Checking Correctness of Concurrents Objects: Tractable Reductions to Reachability

Ahmed Bouajjani
LIAFA, Univ Paris Diderot - Paris 7

Joint work with

Michael Emmi — IMDEA
Constantin Enea — LIAFA, U Paris Diderot - P7
Jad Hamza — LIAFA, U Paris Diderot - P7

FSTTCS, Bangalore, December 16, 2015

Concurrent Systems

- Concurrency at all levels of computer systems

  Hardware (Multicores), OS (device drivers, …), Applications

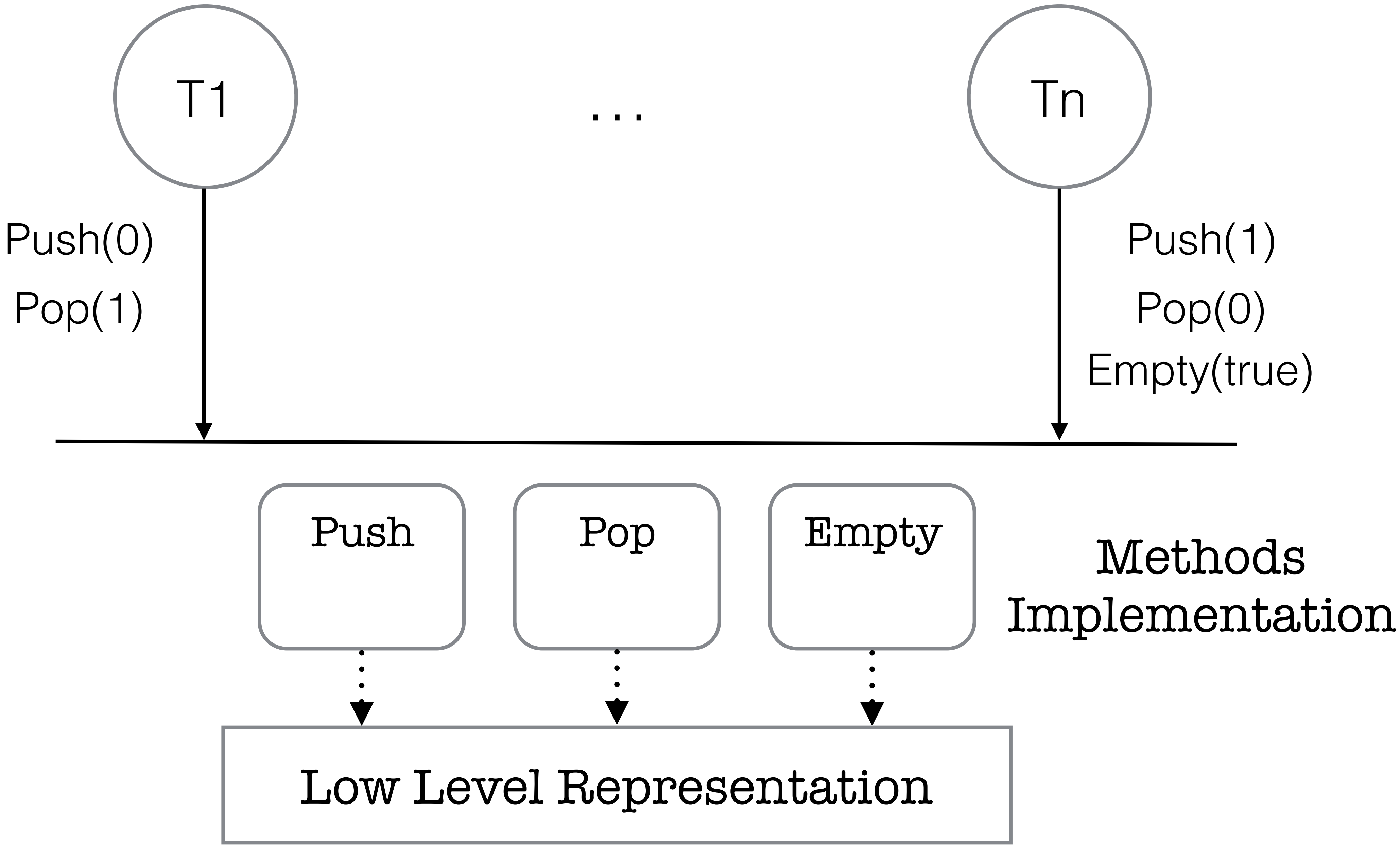
- Concurrent systems are complex

  Huge number of interleavings/action orders, intricate behaviours

- Need of abstractions

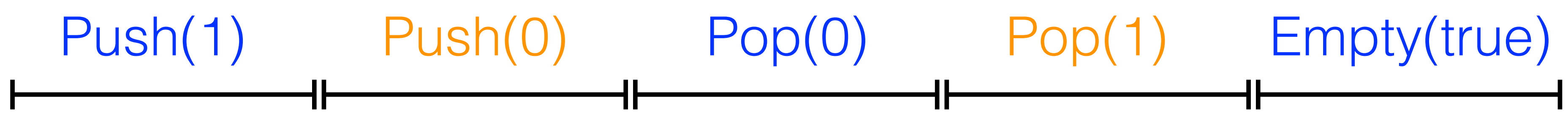
  Atomicity, synchrony, …

Concurrent Data Structures



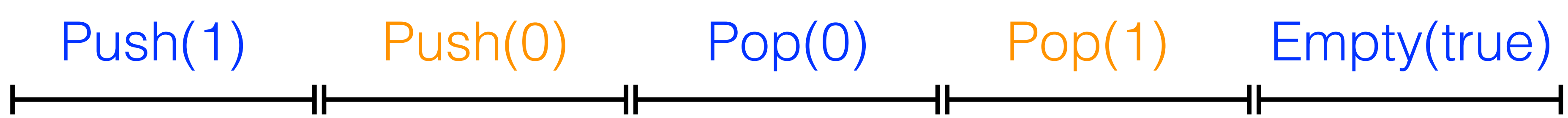
Abstract (Client) View

- Operations are considered to be atomic
- Thread executions are interleaved
- Executions satisfy sequential specifications



Abstract (Client) View

- Operations are considered to be atomic
- Thread executions are interleaved
- Executions satisfy sequential specifications

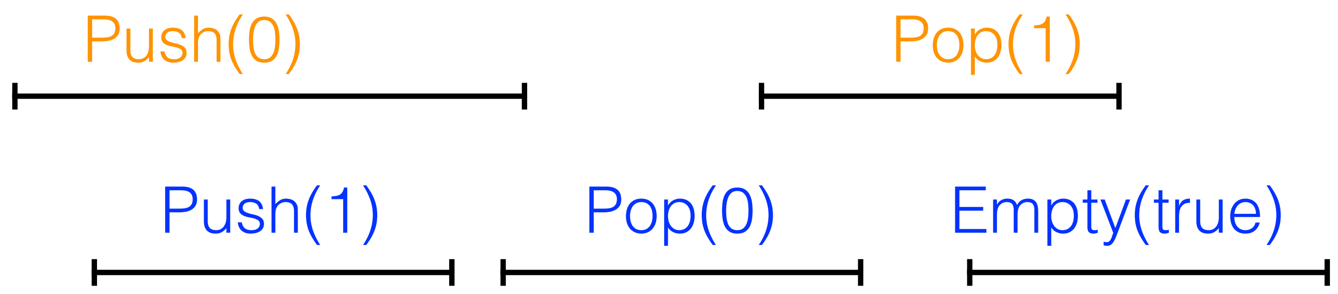


A "simple" implementation:

- Take a sequential implementation
- Lock at the beginning, unlock at the end of each method
- \+ **Reference Implementation**: simple to understand
- \- Low performances in case of contention

Efficient Concurrent Implementations

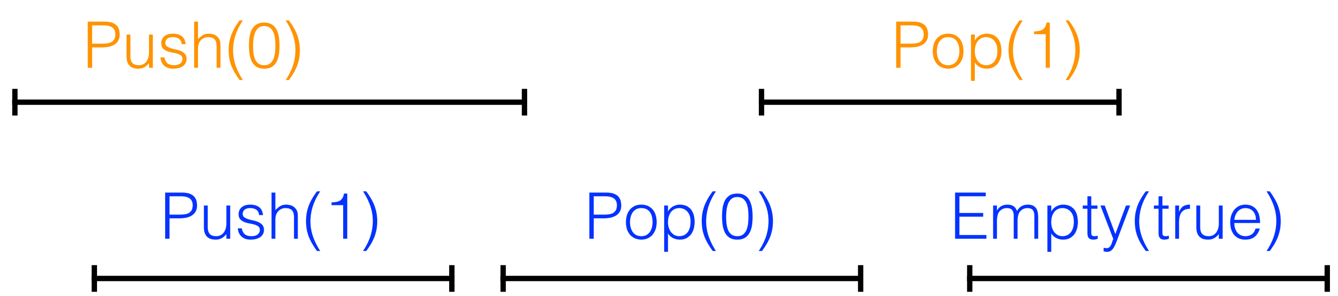
- Avoid the use of locks
- Maximise parallelisation of operations



- Check for interferences, and retry
- Use lower level synchronisation primitives (CAS)

Efficient Concurrent Implementations

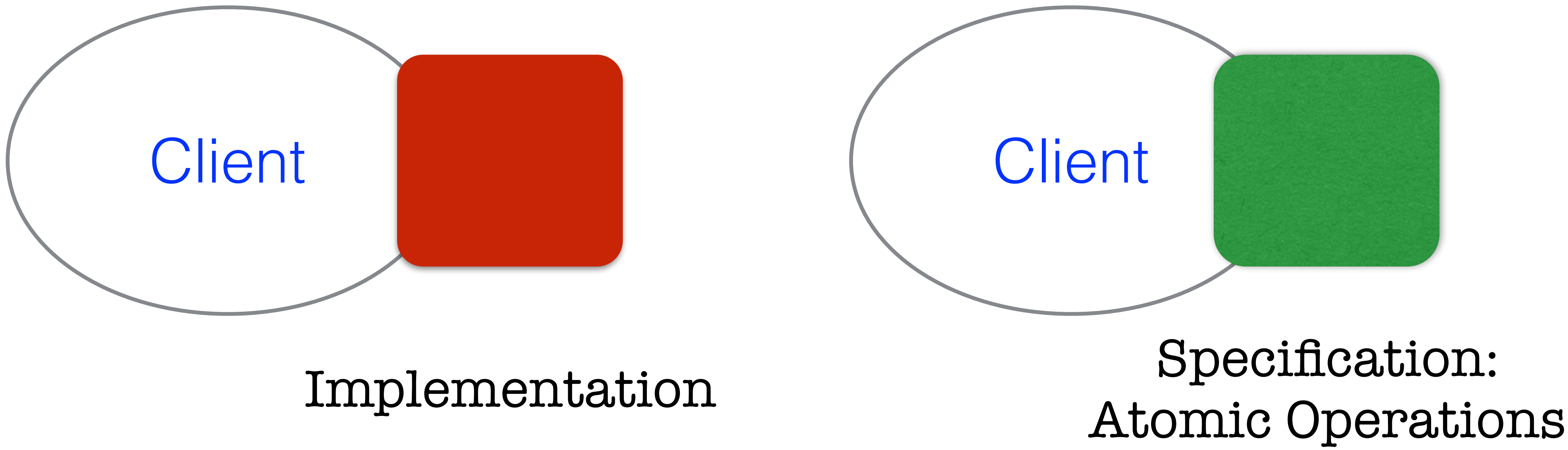
- Avoid the use of locks
- Maximise parallelisation of operations



- Check for interferences, and retry
- Use lower level synchronisation primitives (CAS)

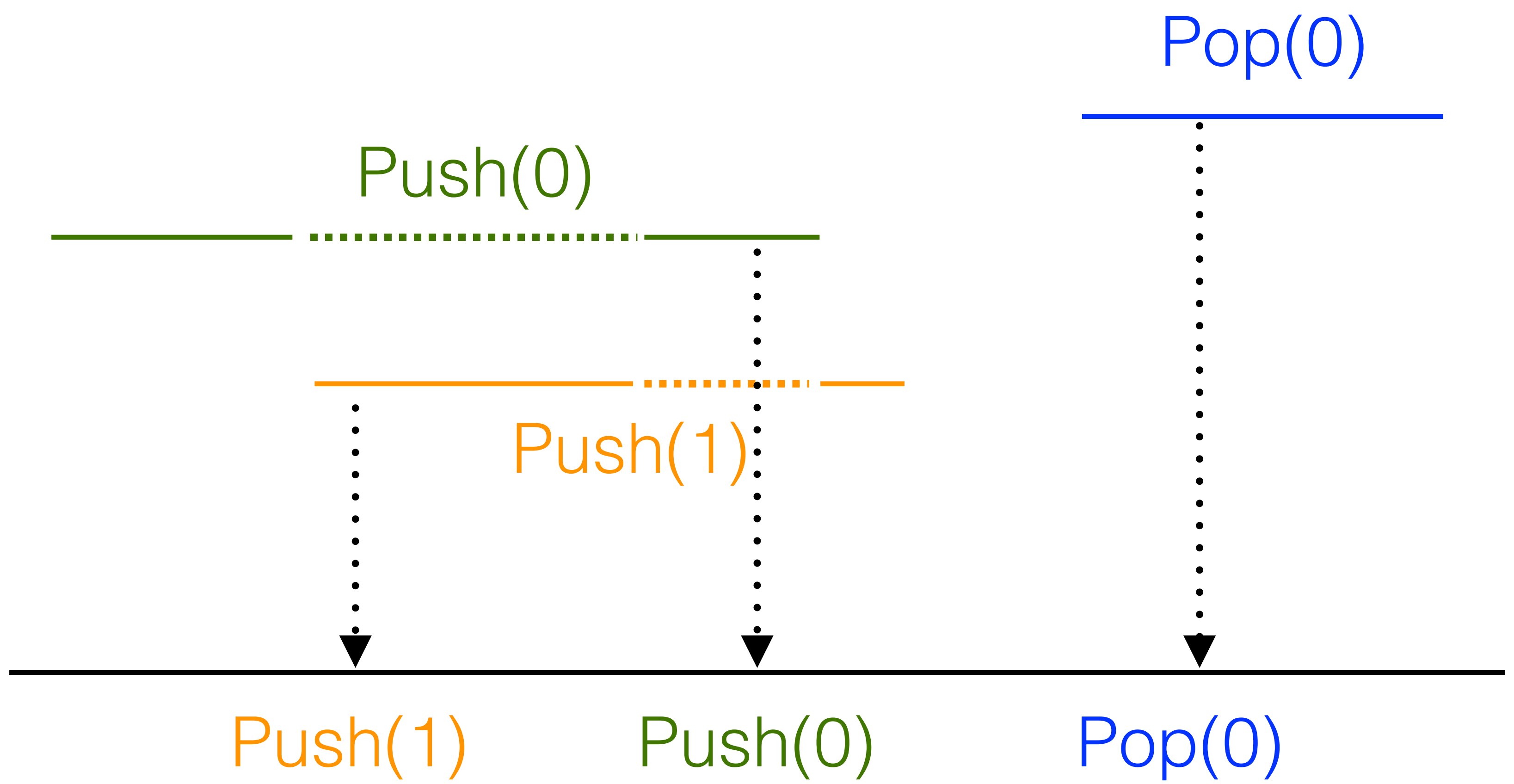
- ==> Complex behaviours!
- ==> Need to ensure the atomic view to the user!

Observational Refinement



For every Client,
Client x Impl is included in Client x Spec

Linearizability [Herlihy, Wing, 1990]



Valid sequence in the sequential specification

- **Reorder** call/return events, while **preserving returns —> calls**
- Find "**linearization points**" within execution time intervals
- s.t. **match some sequential execution**

Linearizability <=> Observational Refinement

[Filipovic, O'Hearn, Rinetzky, Yang, 2009], [B., Enea, Emmi, Hamza, 2015]

Checking Linearizability: Complexity

Existing results

- NP-complete for a single computation [Gibbons, Korach, 1997]
- In EXSPACE for a fixed number of threads, finite-state methods and specifications [Alur et al., 1996]

Recent contributions

- EXPSPACE-hard for FS impl.’s and spec’s [Hamza 2015]
- Undecidable for unbounded number of threads, FS methods and spec.’s [B., Enea, Emmi, Hamza, 2013]

Checking Linearizability: Main Existing Approaches

- **Enumerate** executions and **linearisation orders** (bug detect.)

  e.g. *Line-up* [Burckhardt et al. PLDI'10]

- **Fixed linearisation points** in the code (correctness)

  Checking linearizability —> Reachability problem/Invariant checking

  e.g., [Vafeiadis, CAV'10],
  [Abdulla et al., TACAS 2013]

Checking Linearizability: Main Existing Approaches

- **Enumerate** executions and **linearisation orders** (bug detect.)

e.g. *Line-up* [Burckhardt et al. PLDI'10]

- **Fixed linearisation points** in the code (correctness)

Checking linearizability —> Reachability problem/Invariant checking

e.g., [Vafeiadis, CAV'10],
[Abdulla et al., TACAS 2013]

- **Scalability** issues
- **Fixing** linearisation points is **not** always **possible**

e.g., time-stamping based stack [Dodds, Haas, Kirsch, POPL'15]

Reductions Linearizability to State Reachability?

Why?

- **Reuse existing tools** for State reachability
- **Lower complexity**, decidability

Reductions Linearizability to State Reachability?

Why?

- **Reuse existing tools** for State reachability
- **Lower complexity**, decidability

General Approach:

Given a **library L** and a **specification S**,
define a monitor **M** (**+** designated **bad states**) s.t.
L is linearisable wrt S iff
L x M does not reach a bad state

Reductions Linearizability to State Reachability?

Why?

- **Reuse existing tools** for State reachability
- **Lower complexity**, decidability

General Approach:

Given a **library L** and a **specification S**,
define a monitor **M** (**+** designated **bad states**) s.t.
L is linearisable wrt S iff
L x M does not reach a bad state

Issue:

- The computational power of M?
- **Ideally**, M should be a **finite state machine**
- M should be **"simple" (low overhead)**

Option 1: Under-approximate Analysis

[B, Emmi, Enea, Hamza, POPL'15]

- **Bounded information** about computations
- Useful for **efficient bug detection**

Option 1: Under-approximate Analysis

[B, Emmi, Enea, Hamza, POPL'15]

- **Bounded information** about computations
- Useful for **efficient bug detection**

- **Bounding concept** for detecting linearizability violations?
- Should offer **good coverage**, and **scalability**

Option 1: Under-approximate Analysis

[B, Emmi, Enea, Hamza, POPL'15]

- **Bounded information** about computations
- Useful for **efficient bug detection**

- **Bounding concept** for detecting linearizability violations?
- Should offer **good coverage**, and **scalability**

- **Interval-length** bounded analysis
- Based on characterising *linearizability as history inclusion*
- Monitor uses **counters**
- Allows for symbolic encodings
- Efficient static and dynamic analysis

Option 2: Particular classes of Objects

[B, Emmi, Enea, Hamza, ICALP'15]

What is the situation for **usual objects**?

stacks, queues, etc.

- Violations: **Finite number of bad patterns**
- They can be **captured with** small **finite-state automata**
- **Linear reduction** to state reachability
- *Decidability* for *unbounded* number of threads

Histories

History of an execution *e* :

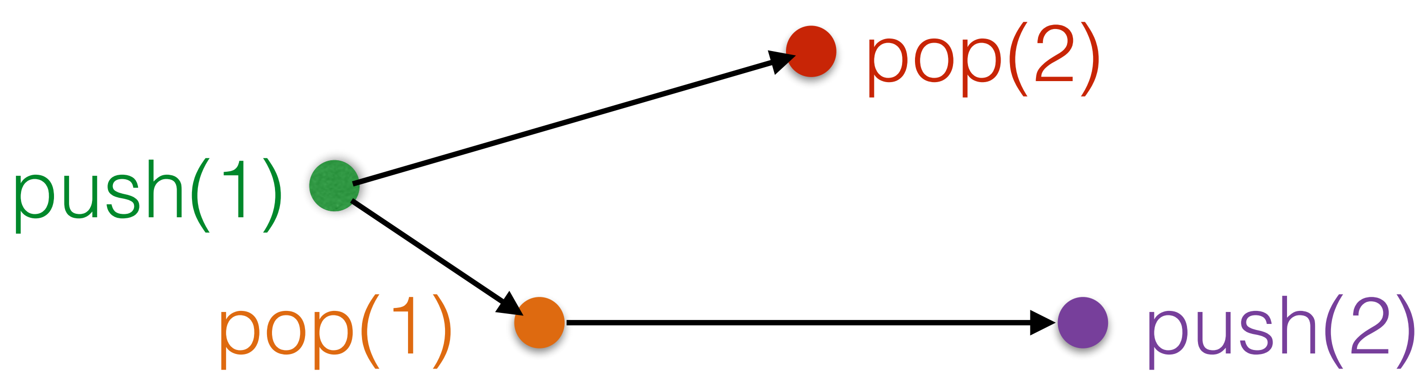
H(*e*) = (O, label, <)

where

- O = Operations(e)
- label: O —> M x V x V
- < is a partial order s.t.

O1 < O2 iff Return(O1) is *before* Call(O2) in *e*

c(push,1) r(push,tt) c(pop,-) c(pop,-) r(pop,1) c(push,2) r(push,tt) r(pop,2)



Linearizability as a History Inclusion

Consider an **abstract data structure**,
let **S** be its **sequential specification**,
and let $\mathbf{L_S}$ be a **sequential implementation** of S,
i.e., *L_S satisfies S*

$\mathbf{L_C}$ reference concurrent implementation =
$\mathbf{L_S}$ + lock/unlock at beginning/end of each method

Linearizability as a History Inclusion

Consider an **abstract data structure**,
let **S** be its **sequential specification**,
and let **L_S** be a **sequential implementation** of S,
i.e., ***L_S satisfies S***

L_C reference concurrent implementation =
L_S + lock/unlock at beginning/end of each method

Lemma:
$H(L_C)$ is the set histories that are linearised to a sequence in S

Thm: **L is linearisable wrt S iff H(L) is included in $H(L_C)$**

Abstracting Histories

Weakening relation

$h_1 \leq h_2$ (h_1 is weaker than h_2)

iff

h_1 has less constraints than h_2

Lemma:

($h_1 \leq h_2$ and h_2 is in H(L)) ==> h_1 is in H(L)

Approximation Schema

Weakening function A_k, for any given $k \geq 0$, s.t.

- $A_k(h) \leq h$
- $A_0(h) \leq A_1(h) \leq A_2(h) \leq \ldots \leq h$
- There is a k s.t. $h = A_k(h)$

Approximation Schema

Weakening function A_k, for any given $k \geq 0$, s.t.

- $A_k(h) \leq h$
- $A_0(h) \leq A_1(h) \leq A_2(h) \leq \ldots \leq h$
- There is a k s.t. $h = A_k(h)$

Approximate History Inclusion Checking, for fixed $k \geq 0$

- Given a library L and a specification S
- Check: **Is there an h in H(L) s.t. A_k(h) is not in H(S)?**
- A_k(h) is not in H(S) => h is not in H(S) — Violation!

Histories are Interval Orders

Interval Orders = partial order (O, <) such that

(o1 < o1' and o2 < o2') implies (o1 < o2' or o2 < o1')

Prop: For every execution *e*, H(*e*) is an interval order

Notion of Length

Let h = (O,<) be an Interval Order (history in our case)

- Past of an operation: past(o) = {o' : o' < o}
- Lemma [Rabinovitch'78]:

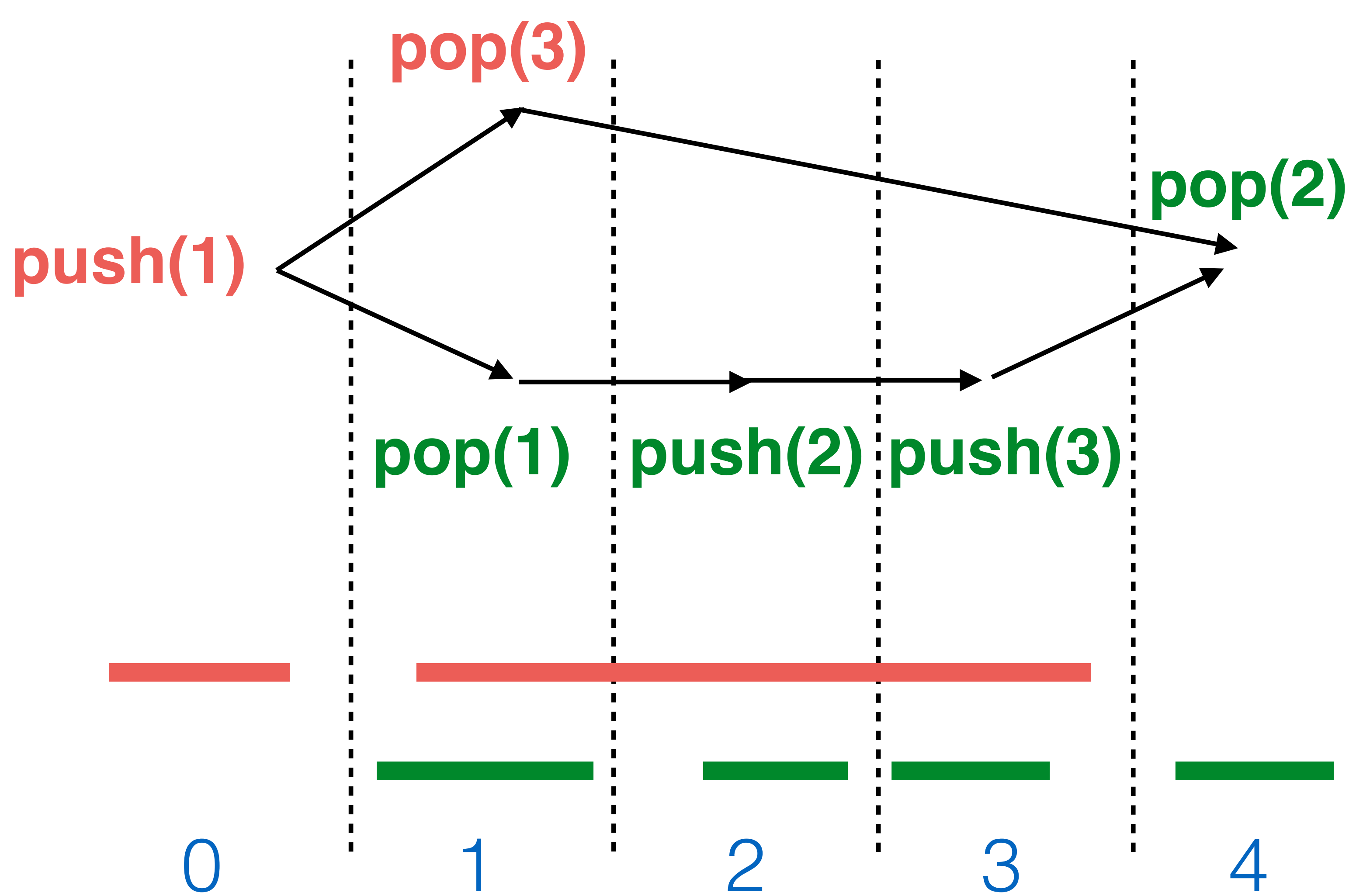
  The set {past(o) : o in O} is linearly ordered

- The *length* of the order = number of pasts - 1

Canonical Representation of Interval Orders

- Mapping I : O —> $[n]^2$ where n = length(h) [Greenough '76]
- I(o) = [i, j], with i, j ≤ n, such that

i = |{past(o') : o' < o}| and
j = |{past(o') : not (o < o')}| - 1



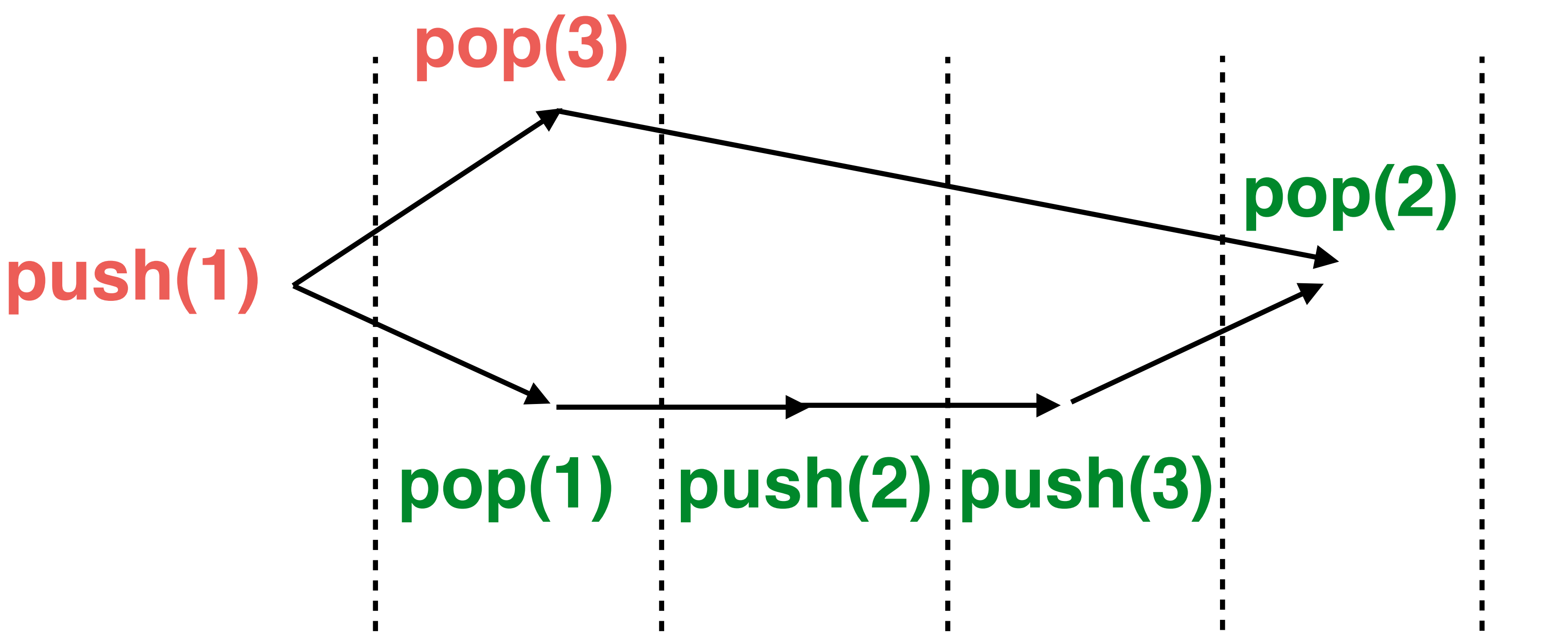
I(push(1)) = [0, 0]
I(pop(1)) = [1, 1]
I(push(2)) = [2, 2]
I(push(3) = [3, 3]
I(pop(3)) = [1, 3]
I(pop(2)) = [4, 4]

length = 4

Bounded Interval-length Approximation

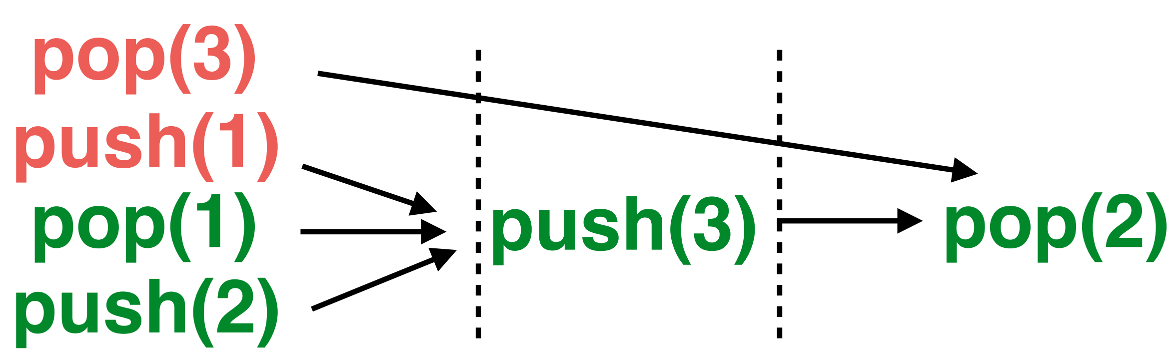
Let A_k maps each h to some h' ≤ h of length k

=> Keep precise the information about the k last intervals



I(push(1)) = [0, 0]
I(pop(1)) = [0, 0]
I(push(2)) = [0, 0]
I(push(3) = [1, 1]
I(pop(3)) = [0, 1]
I(pop(2)) = [2, 2]

K=2



Counting Representation of Interval Orders

Count the number of occurrences
of each operation type in each interval

- h = (O, <) an IO with canonical representation I:O—>$[k]^2$
- Associate a **counter** with each operation type and interval
- **∏(h) is the Parikh image of h**
- It represents the multi-set **{** [label(o), I(o)] : o in O **}**

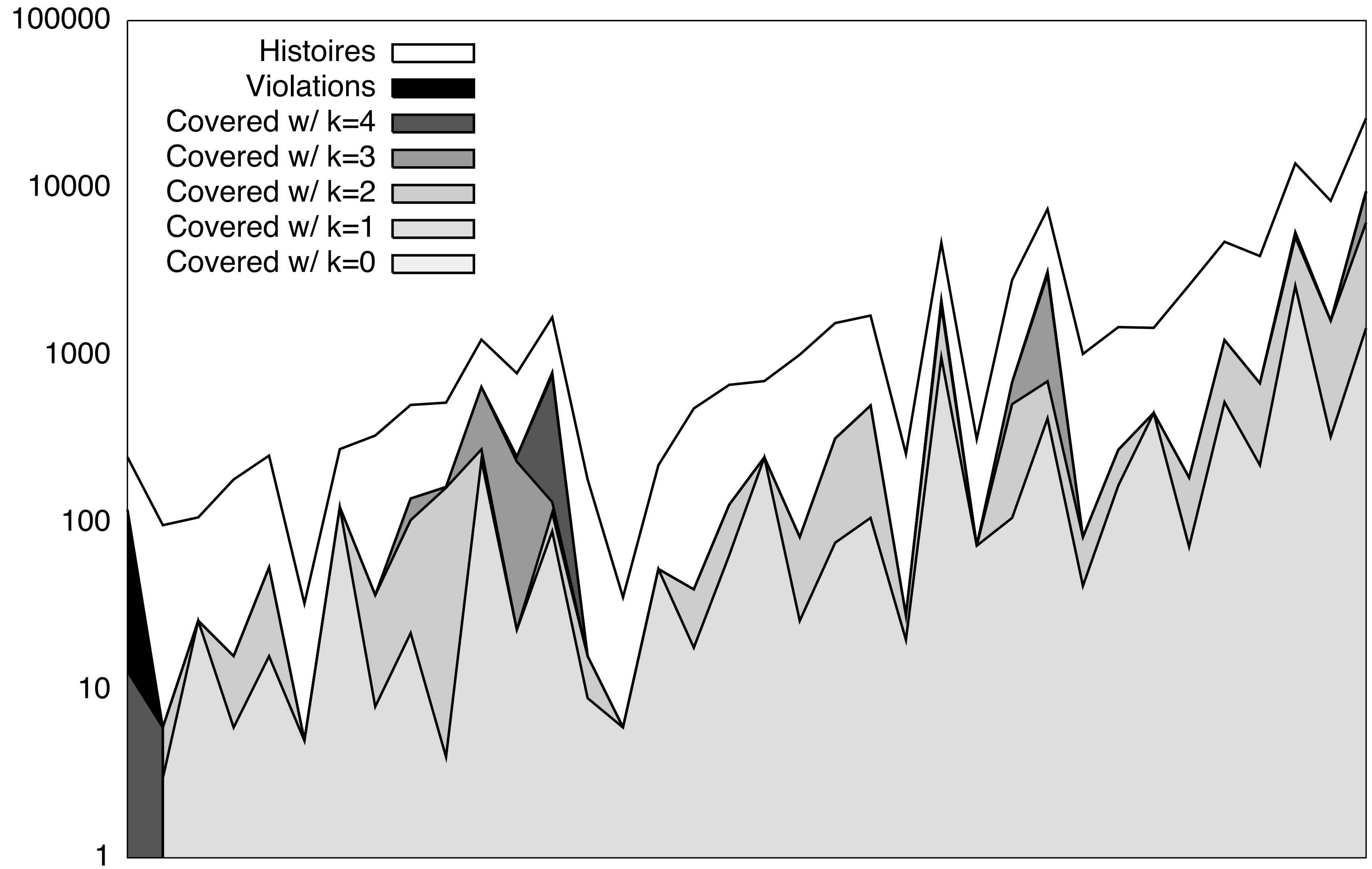
Prop: $H_k(e)$ is in $H_k(L)$ iff ∏($H_k(e)$) is in ∏($H_k(L)$)

Reduction to Reachability with Counters

$H_k(L)$ subset of $H_k(S)$
iff
$\Pi(H_k(L))$ subset of $\Pi(H_k(S))$

- Consider **k-bounded-length abstract histories**
- Track histories of L using a **finite number of counters**
- Use an **arithmetic-based representation of $\Pi(H_k(S))$**
- Π(Hk(S)) can be either computed, or given manually
- Check that **$\Pi(H_k(S))$ is an invariant**

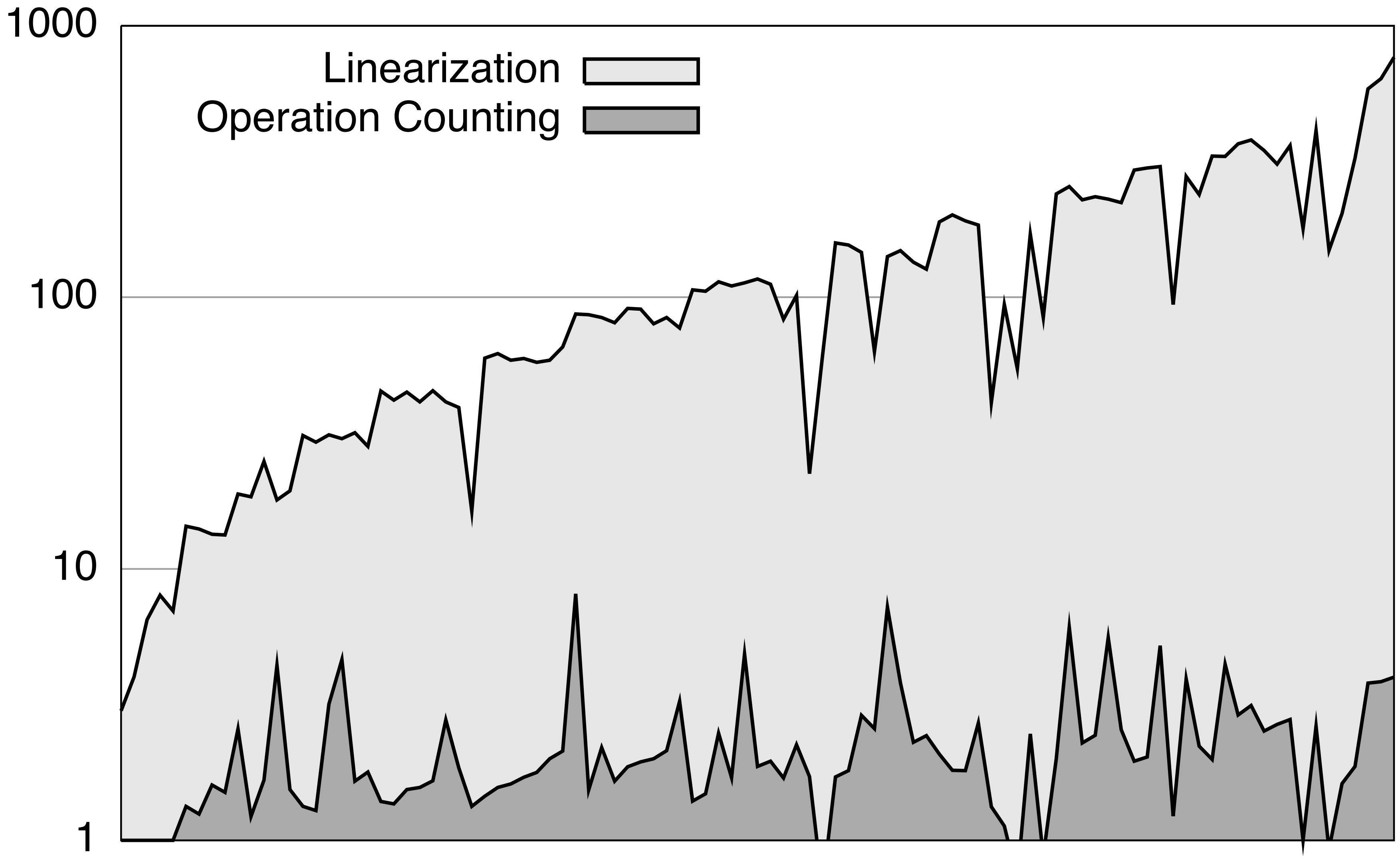
Experimental Results: Coverage



Comparison of violations covered with k ≤ 4

- Data point: Counts in logarithmic scale over all executions (up to 5 preemptions) on Scal's nonblocking bounded-reordering queue with ≤4 enqueue and ≤4 dequeue
- x-axis: increasing number of executions (1023-2359292)
- White: total number of unique histories over a given set of executions
- Black: violations detected by traditional linearizability checker (e.g., Line-up)

Experimental Results: Runtime Monitoring



Comparison of runtime overhead
between Linearization-based monitoring and Operation counting

- Data point: runtime on logarithmic scale, normalised on unmonitored execution time
- Scal's nonblocking Michael-Scott queue, 10 enqueue and 10 dequeue operations.
- x-axis is ordered by increasing number of operations

Experimental Results: Static Analysis

Library	Bug	P	k	m	n	Time
Michael-Scott Queue	B1 (head)	2x2	1	2	2	24.76s
Michael-Scott Queue	B1 (tail)	3x1	1	2	3	45.44s
Treiber Stack	B2	3x4	1	1	2	52.59s
Treiber Stack	B3 (push)	2x2	1	1	2	24.46s
Treiber Stack	B3 (pop)	2x2	1	1	2	15.16s
Elimination Stack	B4	4x1	0	1	4	317.79s
Elimination Stack	B5	3x1	1	1	4	222.04s
Elimination Stack	B2	3x4	0	1	2	434.84s
Lock-coupling Set	B6	1x2	0	2	2	11.27s
LFDS Queue	B7	2x2	1	1	2	77.00s

- Static detection of injected refinement violations with CSeq & CBMC.
- Program Pij with i and j invocations to the push and pop methods, explore n-round round-robin schedules with m loop iterations unrolled, with monitor for Ak.
- Bugs: (B1) non-atomic lock, (B2) ABA bug, (B3) non-atomic CAS operation, (B4) misplaced brace, (B5) forgotten assignment, (B6) misplaced

Focusing on Special Classes of Objects
[B., Emmi, Enea, Hamza, ICALP 2015]

- Inductive definition of sequential objects (restricted language based on *constrained rewrite rules*)
- Characterizing concurrent violations using a **finite number of "bad patterns"**, *one per rule*
- Defining **finite-state automata** recognising each of the "bad patterns" (using *data independence* assumption)
- Reducing *linearizability* to checking the *emptiness of the intersection with these automata*.

Specifying queues and stacks

Queue

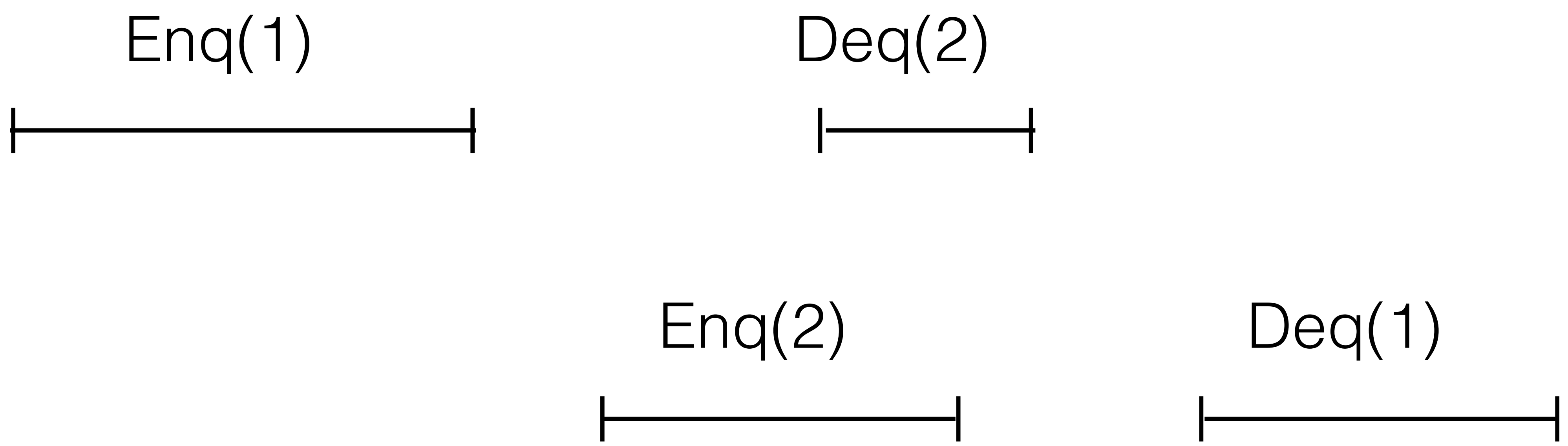
- u . v : Q & u : ENQ* —> **Enq(x)** . u . **Deq(x)** . v : Q
- u . v : Q & no unmatched *Enq* in u —> u . **Emp** . v : Q

Stack

- u . v : S & no unmatched *Push* in u —> **Push(x)** . u . **Pop(x)** . v : S
- u . v : S & no unmatched *Push* in u —> u . **Emp** . v : S

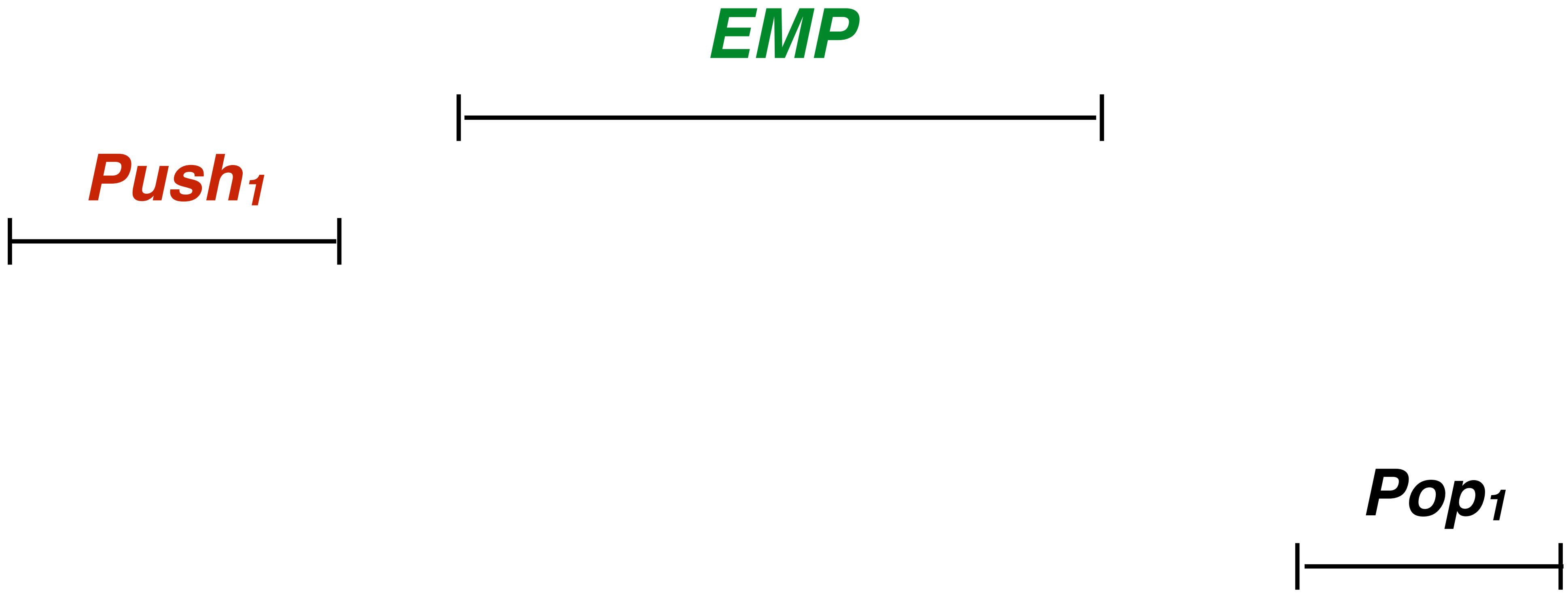
Order Violation

FIFO violation:

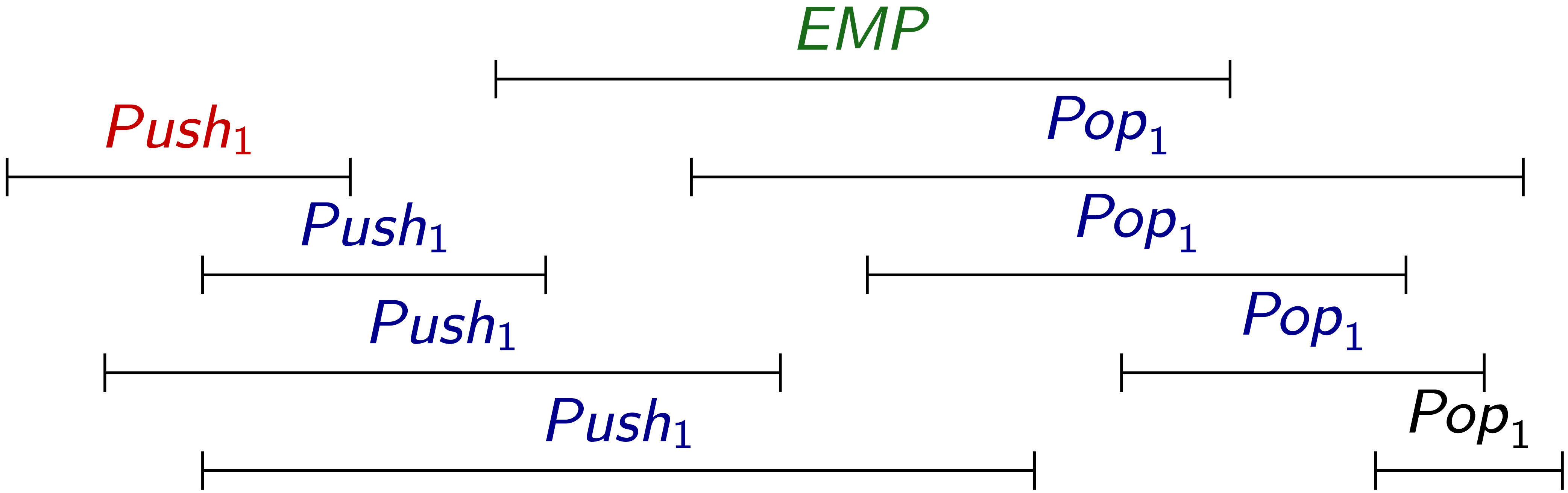


ret(Enq(1)) < call(Enq(2)) & ret(Deq(2)) < call(Deq(1))

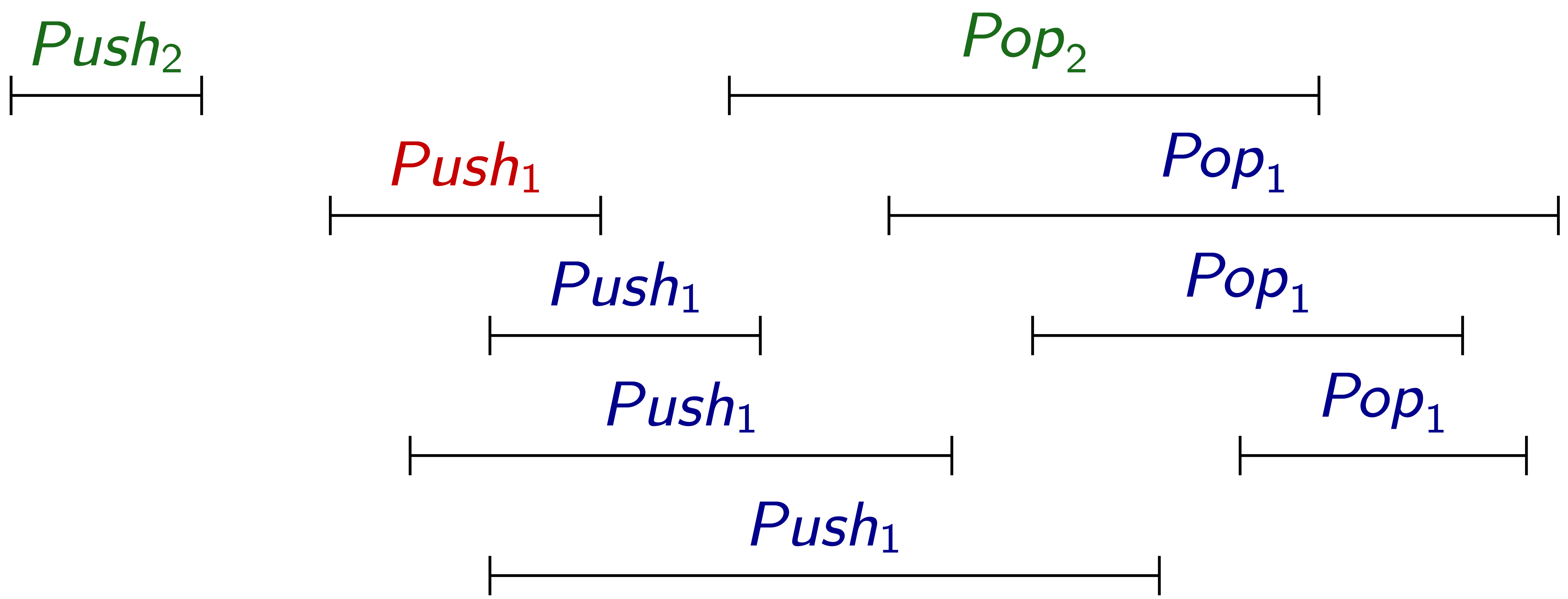
Empty Violation



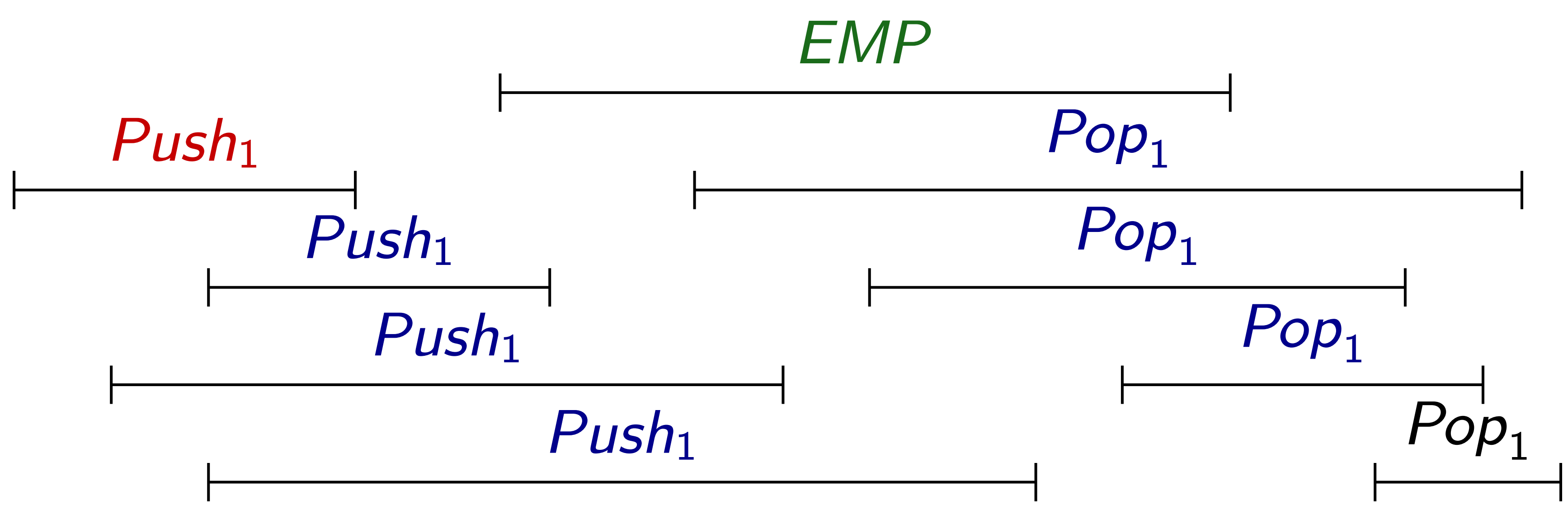
Empty Violation



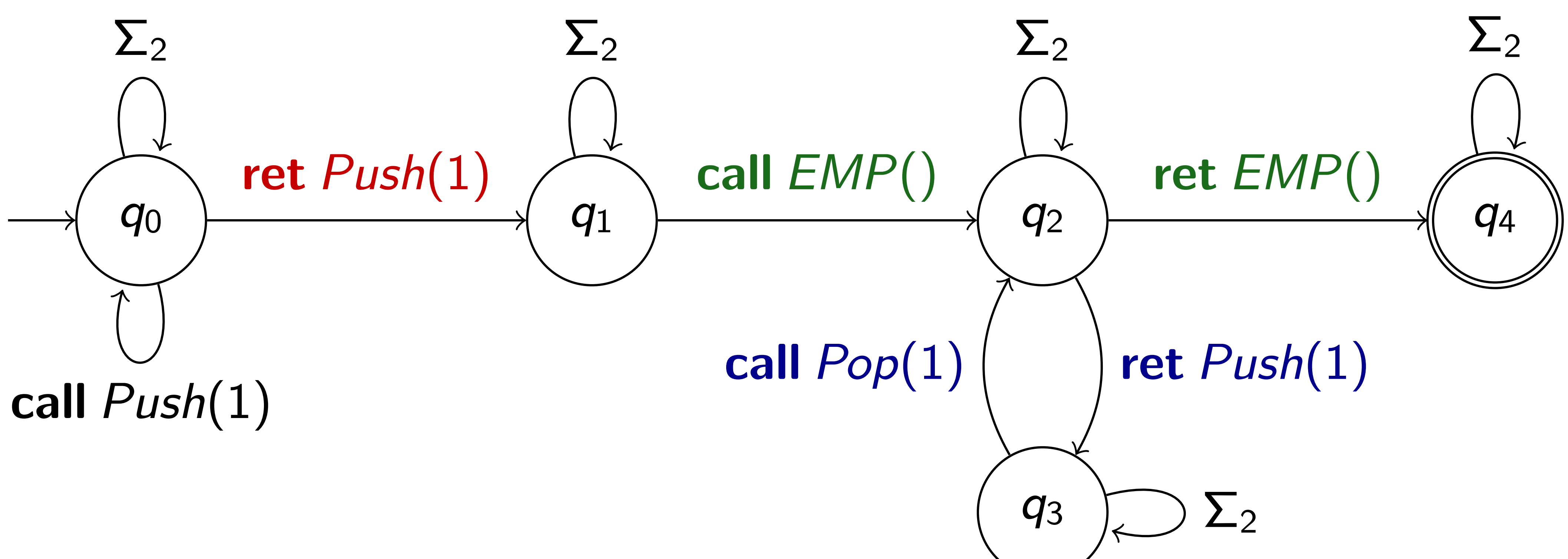
Order Violation cont.



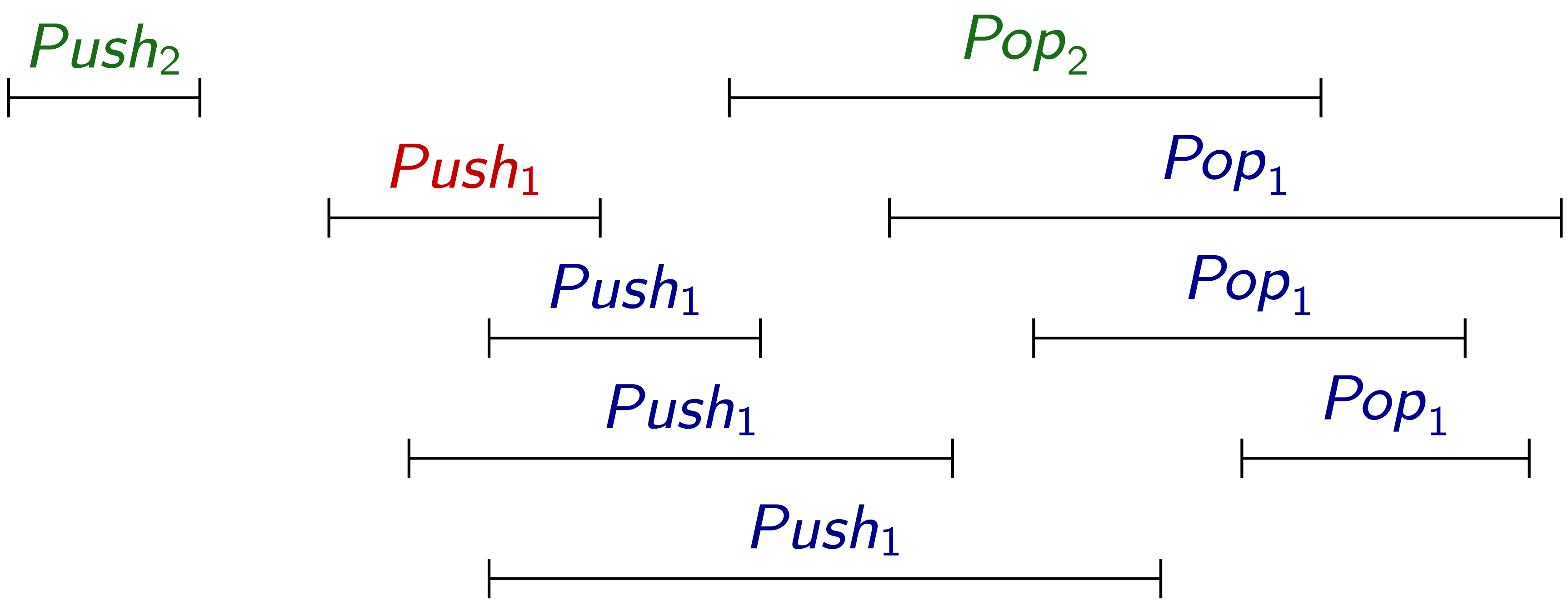
Automaton for Empty Violation



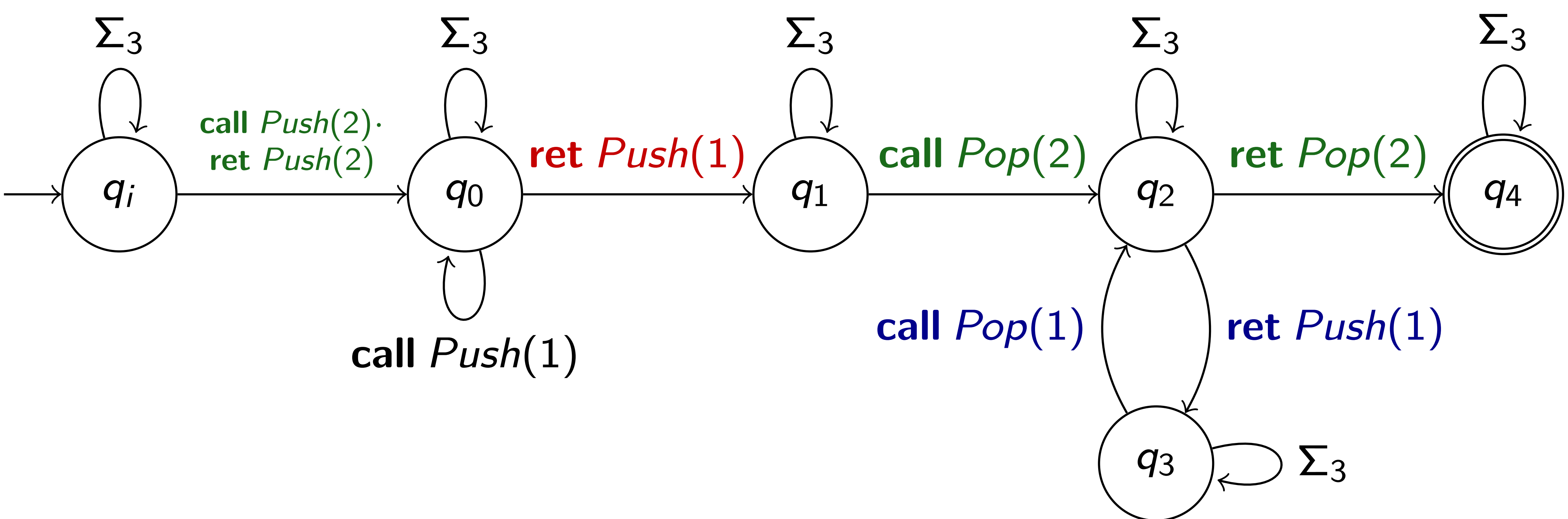
Recognized by:



Automaton for Push-Pop Order Violation



Recognized by:



Linearizability to State Reachability

Thm:

For each **S** in {Stack, Queue, Mutex, Register},

there is an automaton **A(S)** s.t.

for every data independent concurrent implementation **L**,

L is linearisable wrt S iff L intersected with A(S) is empty

Same complexity as state reachability

Conclusion

- **Linearizability** checking is **hard/undecidable** in general
- But **tractable reductions to state reachability** are possible
- **Abstracting histories** using Interval-length Bounding:
  - Monitor uses counters: **simple encoding of order constraints**
  - Use symbolic techniques
  - Static and Dynamic Analysis
  - Good coverage, scalable monitoring
- Consider **relevant** classes of **concurrent objects**:
  - Covers common structures such as stacks and queues
  - Finite-state monitor: **Linear reduction to state reachability**
  - Decidability for unbounded number of threads

Future work

- Extend the 2nd approach to other structures, e.g., sets
- Combine with providing linearisation policies

  [Abdulla et al., TACAS'13]

- Distributed (replicated) data structures

  Weaker consistency notions are needed:
  Eventual consistency, causal consistency, etc.

  - Eventual consistency —> Model-checking, Decidability

    [B., Enea, Hamza, POPL'14]

  - Causal consistency undecidable [Hamza, 2015]

METIS/NETYS 2016

8th Intern. Spring School on Distributed Systems

16-18 May, Rabat, Morocco

This year's topic: **Big Data, Cloud**

http://metis2016.netys.net/home/

Organizers: **Rachid Guerraoui** (EPFL), **Mohammed Erradi** (ENSIAS, Rabat)

4th International Conference on Networked Systems

18-20 May, Rabat, Morocco

http://netys.net/

PC chairs: **Parosh Aziz Abdulla** (U. Uppsala), **Carole Delporte** (U. Paris 7)

+ Workshops