HIGHER-ORDER LINEARISABILITY

Andrzej Murawski
University of Warwick

Nikos Tzevelekos
Queen Mary
University of London

Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

$m : \mathsf{unit} \to \mathsf{unit}$ $\qquad$ $m : \mathsf{int} \to \mathsf{int}$

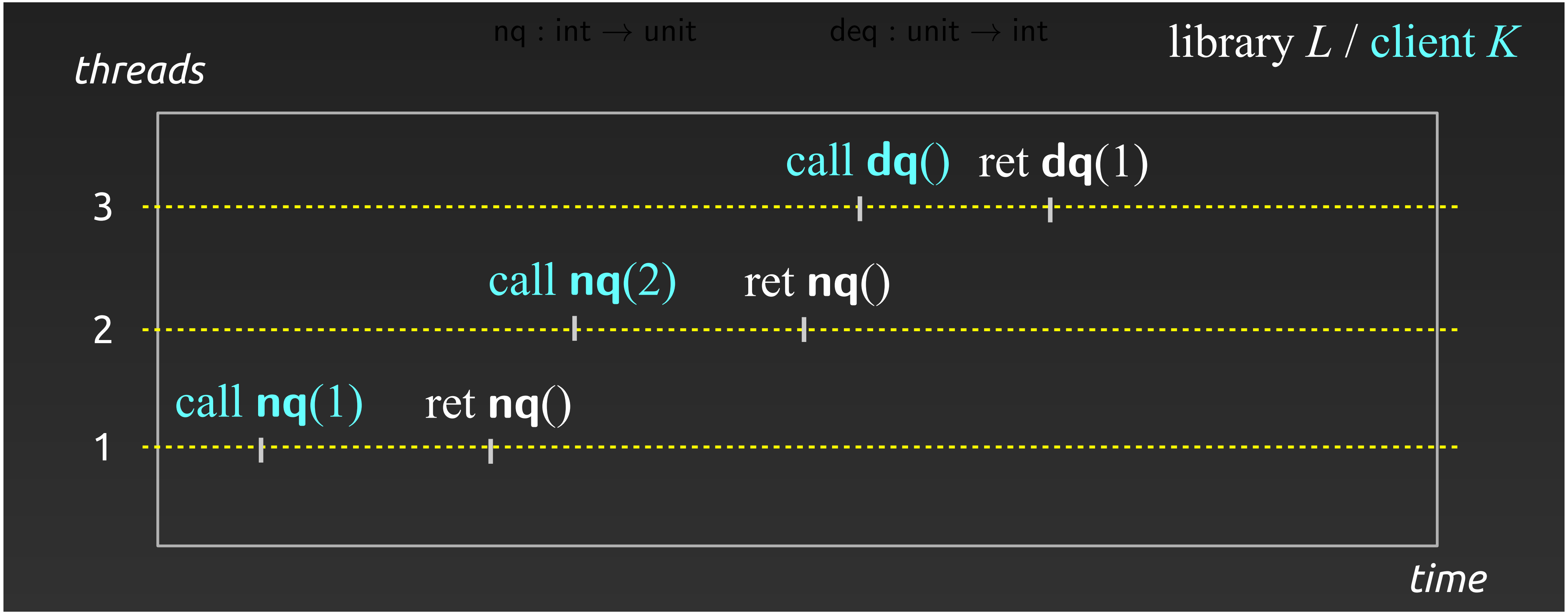
LINEARISABILITY

- **correctness criterion for concurrent libraries**
- **enables proofs of conformance to (sequential) specifications**
- **based on restricted rearrangements of actions in histories**

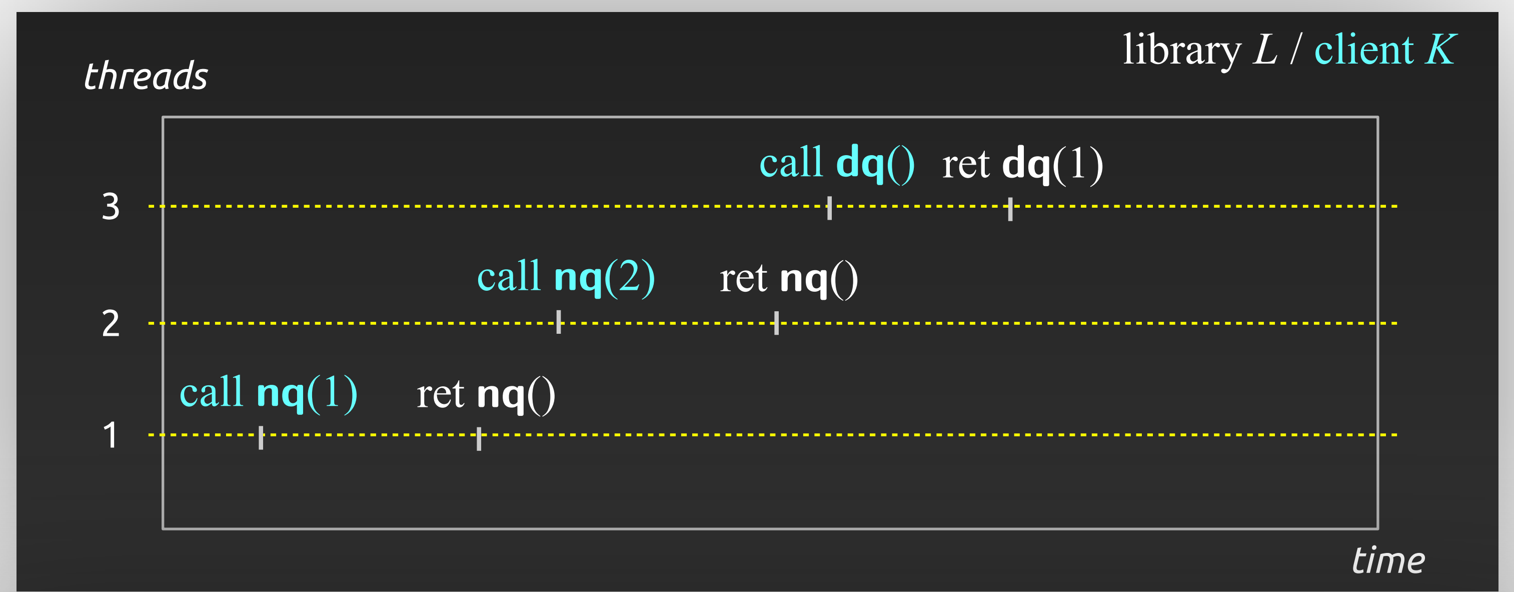
QUEUES

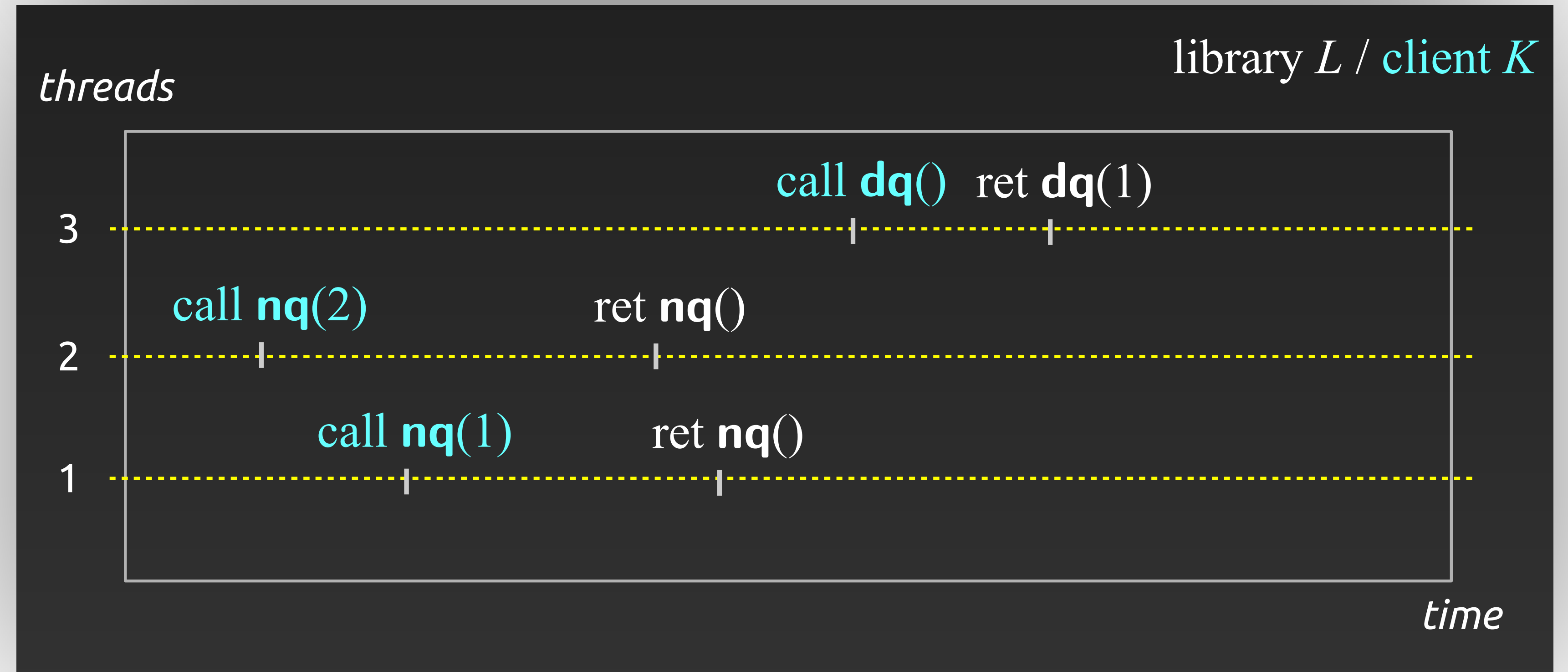
$$\mathsf{nq} : \mathsf{int} \to \mathsf{unit} \qquad\qquad \mathsf{dq} : \mathsf{unit} \to \mathsf{int}$$

$$(1, \mathrm{call}\ \mathsf{nq}(1))\ (1, \mathrm{ret}\ \mathsf{nq}())\ (2, \mathrm{call}\ \mathsf{nq}(2))\ (2, \mathrm{ret}\ \mathsf{nq}())\ (3, \mathrm{call}\ \mathsf{dq}())\ (3, \mathrm{ret}\ \mathsf{dq}(1))$$

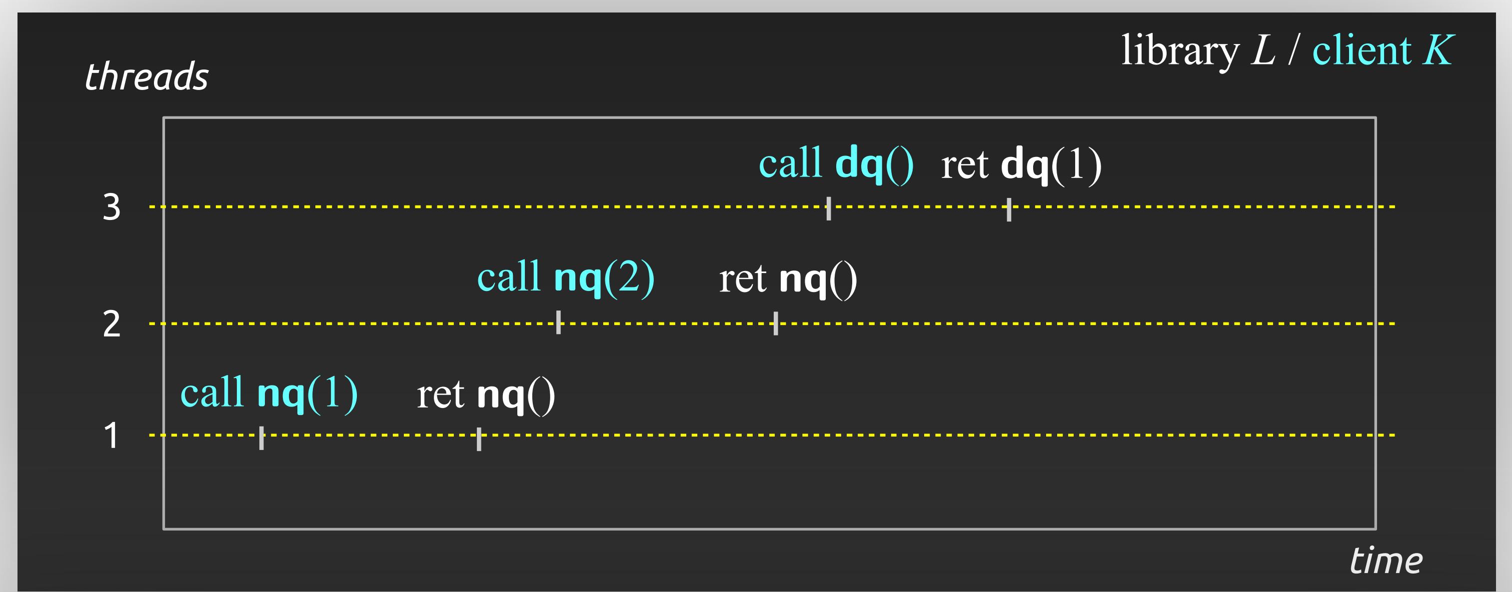


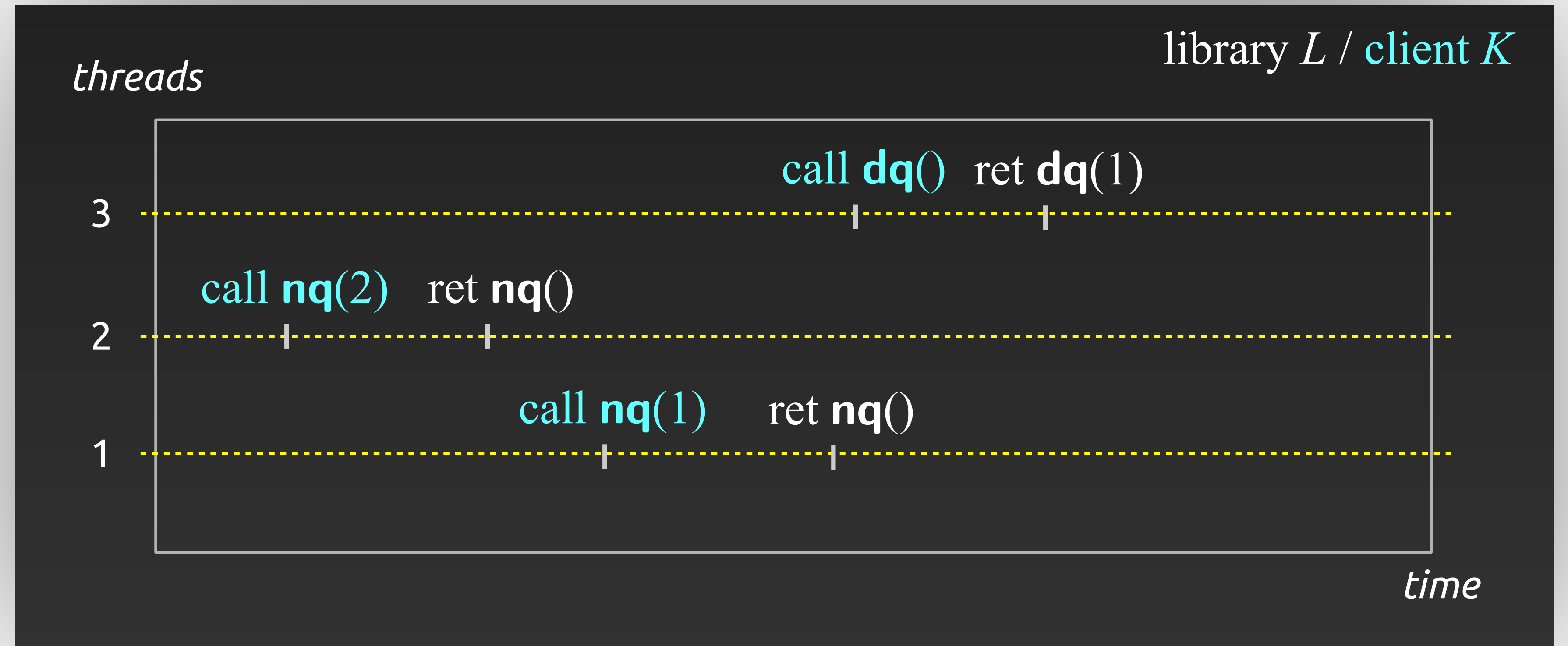
EXAMPLE


threads
library L / client K
call **dq**()
ret **dq**(1)
3
call **nq**(2)
ret **nq**()
2
call **nq**(1)
ret **nq**()
1
time


threads
library L / client K
call **dq**()
ret **dq**(1)
3
call **nq**(2)
ret **nq**()
2
call **nq**(1)
ret **nq**()
1
time

NON-EXAMPLE


library L / client K
threads
call **dq**() ret **dq**(1)
3
call **nq**(2) ret **nq**()
2
call **nq**(1) ret **nq**()
1
time


library L / client K
threads
call **dq**() ret **dq**(1)
3
call **nq**(2) ret **nq**()
2
call **nq**(1) ret **nq**()
1
time

FIRST-ORDER LINEARISABILITY

$t \neq t'$

$$\cdots\ (t, \mathsf{call}\ m(v))\ (t', x')\ \cdots \quad \triangleleft \quad \cdots\ (t', x')\ (t, \mathsf{call}\ m(v))\ \cdots$$

$$\cdots\ (t', x')\ (t, \mathsf{ret}\ m(v))\ \cdots \quad \triangleleft \quad \cdots\ (t, \mathsf{ret}\ m(v))\ (t', x')\ \cdots$$

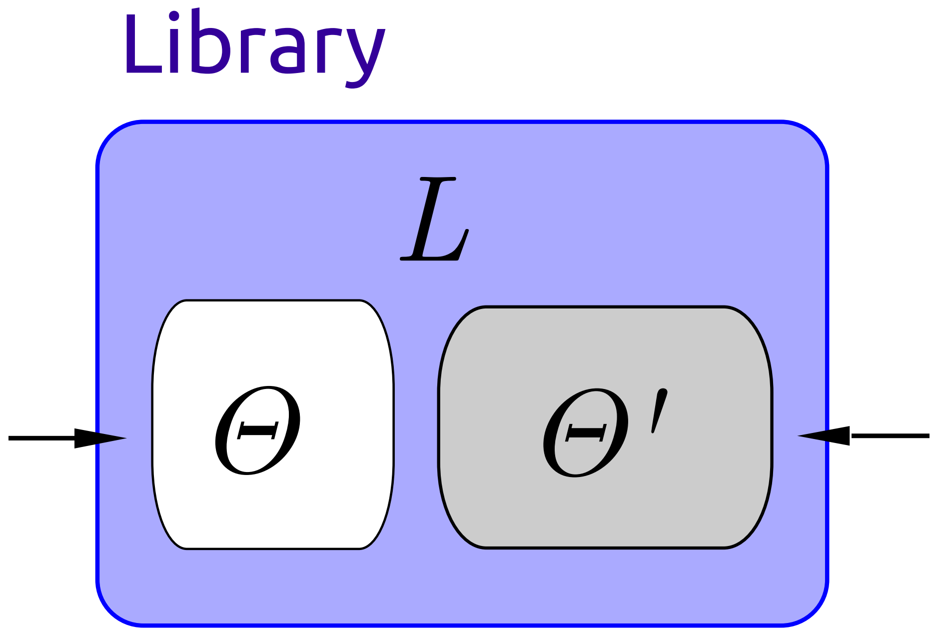
INGREDIENTS

- **histories**
- **sequential histories**
- **linearisability**
- **correctness**

HIGHER-ORDER LIBRARIES

- *higher-order routines (public methods), e.g. HO queue*
- *higher-order parameters (abstract methods), e.g. HO queue with parameter*

HIGHER-ORDER LIBRARY



ESOP'13

Quarantining Weakness*

Compositional Reasoning under Relaxed Memory Models (Extended Abstract)

Radha Jagadeesan[1], Gustavo Petri[2], Corin Pitcher[1], and James Riely[1]

[1] DePaul University
[2] Purdue University

ICALP'14

Parameterised Linearisability

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

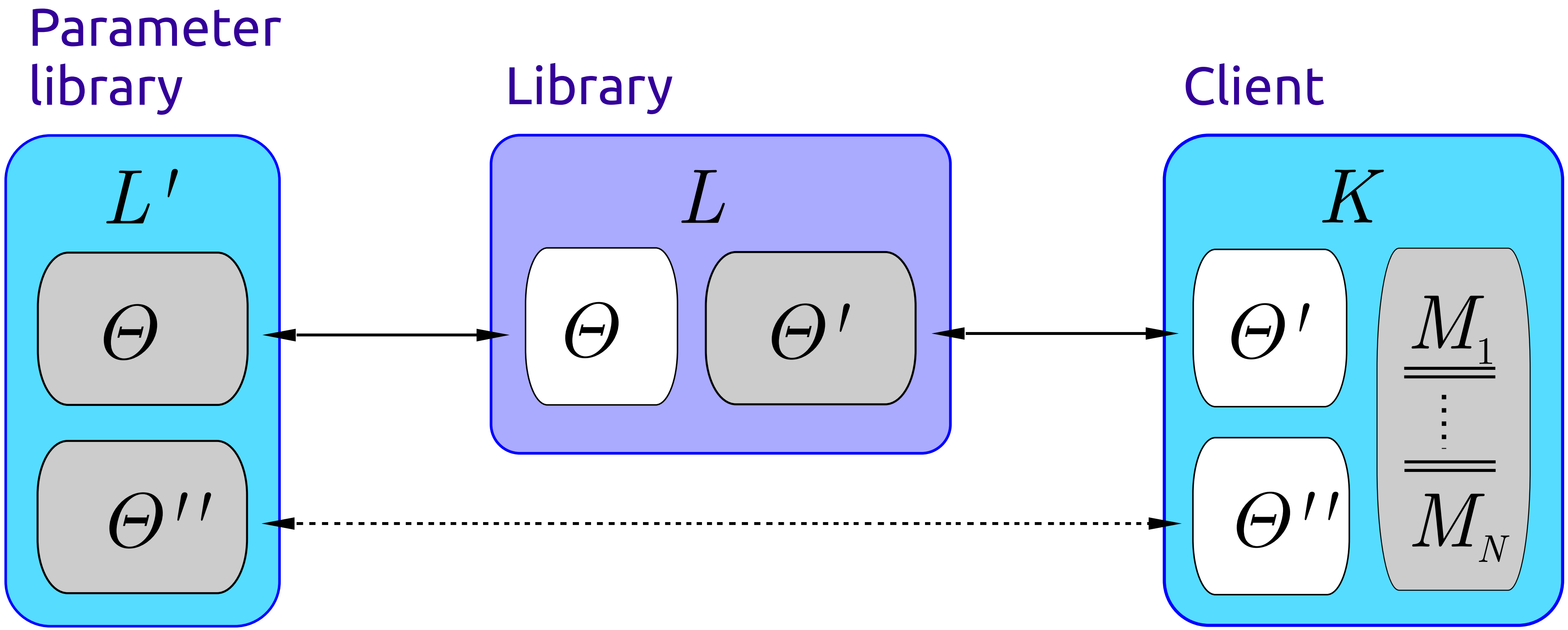
[1] IMDEA Software Institute
[2] University of Oxford

HIGHER-ORDER LIBRARIES

$$
\begin{array}{lrcl}
\textit{Libraries} & L & ::= & B \mid \mathsf{abstract}\; m;\, L \mid \mathsf{public}\; m;\, L \\
\textit{Blocks} & B & ::= & \epsilon \mid m = \lambda x.M;\ B \mid r := \lambda x.M;\ B \mid r := i;\ B \\
\textit{Terms} & M & ::= & () \mid i \mid t_{\mathsf{id}} \mid x \mid m \mid M \oplus M \mid \langle M, M \rangle \mid \pi_1\, M \mid \pi_2\, M \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \\
 & & & \mid \lambda x^{\theta}.M \mid xM \mid mM \mid \mathsf{let}\ x = M\ \mathsf{in}\ M \mid r := M \mid !r
\end{array}
$$

$$
\begin{array}{lrcl}
\textit{Clients} & K & ::= & M \,\|\cdots\|\, M \\
\textit{Values} & v & ::= & () \mid i \mid m \mid \langle v, v \rangle
\end{array}
$$

INTERACTION



HIGHER-ORDER TRACES

- (abstract) interactions of the library with its context
- design guided by game semantics (O-context, P-library)
- example history

$$(1, \text{call}\ \mathsf{m}(\mathsf{m}_1, \mathsf{m}_2))_O \quad (2, \text{call}\ \mathsf{m}(\cdots))_O \quad (2, \text{call}\ \mathsf{m}_1(\cdots))_P \quad (2, \text{ret}\ \mathsf{m}_1(\mathsf{m}_3))_O$$

sequential histories = alternating histories

TRACE SEMANTICS

(INT) $(\mathcal{E}, M, \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \to_t (\mathcal{E}, M', \mathcal{R}', \mathcal{P}, \mathcal{A}, S')$, given that $(M, \mathcal{R}, S) \to_t (M', \mathcal{R}', S')$ and $\mathsf{dom}(\mathcal{R}' \setminus \mathcal{R})$ consists of names that do not occur in $\mathcal{E}, \mathcal{A}$.

(PQY) $(\mathcal{E}, E[mv], \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \xrightarrow{\mathsf{call}\ m(v')_{PY}}_t (m :: E :: \mathcal{E}, -, \mathcal{R}', \mathcal{P}', \mathcal{A}, S)$, given $m \in \mathcal{A}_Y$ and (**PC**).

(OQY) $(\mathcal{E}, -, \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \xrightarrow{\mathsf{call}\ m(v)_{OY}}_t (m :: \mathcal{E}, M\{v/x\}, \mathcal{R}, \mathcal{P}, \mathcal{A}', S)$, given $m \in \mathcal{P}_Y$, $\mathcal{R}(m) = \lambda x.M$ and (**OC**).

(PAY) $(m :: \mathcal{E}, v, \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \xrightarrow{\mathsf{ret}\ m(v')_{PY}}_t (\mathcal{E}, -, \mathcal{R}', \mathcal{P}', \mathcal{A}, S)$, given $m \in \mathcal{P}_Y$ and (**PC**).

(OAY) $(m :: E :: \mathcal{E}, -, \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \xrightarrow{\mathsf{ret}\ m(v)_{OY}}_t (\mathcal{E}, E[v], \mathcal{R}, \mathcal{P}, \mathcal{A}', S)$, given $m \in \mathcal{A}_Y$ and (**OC**).

(**PC**) If v contains the names $m_1, \cdots, m_k$ then $v' = v\{m'_i/m_i \mid 1 \le i \le k\}$ with each m'_i being a fresh name. Moreover, $\mathcal{R}' = \mathcal{R} \uplus \{m'_i \mapsto \lambda x.m_i x \mid 1 \le i \le k\}$ and $\mathcal{P}' = \mathcal{P} \cup_Y \{m'_1, \cdots, m'_k\}$.

(**OC**) If v contains names $m_1, \cdots, m_k$ then $m_i \in \phi(\mathcal{P}, \mathcal{A})$, for each i, and $\mathcal{A}' = \mathcal{A} \cup_Y \{m_1, \cdots, m_k\}$.

Figure 6 Trace semantics rules. The rule (INT) is for embedding internal rules. In the rule (PQY), the library (P) calls one of its abstract methods (either the original ones or those acquired via interaction), while in (PAY) it returns from such a call. The rules (OQY) and (OAY) are dual and represent actions of the context. In all of the rules, whenever we write $m(v)$ or $m(v')$, we assume that the type of v matches the argument type of m.

HIGHER-ORDER LINEARISABILITY

$t \neq t'$

$$\cdots\ (t,x)_O\,(t',x')\ \cdots \quad \lhd \quad \cdots\ (t',x')\,(t,x)_O\ \cdots$$

$$\cdots\ (t',x')\,(t,x)_P\ \cdots \quad \lhd \quad \cdots\ (t,x)_P\,(t',x')\cdots$$

Def. History h_1 *linearises* to h_2 if h_2 can be obtained from h_1 by a series of $\lhd$-transformations.

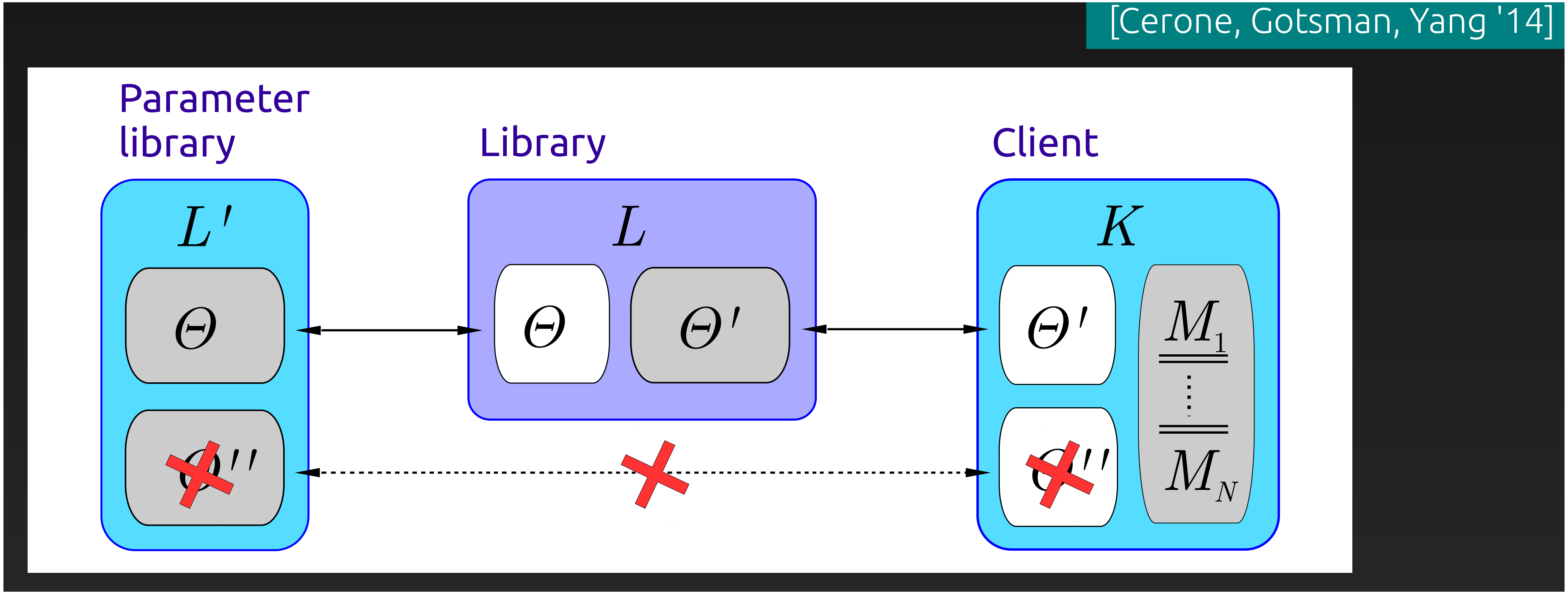
Def. A library L *linearises* to specification S if every history h_1 of L linearises to some h_2 from S.

$$count : \mathsf{int} \to \mathsf{int}, \quad update : (\mathsf{int} \times (\mathsf{int} \to \mathsf{int})) \to \mathsf{int}$$

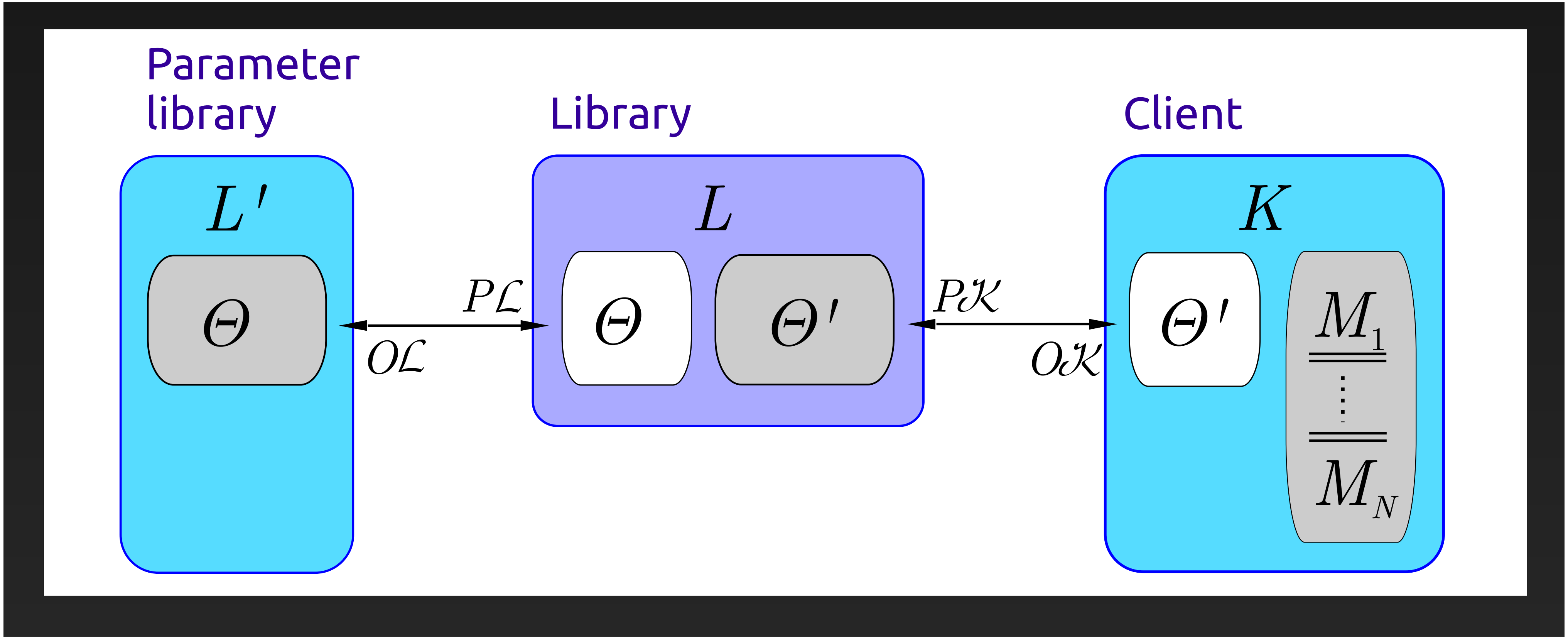
```
public count, update;
Lock lock;
F := λx.0;

count = λi. (!F)i
update = λ(i, g). aux(i, g, count i)

aux = λ(i, g, j).
   let y = |g j| in
     lock.acquire();
     let f = !F in
       if (j == (f i)) then {
         F := λx. if (x == i) then y
                      else (f x) ;
         lock.release();
         y }
       else {
         lock.release();
         aux(i, g, f i) }
```

http://c-cube.github.io/ocaml-containers/0.21/CCMultiSet.S.html

ENCAPSULATED CASE



ENCAPSULATED CASE



LINEARISABILITY (ENCAPSULATED)

$t \neq t'$

$$\cdots\,(t,x)_{\mathcal{K}}\,(t',x')_{\mathcal{L}}\cdots \quad \diamond \quad \cdots\,(t',x')_{\mathcal{L}}\,(t,x)_{\mathcal{K}}\cdots$$

$$\cdots\,(t',x')_{\mathcal{L}}\,(t,x)_{\mathcal{K}}\cdots \quad \diamond \quad \cdots\,(t,x)_{\mathcal{K}}\,(t',x')_{\mathcal{L}}\cdots$$

Def. History h_1 *linearises* to h_2 if h_2 can be obtained from h_1 by a series of $\triangleleft$- and $\diamond$-transformations.

RELATIONAL LINEARISABILITY

- **linearisability under additional assumptions about clients**
- **the extra constraints are expressed as closure under a relation R**
- **correctness with respect to clients whose behaviour (trace set) is R-closed**

COMBINING

$$run = \lambda(f,x).\ \ (lock.acquire();\ \ \text{let}\ \ result = f(x)\ \ \text{in}\ \ lock.release();\ \ result)$$

FLAT COMBINING

```
public run; . . . ;
Lock lock;
struct {fun, arg, wait, retv} requests[N];

run = λ (f,x).
    requests[t_id].fun := f;
    requests[t_id].arg := x;
    requests[t_id].wait := 1;
   while (requests[t_id].wait)
       if (lock.tryacquire()) {
          for (t=0; t<N; t++)
             if (requests[t].wait) {
                   requests[t].retv :=
                            requests[t].fun (requests[t].arg);
                   requests[t].wait := 0;
             }; lock.release() };
    requests[t_id].retv;
```

SUMMARY

- **new framework for higher-order linearisability in various cases (general, encapsulated, relational)**
- **soundness and compositionality**
- **case studies**
- **main target for future work:**

proof techniques for higher-order linearisability