1$ is linearised by a history $h_2$ when the latter is a permutation of the former preserving the order of actions within threads and the precedence relation between the actions initiated by the library and those initiated by its environment. As we explain below, we have $h_1 \sqsubseteq h_2$ for the histories $h_1, h_2$ in Figure 4. Hence, parameterised linearisability is able to match a history of a concurrent library with a simpler one where every contiguous block of library execution (e.g., the one between $(t_1, \mathsf{call?}\, m_1(z_1))$ and $(t_1, \mathsf{call!}\, m_a(z_a))$) is executed without interleaving with other such blocks. On the other hand, $h_2 \not\sqsubseteq h_1$, since $(t_1, \mathsf{call!}\, m_a(z_a))$ precedes $(t_3, \mathsf{call?}\, m_3(z_3))$ in $h_2$, but not in $h_1$.

When $h_1, h_2 : \emptyset \to M'$, i.e., these are histories of a ground library and thus contain only call? and ret! actions, Definition 2 coincides with a variant of the classical linearisability [8], which requires preserving the order between ret! and call? actions. For example, Definition 2 requires preserving the order between $(t_2, \mathsf{ret!}\, m_2(z_2'))$ and $(t_3, \mathsf{call?}\, m_4(z_4))$ in $h_1$ from Figure 4 (shown by a diagonal arrow). This requirement is needed for linearisability to imply observational refinement: informally, during the interval of time between $(t_2, \mathsf{ret!}\, m_2(z_2'))$ and $(t_3, \mathsf{call?}\, m_4(z_4))$ in an execution of a program producing $h_1$, both threads $t_2$ and $t_3$ execute pieces of client code, which can communicate via the client memory. To preserve the behaviour of the client when replacing the concrete library in the program by an abstract one in observational refinement, this communication must not be affected, and, for this, the abstract library has to admit a history in which the order between the above actions is preserved.

When $h_1, h_2 : M \to M'$ correspond to a non-ground parameterised library, i.e., $M \neq \emptyset$, a similar situation arises with communication between the methods of the parameter library executing in different threads. For this reason, our generalisation of linearisability requires preserving the order between call! and ret? actions, such as

$(t_2, \mathsf{call!}\, m_b(z_b))$ and $(t_1, \mathsf{ret?}\, m_a(z'_a))$ in Figure 4; this requirement is dual to the one considered in classical linearisability. It is not enough, however. Definition 2 also requires preserving the order between call! and call?, as well as ret! and ret? actions, e.g., $(t_3, \mathsf{ret!}\, m_3(z'_3))$ and $(t_2, \mathsf{ret?}\, m_b(z'_b))$ in Figure 4. In the case when $M \cap M' \neq \emptyset$, this is also required to validate observational refinement. For example, during the interval of time between $(t_3, \mathsf{ret!}\, m_3(z'_3))$ and $(t_2, \mathsf{ret?}\, m_b(z'_b))$ in an execution producing $h_1$, the client code in thread $t_3$ can call a method $m'_b \in M \cap M'$ of the parameter library (cf. the dashed arrows in Figure 3). The code of the method $m'_b$ executed by $t_3$ can then communicate with that of the method $m_b$ executed by $t_2$, and to preserve this communication, we need to preserve the order between $(t_3, \mathsf{ret!}\, m_3(z'_3))$ and $(t_2, \mathsf{ret?}\, m_b(z'_b))$.

In §5 and §6 we prove that the above notion of linearisability indeed validates observational refinement. If the library $L : M \to M'$ producing the histories $h_1, h_2$ in Definition 2 is such that $M \cap M' = \emptyset$, then the client cannot directly call methods of its parameter library, and, as we show, parameterised linearisability can be weakened without invalidating observational refinement.

DEFINITION 3 (Parameterised linearisability: encapsulated case) *For $h_1, h_2 : M \to M'$ with $M \cap M' = \emptyset$ we let $h_1 \sqsubseteq_{\mathsf{e}} h_2$ if there exists a permutation $\pi : \mathbb{N} \to \mathbb{N}$ such that*

$$\forall i.\, h_1(i) = h_2(\pi(i)) \wedge (\forall j.\, i < j \wedge ((\exists t.\, h_1(i) = (t, -) \wedge h_2(j) = (t, -)) \vee (h_1(i), h_1(j)) \in (\mathsf{ActRet!} \times \mathsf{ActCall?}) \cup (\mathsf{ActCall!} \times \mathsf{ActRet?})) \implies \pi(i) < \pi(j)).$$

Since this definition does not take into account the order between $(t_1, \mathsf{call!}\, m_a(z_a))$ and $(t_3, \mathsf{call?}\, m_3(z_3))$ in $h_2$ from Figure 4, we have $h_2 \sqsubseteq_{\mathsf{e}} h_1$ even though $h_2 \not\sqsubseteq h_1$.

Definitions 2 and 3 do not make any assumptions about the implementation of the parameter library. However, sometimes the correctness of a parameterised library can only be established under certain assumptions about the behaviour of its parameter. In particular, this is the case for the flat combining library from §2. In its implementation FC from Figure 1, a request by a thread $t$ to execute a method $m_i$ of the parameter library can be fulfilled by another thread $t'$ who happens to act as a combiner; in contrast, the specification $\mathsf{FC}^\sharp$ in Figure 2 pretends that $m_i$ is executed in the requesting thread. Thus, FC and $\mathsf{FC}^\sharp$ will behave differently if we supply as their parameter a library whose methods depend on the identifiers of executing threads (e.g., with $m_i$ implemented as "return mytid()"). As a consequence, FC does not simulate $\mathsf{FC}^\sharp$. On the other hand, this will be the case if we restrict ourselves to parameter libraries whose behaviour is independent of thread identifiers. The following version of parameterised linearisability allows us to use such assumptions, formulated as closure properties on histories of interactions between a parameterised library and its parameter. Given a history $h$, let $\overline{h}$ be the history obtained by swapping ! and ? actions in $h$.

DEFINITION 4 (Up-to linearisability) *For $h_1, h_2 : M \to M'$ such that $M \cap M' = \emptyset$ and a binary relation $\mathcal{R}$ on histories of type $\emptyset \to M$, we say that $h_1$ **is linearised by** $h_2$ **up to** $\mathcal{R}$, written $h_1 \sqsubseteq_{\mathcal{R}} h_2$, if $(h_1|_{\mathsf{ClAct}}) \sqsubseteq (h_2|_{\mathsf{ClAct}})$ and $\overline{(h_1|_{\mathsf{AbsAct}})}\ \mathcal{R}\ \overline{(h_2|_{\mathsf{AbsAct}})}$.*

For flat combining, a suitable relation $\mathcal{R}_{\mathsf{t}}$ relates two histories if one can be obtained from the other by replacing thread identifiers of some pairs of a call and a corresponding (if any) return action. There are other useful choices of $\mathcal{R}$, such as equivalence up to commuting abstract method invocations [7].

So far we have defined our notions of linearisability abstractly, on sets of histories. We next introduce a language for parameterised libraries and show how to generate sets

of histories of a library in this language. This lets us lift the notion of linearisability to libraries and prove that FC in Figure 1 is indeed linearised up to $\mathcal{R}_{\mathsf{t}}$ by $\mathsf{FC}^\sharp$ in Figure 2.

## 4 Lifting Linearisability to Libraries

**Library Syntax.** We use the following language to define libraries:

$$L \ ::= \ \langle \mathsf{public} : B; \mathsf{private} : B \rangle \qquad B \ ::= \ \varepsilon \ \mid \ (m \Leftarrow C); B \ \mid \ (\mathsf{abstract}\ m); B$$
$$C \ ::= \ c \ \mid \ m() \ \mid \ C; C \ \mid \ \mathsf{if}(E)\ \mathsf{then}\ C\ \mathsf{else}\ C \ \mid \ \mathsf{while}(E)\ C$$

A parameterised library $L$ is a collection of methods, some implemented by commands $C$ and others declared as abstract, meant to be implemented by a parameter library. Methods can be public or private, with only the former made available to clients. In §5 and §6 we extend the language to complete programs, consisting of a multithreaded client using a parameterised library with its parameter instantiated. In particular, we introduce private methods here to define parameter library instantiation in §5.

In commands, $c$ ranges over ***primitive commands*** from a set $\mathsf{PComm}$, and $E$ over expressions, whose set we leave unspecified. The command $m()$ invokes the method $m$; it does not mention its parameter or return value, since, as we explain below, these are passed via dedicated thread-local memory locations. We consider only well-formed libraries where a method is declared at most once and every method called is declared. We identify libraries up to the order of method declarations and $\alpha$-renaming of private non-abstract methods. For a library $L = \langle \mathsf{public} : B_{\mathsf{pub}}; \mathsf{private} : B_{\mathsf{pvt}} \rangle$ we have $L : \mathsf{Abs}(L) \to \mathsf{Pub}(L)$, where $\mathsf{Pub}(L)$ is the set of methods declared in $B_{\mathsf{pub}}$, and $\mathsf{Abs}(L)$ of those declared as abstract in $B_{\mathsf{pub}}$ or $B_{\mathsf{pvt}}$.

**Linearisability on Libraries and the Semantics Idea.** We now show how to generate the set of histories $[\![L]\!] \in 2^{\mathsf{Hist}}$ of a library $L$. Then we let a library $L_1$ be ***linearised by*** a library $L_2$, written $L_1 \sqsubseteq L_2$, if $[\![L_1]\!] \sqsubseteq [\![L_2]\!]$; similarly for $\sqsubseteq_{\mathsf{e}}$ and $\sqsubseteq_{\mathcal{R}}$.

We actually generate all library *traces*, which, unlike histories, also record its internal actions. Let us extend the set of actions $\mathsf{Act}$ with elements of the forms $(t, c)$ for $c \in \mathsf{PComm}$, $(t, \mathsf{call}\, m(z))$ and $(t, \mathsf{ret}\, m(z))$, leading to a set $\mathsf{TrAct}$. The latter two kinds of actions correspond to calls and returns between methods implemented inside the library. A ***trace*** $\tau$ is a finite sequence of elements in $\mathsf{TrAct}$; we let $\mathsf{Traces} = \mathsf{TrAct}^*$.

The denotation $[\![L]\!]$ of a library $L : M \to M'$ includes the histories extracted from traces that $L$ produces in any possible environment, i.e., assuming that client threads perform any sequences of calls to methods in $M'$ with arbitrary parameter values and that abstract methods in $M$ return arbitrary values. The definition of $[\![L]\!]$ follows the intuitive semantics of our programming language. An impatient reader can skip it on first reading and jump directly to Theorem 1 at the end of this section.

**Heaps and Primitive Command Semantics.** Let $\mathsf{Locs}$ be the set of memory locations. As we noted in §3, we impose a standard restriction that different libraries and their clients access different sets of memory locations, except the ones used for method parameter passing. Formally, we assume that each library $L$ is associated with a set of its locations $\mathsf{Locs}_L \subseteq \mathsf{Locs}$. The state of $L$ is thus given by a ***heap*** $\sigma \in \mathsf{Locs}_L \to \mathbb{Z}$. We assume a special subset of locations $\{\mathsf{arg}_t\}_{t \in \mathcal{T}}$ belonging to every $\mathsf{Locs}_L$, which we use to pass parameters and return values for method invocations in thread $t$.

We assume that the execution of primitive commands and the evaluation of expressions are atomic. The semantics of a primitive command $c \in \mathsf{PComm}$ used by a

**Traces of commands** $(\!|C|\!)_t : (\mathcal{M} \times \mathcal{T} \to 2^{\mathsf{Traces}}) \to 2^{\mathsf{Traces}}$

$(\!|c|\!)_t\eta = \{(t, c)\}$ $\qquad (\!|C_1; C_2|\!)_t\eta = \{\tau_1\tau_2 \mid \tau_1 \in (\!|C_1|\!)_t\eta \wedge \tau_2 \in (\!|C_2|\!)_t\eta\}$

$(\!|\mathsf{if}(E)\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2|\!)_t\eta = (t, \mathsf{assume}(E))\,((\!|C_1|\!)_t\eta) \cup (t, \mathsf{assume}(!E))((\!|C_2|\!)_t\eta)$

$(\!|\mathsf{while}(E)\ C|\!)_t\eta = ((t, \mathsf{assume}(E))([\![C]\!]\eta))^*(t, \mathsf{assume}(!E))$

$$(\!|m()|\!)_t\eta = \begin{cases} \{(t, \mathsf{call!}\, m(z))\,\tau\,(t, \mathsf{ret?}\, m(z')) \mid \tau \in \eta(m,t) \wedge z, z' \in \mathbb{Z}\}, & \text{if } m \in M \\ \{(t, \mathsf{call}\, m(z))\,\tau\,(t, \mathsf{ret}\, m(z')) \mid \tau \in \eta(m,t) \wedge z, z' \in \mathbb{Z}\}, & \text{otherwise} \end{cases}$$

**Traces of library bodies**

$\mathcal{F} : (\mathcal{M} \times \mathcal{T} \to 2^{\mathsf{Traces}}) \to (\mathcal{M} \times \mathcal{T} \to 2^{\mathsf{Traces}})$

$$(\mathcal{F}(\eta))(m,t) = \begin{cases} \eta(m,t) \cup ((\!|C|\!)_t\eta), & \text{if } (m \Leftarrow C) \text{ appears in } L \\ \{\varepsilon\}, & \text{if } m \in M \\ \emptyset, & \text{otherwise} \end{cases}$$

$(\!|B|\!) : \mathcal{M} \times \mathcal{T} \to 2^{\mathsf{Traces}}$

$(\!|B_{\mathsf{pub}}; B_{\mathsf{pvt}}|\!) = \mathsf{lfp}(\mathcal{F})$

**Traces of libraries** $(\!|L : M \to M'|\!) : 2^{\mathsf{Traces}}$

$$(\!|L|\!) = \mathsf{prefix}\left(\bigcup\nolimits_{k>0} \Big\|_{t=1}^{k} \left(\bigcup\nolimits_{\substack{z,z' \in \mathbb{Z} \\ m \in M' \setminus M}} (t, \mathsf{call?}\, m(z))\,((\!|B_{\mathsf{pub}}; B_{\mathsf{pvt}}|\!)(m,t))\,(t, \mathsf{ret!}\, m(z'))\right)^*\right)$$

**Figure 5.** Possible traces of a library $L = \langle \mathsf{public} : B_{\mathsf{pub}}; \mathsf{private} : B_{\mathsf{pvt}} \rangle : M \to M'$. Here $\Big\|_{t=1}^{k} T_t$ denotes the set of all interleavings of traces from the sets $T_1, \ldots, T_k$.

$\sigma \rightsquigarrow^L_{\mathsf{call}\ m(z),t} \sigma'$ iff $\sigma' = \sigma, \sigma(\mathsf{arg}_t) = z$
$\sigma \rightsquigarrow^L_{\mathsf{call?}\ m(z),t} \sigma'$ iff $\sigma' = \sigma[\mathsf{arg}_t \mapsto z]$
$\sigma \rightsquigarrow^L_{\mathsf{call!}\ m(z),t} \sigma'$ iff $\sigma' = \sigma, \sigma(\mathsf{arg}_t) = z$
$\sigma \rightsquigarrow^L_{\mathsf{ret}\ m(z),t} \sigma'$ iff $\sigma' = \sigma, \sigma(\mathsf{arg}_t) = z$
$\sigma \rightsquigarrow^L_{\mathsf{ret!}\ m(z),t} \sigma'$ iff $\sigma' = \sigma, \sigma(\mathsf{arg}_t) = z$
$\sigma \rightsquigarrow^L_{\mathsf{ret?}\ m(z),t} \sigma'$ iff $\sigma' = \sigma[\mathsf{arg}_t \mapsto z]$

**Figure 6.** Transformers for calls and returns to, from and inside a library $L$.

library $L$ is defined by a family of transformers $\{\rightsquigarrow^L_{c,t}\}_{t \in \mathcal{T}}$, where $\rightsquigarrow^L_{c,t} \subseteq (\mathsf{Locs}_L \to \mathbb{Z}) \times (\mathsf{Locs}_L \to \mathbb{Z})$ describes how $c$ affects the state of the library. The fact that the transformers are defined on locations from $\mathsf{Locs}_L$ formalises our assumption that $L$ accesses only these locations. We assume that the transformers satisfy some standard properties [15], deferred to [1, §A] due to space constraints. To define the semantics of expressions, we assume that for each $E$ the set $\mathsf{PComm}$ contains a special command $\mathsf{assume}(E)$, used only in defining the semantics, that allows the computation to proceed only if $E$ is non-zero: $\sigma \rightsquigarrow^L_{\mathsf{assume}(E),t} \sigma'$ iff $\sigma' = \sigma$ and $E$ is non-zero in $\sigma$.

**Library Denotations.** The set of traces of a library is generated in two stages. First, we generate a superset $(\!|L|\!) \subseteq 2^{\mathsf{Traces}}$ of traces produced by $L$, defined in Figure 5. If we think of commands as control-flow graphs, these traces contain interleavings of all possible paths through the control-flow graphs of $L$'s methods, invoked in an arbitrary sequence. We then select those traces in $(\!|L|\!)$ that correspond to valid executions starting in a given heap using a predicate $[\![\tau]\!]_L : (\mathsf{Locs}_L \to \mathbb{Z}) \to \{\mathsf{true}, \mathsf{false}\}$. We define $[\![\cdot]\!]_L$ by generalising $\rightsquigarrow$ to calls and returns as shown in Figure 6 and letting

$$[\![\varepsilon]\!]_L\sigma = \mathsf{true}; \quad [\![(t,a)\,\tau]\!]_L\sigma = \mathsf{if}\ (\exists\sigma'.\, \sigma \rightsquigarrow^L_{a,t} \sigma' \wedge [\![\tau]\!]_L\sigma' = \mathsf{true})\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}.$$

Finally, we let the set of histories $[\![L]\!]$ of a library $L$ consist of those obtained from traces representing its valid executions from a heap with all locations set to 0:

$$[\![L]\!] = \mathsf{history}(\{\tau \in (\!|L|\!) \mid [\![\tau]\!]_L(\lambda x \in \mathsf{Locs}_L.\, 0) = \mathsf{true}\}),$$

where history projects to actions in Act.

THEOREM 1 (Correctness of flat combining) *For the libraries* $\mathsf{FC}$ *in Figure 1 and* $\mathsf{FC}^\sharp$ *in Figure 2 and the relation* $\mathcal{R}_\mathsf{t}$ *from §3 we have* $\mathsf{FC} \sqsubseteq_{\mathcal{R}_\mathsf{t}} \mathsf{FC}^\sharp$.

PROOF SKETCH. Consider $h \in [\![\mathsf{FC}]\!]$. In such a history, any invocation of an abstract method $(t, \mathsf{call}!\, m_i(z_i))\,(t, \mathsf{ret}?\, m_i(z_i'))$ happens within the execution of the corresponding wrapper method $(t', \mathsf{call}?\, \mathsf{do\_}m_i(z_i))\,(t', \mathsf{ret}!\, \mathsf{do\_}m_i(z_i'))$ (or just $(t', \mathsf{call}?\, \mathsf{do\_}m_i(z_i))$ if the execution of the method is uncompleted in $h$), though not necessarily in the same thread. This correspondence is one-to-one, as different invocations of abstract methods correspond to different requests to perform them. Furthermore, abstract methods in $h$ are executed sequentially. We then construct a history $h'$ by replacing every abstract method call $(t, \mathsf{call}!\, m_i(z_i))\,(t, \mathsf{ret}?\, m_i(z_i'))$ in $h|_{\mathsf{AbsAct}}$ by

$$(t', \mathsf{call}?\, \mathsf{do\_}m_i(z_i))\,(t', \mathsf{call}!\, m_i(z_i))\,(t', \mathsf{ret}?\, m_i(z_i'))\,(t', \mathsf{ret}!\, \mathsf{do\_}m_i(z_i')),$$

where $t'$ is the thread identifier of the corresponding wrapper method invocation (similarly for uncompleted invocations). It is easy to see that $\overline{(h|_{\mathsf{AbsAct}})}\ \mathcal{R}_\mathsf{t}\ \overline{(h'|_{\mathsf{AbsAct}})}$ and $h' \in [\![\mathsf{FC}^\sharp]\!]$. Since the execution of an abstract method in $h$ happens within the execution of the corresponding wrapper method, we also have $(h|_{\mathsf{ClAct}}) \sqsubseteq (h'|_{\mathsf{ClAct}})$. □

## 5 Instantiating Library Parameters and Contextuality

We now define how library parameters are instantiated and show that our notions of linearisability are preserved under such instantiations. To this end, we introduce a partial operation $\circ$ on libraries of §4: informally, for $L_1 : M \to M'$ and $L_2 : M' \to M''$ the library $L_2 \circ L_1 : M \to M''$ is obtained by instantiating abstract methods in $L_2$ with their implementations from $L_1$. Note that $L_1$ can itself have abstract methods $M$, which are left unimplemented in $L_2 \circ L_1$. Since we assume that different libraries operate in disjoint address spaces, for $\circ$ to be defined we require that the sets of locations of $L_1$ and $L_2$ be disjoint, with the exception of those used for method parameter passing. To avoid name clashes, we also require that public non-abstract methods of $L_2$ not be declared as abstract in $L_1$ (private non-abstract methods are not an issue, since we identify libraries up to their $\alpha$-renaming); this also disallows recursion between $L_2$ and $L_1$.

DEFINITION 5 (Parameter library instantiation) *Consider* $L_1 : M \to M'$ *and* $L_2 : M' \to M''$ *such that* $(M'' \setminus M') \cap M = \emptyset$ *and* $\mathsf{Locs}_{L_1} \cap \mathsf{Locs}_{L_2} = \{\mathsf{arg}_t\}_{t \in \mathcal{T}}$. *Then* $L_2 \circ L_1 : M \to M''$ *is the library with* $\mathsf{Locs}_{L_2 \circ L_1} = \mathsf{Locs}_{L_1} \cup \mathsf{Locs}_{L_2}$ *obtained by erasing the declarations for methods in* $M'$ *from* $L_2$, *reclassifying the methods from* $M' \setminus M''$ *in* $L_1$ *as private, and concatenating the method declarations of the resulting two libraries. We write* $(L_2 \circ L_1)\!\downarrow$ *when* $L_2 \circ L_1$ *is defined.*

We now show that the notions of parameterised linearisability we proposed are ***contextual***, i.e., closed under library instantiations. This property is useful in that it allows us to break the reasoning about a complex library into that about individual libraries from which it is constructed. As we show in §6, contextuality also helps us establish observational refinement.

THEOREM 2 (Contextuality of parameterised linearisability: general case) *For* $L_1, L_2 : M \to M'$ *such that* $L_1 \sqsubseteq L_2$*:*

*(i)* $\forall L : M'' \to M.\, (L_1 \circ L)\!\downarrow \wedge\, (L_2 \circ L)\!\downarrow \implies L_1 \circ L \sqsubseteq L_2 \circ L.$

*(ii)* $\forall L : M' \to M''.\, (L \circ L_1)\!\downarrow \wedge\, (L \circ L_2)\!\downarrow \implies L \circ L_1 \sqsubseteq L \circ L_2.$

THEOREM 3 (Contextuality of parameterised linearisability: encapsulated case) *For $L_1, L_2 : M \to M'$ such that $M \cap M' = \emptyset$ and $L_1 \sqsubseteq_{\mathsf{e}} L_2$:*

*(i)* $\forall L : M'' \to M.\, (L_1 \circ L)\!\downarrow \wedge\, (L_2 \circ L)\!\downarrow \implies L_1 \circ L \sqsubseteq_{\mathsf{e}} L_2 \circ L.$
*(ii)* $\forall L : M' \to M''.\, (L \circ L_1)\!\downarrow \wedge\, (L \circ L_2)\!\downarrow \implies L \circ L_1 \sqsubseteq_{\mathsf{e}} L \circ L_2.$

The restriction on method names in Definition 5 ensures that the library compositions in Theorem 3 have no public abstract methods and can thus be compared by $\sqsubseteq_{\mathsf{e}}$. Note that if $L$ is ground, then so are $L_1 \circ L$ and $L_2 \circ L$. In this case, Theorems 2(i) and 3(i) allow us to establish classical linearisability from parameterised one.

Stating the contextuality of $\sqsubseteq_{\mathcal{R}}$ is more subtle. The relationship $L_1 \sqsubseteq_{\mathcal{R}} L_2$ allows the use of abstract methods by $L_1$ and $L_2$ to differ according to $\mathcal{R}$. As a consequence, for a non-ground parameter library $L$, their use by $L_1 \circ L$ and $L_2 \circ L$ may also differ according to another relation $\mathcal{G}$. We now introduce a property of $L$ ensuring that a change in $L$'s interactions with its client according to $\mathcal{R}$ (the *rely*) leads to a change in $L$'s interactions with its abstract methods according to $\mathcal{G}$ (the *guarantee*).

DEFINITION 6 (Rely-guarantee closure) *Let $\mathcal{R}, \mathcal{G}$ be relations between histories of type $\emptyset \to M'$ and $\emptyset \to M$, respectively. A library $L : M \to M'$ is $\binom{\mathcal{R}}{\mathcal{G}}$-**closed** if for all $h \in [\![L]\!]$ and $h' : \emptyset \to M'$ we have*

$$(h|_{\mathsf{ClAct}})\ \mathcal{R}\ h' \implies \exists h'' \in [\![L]\!].\, (h''|_{\mathsf{ClAct}} = h') \wedge \overline{(h|_{\mathsf{AbsAct}})}\ \mathcal{G}\ \overline{(h''|_{\mathsf{AbsAct}})}.$$

Due to space constraints, we state contextuality of $\sqsubseteq_{\mathcal{R}}$ only for the case in which library parameters do not have public abstract methods. A more general statement which relaxes this assumption is given in [1, §B].

THEOREM 4 (Contextuality of linearisability up to $\mathcal{R}$) *For $L_1, L_2 : M \to M'$ such that $M \cap M' = \emptyset$ and a relation $\mathcal{R}$ such that $L_1 \sqsubseteq_{\mathcal{R}} L_2$:*

*(i)* $\forall L : M'' \to M.\, \forall \mathcal{G}.\, M'' \cap M = \emptyset \wedge (L \text{ is } \binom{\mathcal{R}}{\mathcal{G}}\text{-closed}) \wedge$
$(L_1 \circ L)\!\downarrow \wedge\, (L_2 \circ L)\!\downarrow \implies L_1 \circ L \sqsubseteq_{\mathcal{G}} L_2 \circ L.$
*(ii)* $\forall L : M' \to M''.\, (L \circ L_1)\!\downarrow \wedge\, (L \circ L_2)\!\downarrow \implies L \circ L_1 \sqsubseteq_{\mathcal{R}} L \circ L_2.$

When $L$ in Theorem 4(i) is ground, $\mathcal{G}$ becomes irrelevant. In this case we say that $L$ is $\mathcal{R}$-***closed*** if it is $\binom{\mathcal{R}}{\{(\varepsilon,\varepsilon)\}}$-closed. Hence, from Theorems 1 and 4(i) we get that for any $\mathcal{R}_{\mathsf{t}}$-closed (§3) library $L$ we have $\mathsf{FC} \circ L \sqsubseteq \mathsf{FC}^\sharp \circ L$: instantiating flat combining with a library insensitive to thread identifiers, e.g., a sequential stack or a queue, yields a concurrent library linearisable in the classical sense.

Given two libraries $L_1 : M_1 \to M_1'$ and $L_2 : M_2 \to M_2'$ that do not interact, i.e., $(M_1 \cup M_1') \cap (M_2 \cup M_2') = \emptyset$, we may wish to compose them by merging their method declarations into a library $L_1 \uplus L_2 : M_1 \uplus M_2 \to M_1' \uplus M_2'$, as originally proposed in [8]. Our notions of linearisability are also closed under this composition.

THEOREM 5 (Compositionality of parameterised linearisability) *For $L_1, L_1' : M_1 \to M_1'$ and $L_2, L_2' : M_2 \to M_2'$ such that $(M_1 \cup M_1') \cap (M_2 \cup M_2') = \emptyset$:*

*(i)* $L_1 \sqsubseteq L_1' \wedge L_2 \sqsubseteq L_2' \implies L_1 \uplus L_2 \sqsubseteq L_1' \uplus L_2'.$
*(ii)* $L_1 \sqsubseteq_{\mathsf{e}} L_1' \wedge L_2 \sqsubseteq_{\mathsf{e}} L_2' \implies L_1 \uplus L_2 \sqsubseteq_{\mathsf{e}} L_1' \uplus L_2'.$
*(iii)* $\forall \mathcal{R}, \mathcal{G}.\, L_1 \sqsubseteq_{\mathcal{R}} L_1' \wedge L_2 \sqsubseteq_{\mathcal{G}} L_2' \implies L_1 \uplus L_2 \sqsubseteq_{\mathcal{R} \otimes \mathcal{G}} L_1' \uplus L_2'$, *where $\mathcal{R} \otimes \mathcal{G}$ relates histories if their projections to $M_1$ actions are related by $\mathcal{R}$ and the projections to $M_2$ actions are related by $\mathcal{G}$.*

## 6 Clients and Observational Refinement

A ***program*** $P$ has the form $\mathsf{let}\ L\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n$, where $L : \emptyset \to M$ is a ground library and $C_1 \parallel \ldots \parallel C_n$ is a client such that $C_1, \ldots, C_n$ call only methods in $M$, written $(C_1 \parallel \ldots \parallel C_n) : M$. Using the contextuality results from §5, we now show that our notions of linearisability imply observational refinement for such programs.

The semantics of a program $P$ is given by the set of its traces $[\![P]\!] \in 2^{\mathsf{Traces}}$, which include actions $(t, c)$ recording the execution of primitive commands $c$ by client threads $C_t$ and the library $L$, as well as $(t, \mathsf{call}\, m(z))$ and $(t, \mathsf{ret}\, m(z))$ actions corresponding to the former invoking methods of the latter. The semantics $[\![P]\!]$ is defined similarly to that of libraries in §4. In particular, we assume that client threads $C_t$ access only locations in a set $\mathsf{Locs}_{\mathsf{client}}$ such that $\mathsf{Locs}_{\mathsf{client}} \cap \mathsf{Locs}_L = \{\mathsf{arg}_t\}_{t \in \mathcal{T}}$ for any $L$. Due to space constraints, we defer the definition of $[\![P]\!]$ to [1, §A]. We define the ***observable behaviour*** $\mathsf{obs}(\tau)$ of a trace $\tau \in [\![P]\!]$ as its projection to client actions, i.e., those outside method invocations, and lift obs to sets of traces as expected.

DEFINITION 7 (Observational refinement) *For $L_1, L_2 : M \to M'$ we say that $L_1$ **observationally refines** $L_2$, written $L_1 \sqsubseteq_{\mathsf{obs}} L_2$, if for any ground library $L : \emptyset \to M$ and client $(C_1 \parallel \ldots \parallel C_n) : M'$ we have*

$$\mathsf{obs}([\![\mathsf{let}\ (L_1 \circ L)\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n]\!]) \subseteq \mathsf{obs}([\![\mathsf{let}\ (L_2 \circ L)\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n]\!]).$$

*For a binary relation $\mathcal{R}$ on histories we say that $L_1$ **observationally refines** $L_2$ **up to** $\mathcal{R}$, written $L_1 \sqsubseteq^{\mathcal{R}}_{\mathsf{obs}} L_2$, if the above is true under the assumption that $L$ is $\mathcal{R}$-closed.*

Thus, $L_1 \sqsubseteq_{\mathsf{obs}} L_2$ means that $L_1$ can be replaced by $L_2$ in any program that uses it while keeping observable behaviours reproducible. This allows us to check a property of a program using $L_1$ (e.g., the flat combining implementation in Figure 1) by checking this property on a program with $L_1$ replaced by a possibly simpler $L_2$ (e.g., the flat combining specification in Figure 2). Using Theorems 2–4, we can show that our notions of linearisability validate observational refinement.

THEOREM 6 (Observational refinement) *For any libraries $L_1, L_2 : M \to M'$:*

*(i)* $L_1 \sqsubseteq L_2 \implies L_1 \sqsubseteq_{\mathsf{obs}} L_2$.
*(ii)* $M \cap M' = \emptyset \wedge L_1 \sqsubseteq_{\mathsf{e}} L_2 \implies L_1 \sqsubseteq_{\mathsf{obs}} L_2$.
*(iii)* $\forall \mathcal{R}.\, M \cap M' = \emptyset \wedge L_1 \sqsubseteq_{\mathcal{R}} L_2 \implies L_1 \sqsubseteq^{\mathcal{R}}_{\mathsf{obs}} L_2$.

## 7 Related Work

Linearisability has recently been extended to handle liveness properties, ownership transfer and weak memory models [4,5,10]. Most of these extensions have exploited the connection between linearisability and observational refinement [2]. The same methodology is adopted in the present work, but for studying two previously unexplored topics: parameterised libraries and the impact that common restrictions on their contexts have on the definition of linearisability. We believe that our results are compatible with the existing ones and can thus be extended to cover liveness and ownership transfer [4,5].

Our work shares techniques with game semantics of concurrent programming languages [12,3] and Jeffrey and Rathke's semantics of concurrent objects [11] (in particular, we use the ? and ! notation from the latter). The proofs of our contextuality theorems rely on the fact that library denotations satisfy certain closure properties related to $\sqsubseteq$, $\sqsubseteq_{\mathsf{e}}$

and $\sqsubseteq_{\mathcal{R}}$, which are similar to those exploited in these prior works. However, there are two important differences. First, prior work has not studied common restrictions on library contexts (such as the encapsulation and closure conditions in Definitions 3 and 4) and the induced stronger notions of refinement between libraries, the two key topics of this paper. Second, prior works have considered all higher-order functions, while our parameterised libraries are limited to second order. Our motivation for constraining the setting in this way is to use a simple semantics and study the key issues involved in linearisability of parameterised libraries without using sophisticated machinery from game semantics, such as justification pointers and views [9], designed for accurately modelling higher-order features. However, it is definitely a promising direction to look for appropriate notions of linearisability for full higher-order concurrent libraries by combining the ideas from this paper with those from game semantics.

Turon et al. proposed CaReSL [14], a logic that allows proving observational refinements between higher-order concurrent programs directly, without going via linearisability. Their work is complimentary to ours: it provides efficient proof techniques, whereas we identify obligations to prove, independent of a particular proof system.

**Acknowledgements.** We thank Thomas Dinsdale-Young and Ilya Sergey for comments that helped improve the paper. This work was supported by the EU FET project ADVENT.

## References

1. Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Parameterised linearisability (extended version). Available from http://software.imdea.org/~gotsman/.
2. Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52), 2010.
3. Dan R. Ghica and Andrzej S. Murawski. Angelic semantics of fine-grained concurrency. *Ann. Pure Appl. Logic*, 151(2-3), 2008.
4. Alexey Gotsman and Hongseok Yang. Liveness-preserving atomicity abstraction. In *ICALP*, 2011.
5. Alexey Gotsman and Hongseok Yang. Linearizability with ownership transfer. In *CONCUR*, 2012.
6. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.
7. Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPOPP*, 2008.
8. Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.
9. J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2), 2000.
10. Radha Jagadeesan, Gustavo Petri, Corin Pitcher, and James Riely. Quarantining weakness - compositional reasoning under relaxed memory models. In *ESOP*, 2013.
11. Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3), 2005.
12. James Laird. A game semantics of idealized CSP. *ENTCS*, 45, 2001.
13. Claudio Russo. The Joins concurrency library. In *PADL*, 2007.
14. Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.
15. Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In *FoSSaCS*, 2002.