Refinement Checking for Libraries of Concurrents Objects

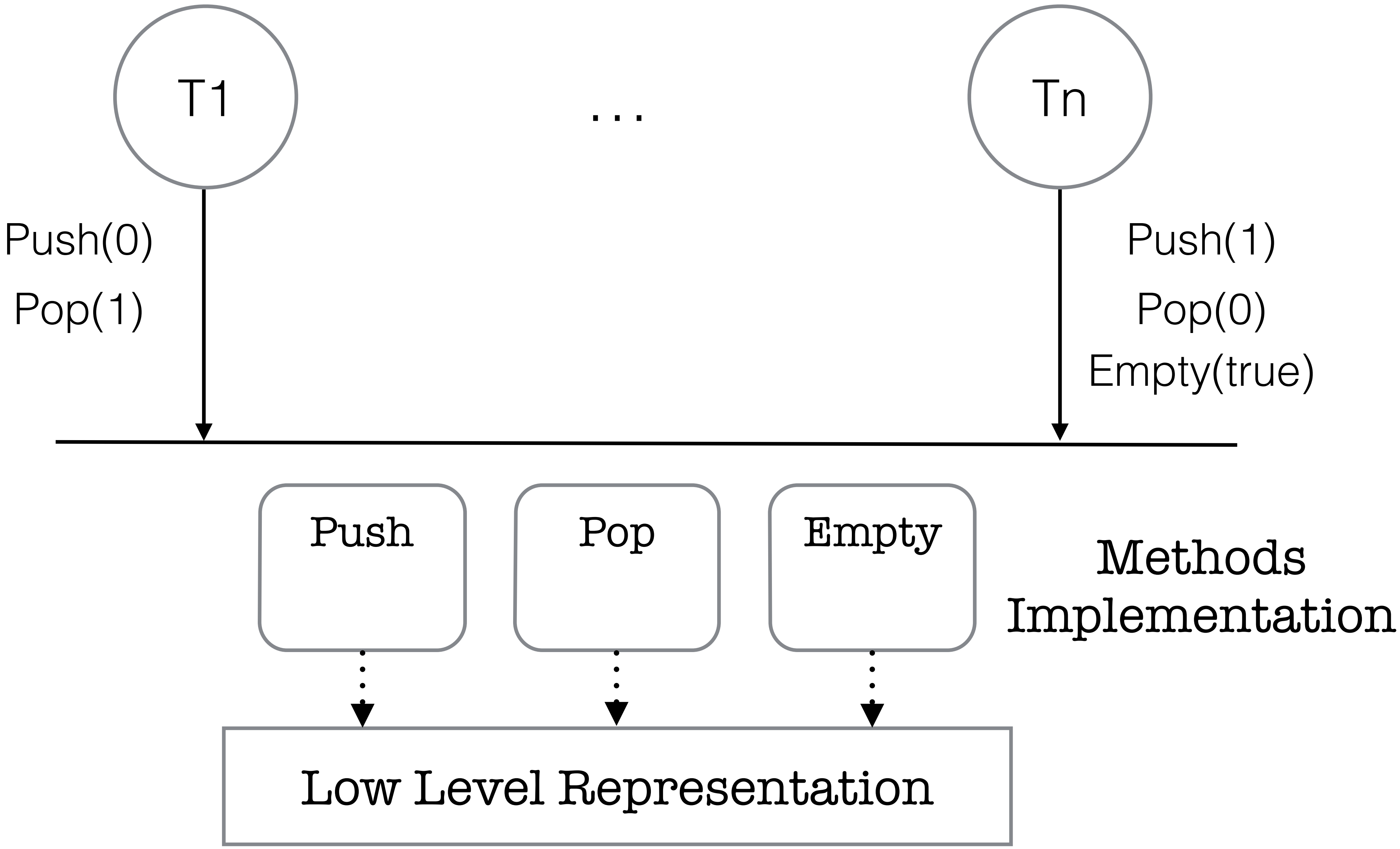
Ahmed Bouajjani
LIAFA, Univ Paris Diderot - Paris 7

Joint work with

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

PV, Madrid, September 5, 2015

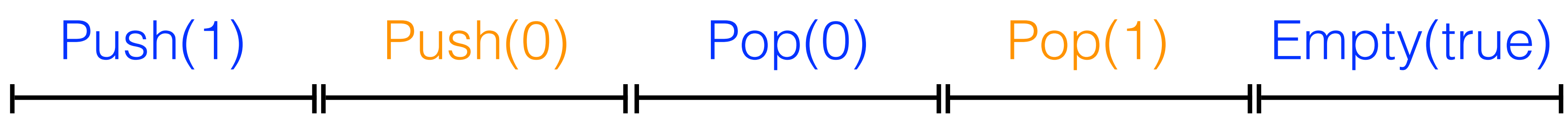
Concurrent Data Structures



Different atomicity levels

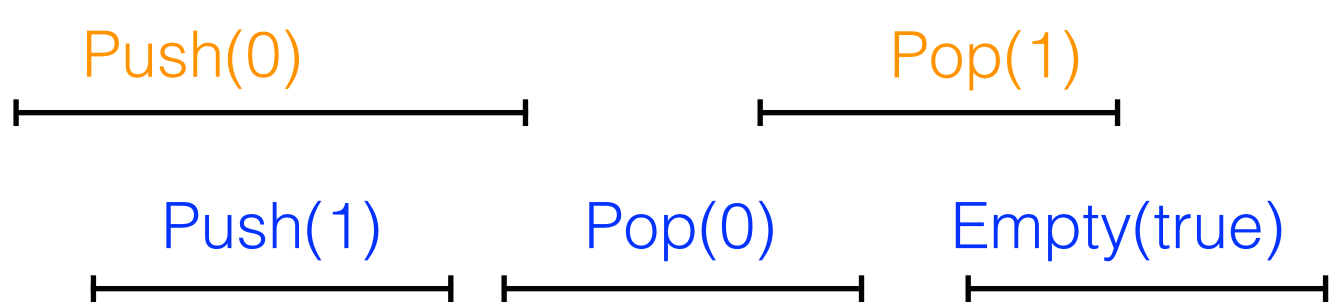
Client view:

- Operations are atomic
- Thread executions are interleaved

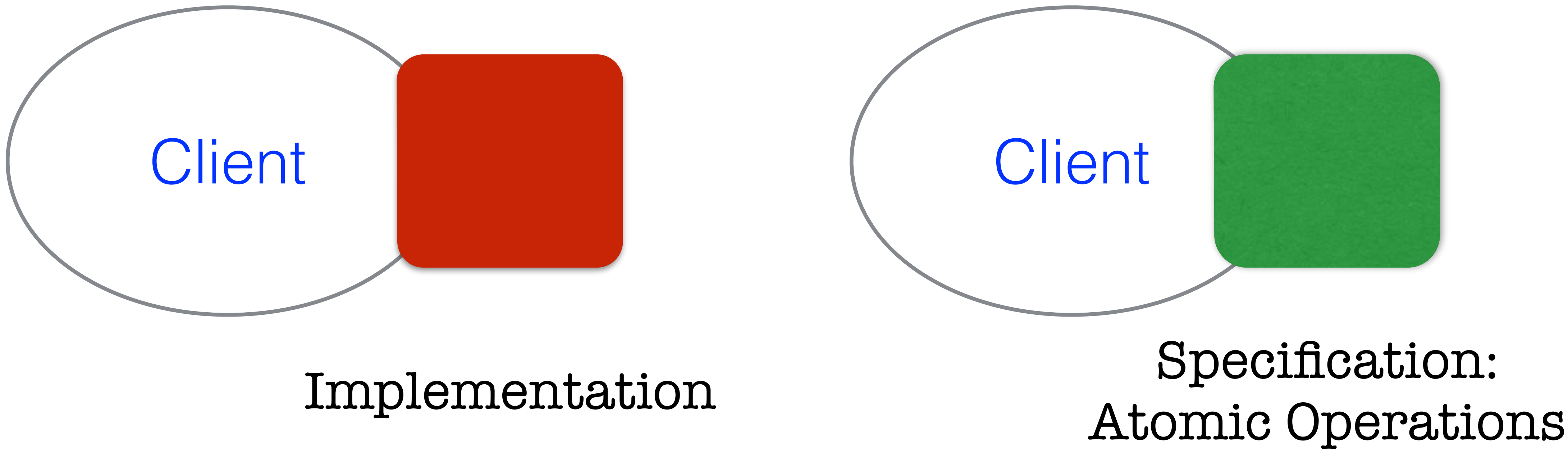


Implementation:

- Naive solution: Coarse-grain locking
- Performances: Avoid coarse-grain locking
- => Execution intervals may overlap

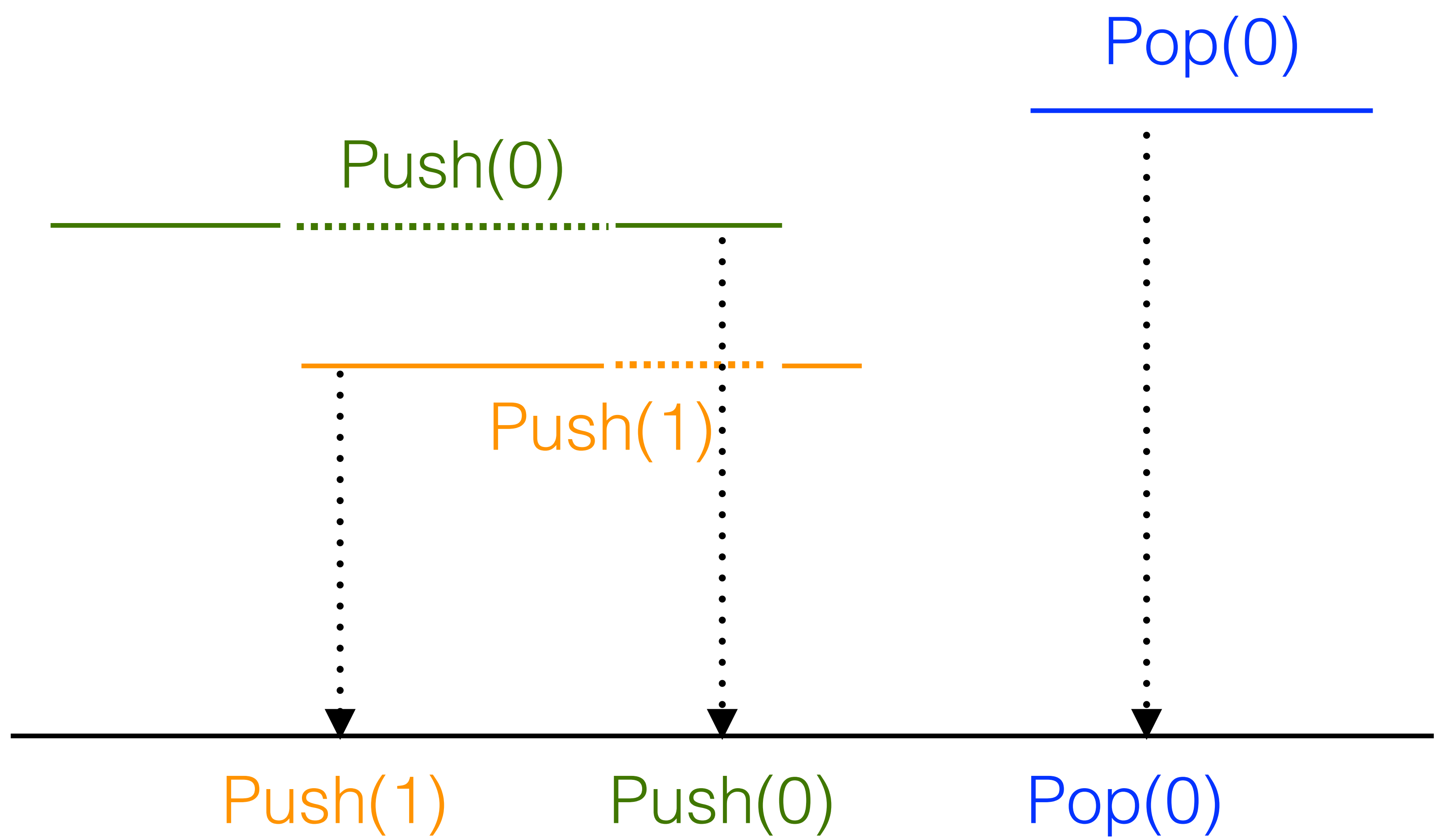


Observational Refinement



For every Client,
Client x Impl 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
- => match a sequential execution

Linearizability *implies* Observational Refinement
[Filipovic, O'Hearn, Rinetzky, Yang, 2009]

Complexity

- Checking linearizability of a single execution is NP-complete [Gibbons, Korach, 1997]
- For finite-state implementations and specifications:

	Linearizability	State Reachability
Fixed Nb Threads	EXPSPACE-complete (1)	PSPACE-complete
Unbounded Nb of Threads	Undecidable (2)	EXPSPACE-complete

(1) Upper Bound: Alur, McMillan, Peled 1996 — Lower Bound: Hamza 2015
(2) B., Emmi, Enea, Hamza, 2013

Reducing Linearizability/OR to State Reachability?

Why?

- Reuse existing tools for SR
- Lower complexity
- Decidability

Two approaches:

- Under approximations: special classes of executions
  - Parametrized under-approximation schema?
  - Good coverage, low complexity, scalability
- Special classes of objects:

  Stacks, queues, etc.

Under-approximate detection of OR violations ?

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

Characterizing OR as a History Inclusion Problem

- "Histories" are a special partial orders: Interval orders
- => Interval-length bounding

Efficient Reduction to Reachability

- Use counting representations
- => Correctness of a single history is Polynomial time
- => Scalable dynamic and static analysis techniques

Good coverage with small bounds

- ≤ 3
- Cut-off bounds for common data structures

Histories

History of an execution *e* :

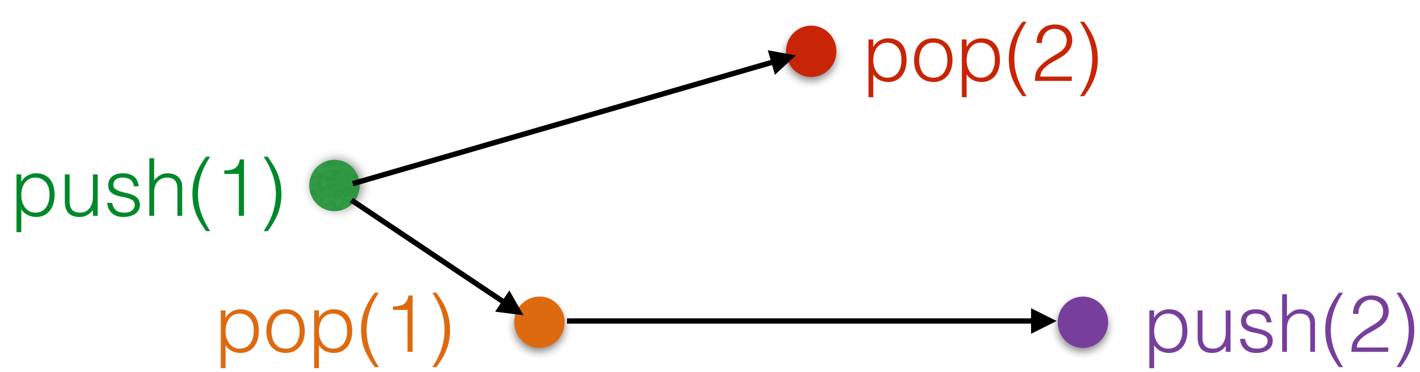
$$H(e) = (O, \text{label}, <)$$

where

- $O = \text{Operations}(e)$
- label: $O \longrightarrow M \times V \times V$
- $<$ 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)



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:

If $h_1 \leq h_2$ and h_2 is in H(L), then h_1 is in H(L) too

Approximation Schema for detecting OR violations

Parametrized weakening function A_k, for any $k \geq 0$, 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. $h \leq A_k(h)$
- Checking if $A_k(h)$ is in $H(L)$ decidable in polynomial time

Approximation Schema for detecting OR violations

Parametrized weakening function A_k, for any $k \geq 0$, 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. $h \leq A_k(h)$
- Checking if $A_k(h)$ is in $H(L)$ decidable in polynomial time

Approximating History Inclusion

- Choose a parameter $k \geq 0$
- Is there an h in $H(L_1)$ s.t. $A_k(h)$ is not in $H(L_2)$?
- Lemma => If $A_k(h)$ not in $H(L_2)$, then h is not in $H(L_2)$

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

A Bounding Concept for Histories

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

Notion of length:

- 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

A Bounding Concept for Histories

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

Notion of length:

- 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

Bounded interval-length approximation

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

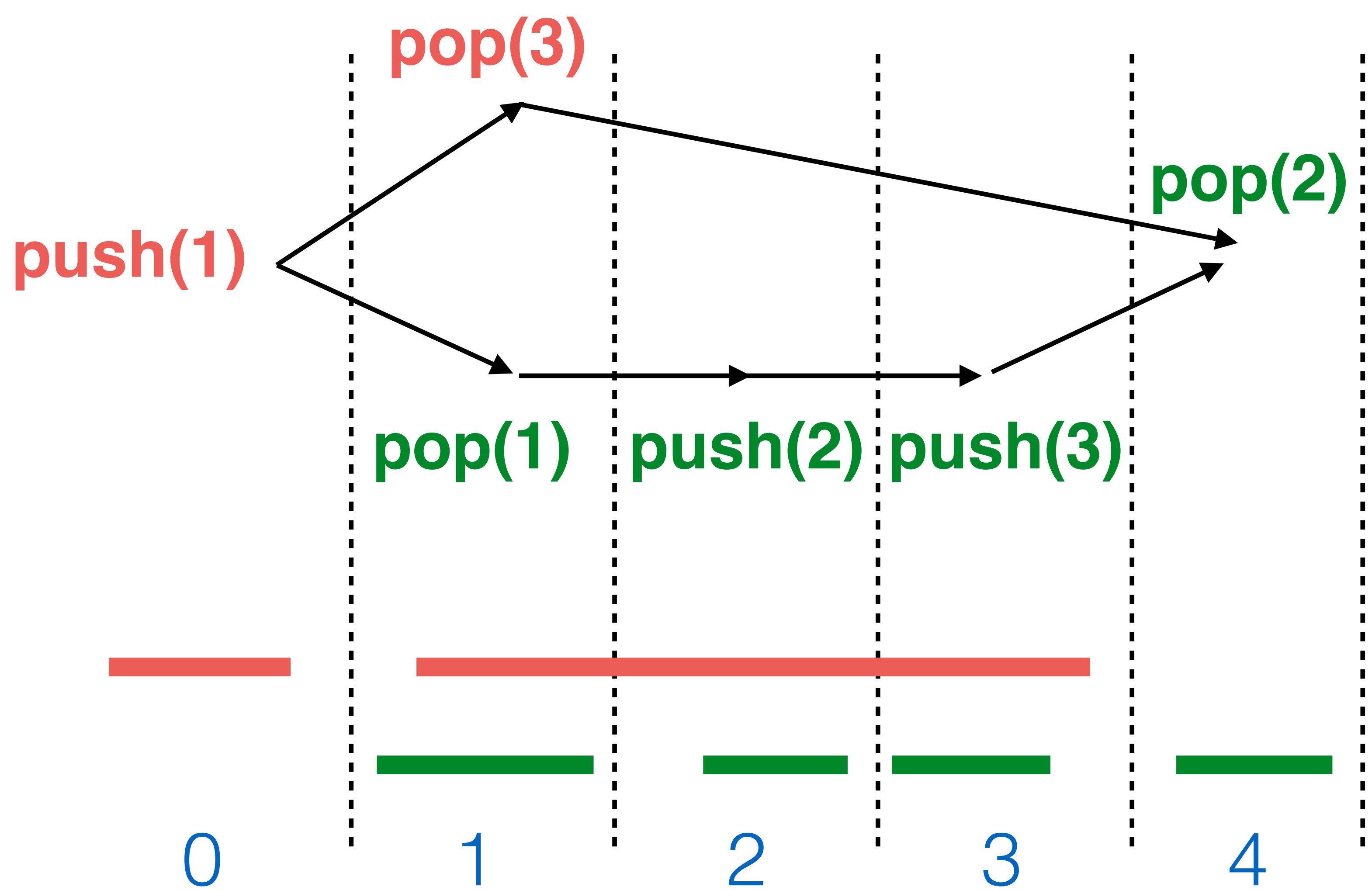
=> A_k keeps precise the information (bounds) about the k last intervals

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]

Counting Representation of Interval Orders

Count the number of occurrences
of each operation in each interval

- $h = (O, <)$ an IO with canonical representation $I{:}O \longrightarrow [n]^2$
- Then, let $\prod(h)$ be the multi-set **{** [label(o), I(o)] : o in O **}**
- Use a counter per type of operation and interval

Prop: H(e) is in H(L) iff $\prod$(H(e)) is in $\prod$(H(L))

Reduction to Reachability with Counters

$H(L_1)$ subset of $H(L_2)$
iff
$\Pi(H(L_1))$ subset of $\Pi(H(L_2))$

- Consider *only* k-bounded-length histories
- Track histories of L_1 using a finite number of counters
- Use an arithmetic-based representation of $\Pi(H(L_2))$
- Check that $\Pi(H(L_2))$ is invariant

- => Dynamic and static analysis

Reduction to Reachability with Counters

$H(L_1)$ subset of $H(L_2)$
iff
$\Pi(H(L_1))$ subset of $\Pi(H(L_2))$

- Consider *only* k-bounded-length histories
- Track histories of L_1 using a finite number of counters
- Use an arithmetic-based representation of $\Pi(H(L_2))$
- Check that $\Pi(H(L_2))$ is invariant
- => Dynamic and static analysis

How to get $\Pi(H(L))$?

Operation counting formulas

- Logic for expressing operation counting constraints

Presburger arithmetics with #(a, i, j) predicates
where

[#(a, i, j)](h) = |{o : I(o) = [i,j] and label(o) = a}|

Number of occurrences of a in the interval [i,j]

- h |= F is polynomial time (for a fixed quantifier count)

Building operation counting formulas

- For any k, a counting formula for $\prod(H(L))$ can be provided for common data structures (stack, queue, set,…)
- Assume data independence => push's have ≠ values

Violation of FIFO property (FV):

$$\exists\, x1,x2,y1,y2.$$
$$\mathrm{Match}(x1, y1)\ \&\ \mathrm{Match}(x2,y2)$$
$$\mathrm{Before}_K(x1,x2)\ \&\ \mathrm{Before}_K(y2, y1)$$

where

$$\mathrm{Match}(x,y) = \mathrm{IsPush}(x)\ \&\ \mathrm{IsPop}(y)\ \&\ \mathrm{SameVal}(x,y)$$

$$\mathrm{Before}_K(x,y) = \bigvee_{i \le k} (\mathrm{count}(x,0,i) > 0)\ \&\ \mathrm{count}(y,0,i)=0\ \&\ \mathrm{count}(x,i+1,k)=0)$$

where

$$\mathrm{count}(x,i,j) = \sum_{i \le i' \le j' \le j} \#(x,i',j')$$

Building operation counting formulas

FIFO queue data structure:

Violation of FIFO property (FV):

$$\exists\, x1,x2,y1,y2.\ \mathrm{Match}(x1, y1)\ \&\ \mathrm{Match}(x2,y2)\ \&\ \mathrm{Before}_K(x1,x2)\ \&\ \mathrm{Before}_K(y2, y1)$$

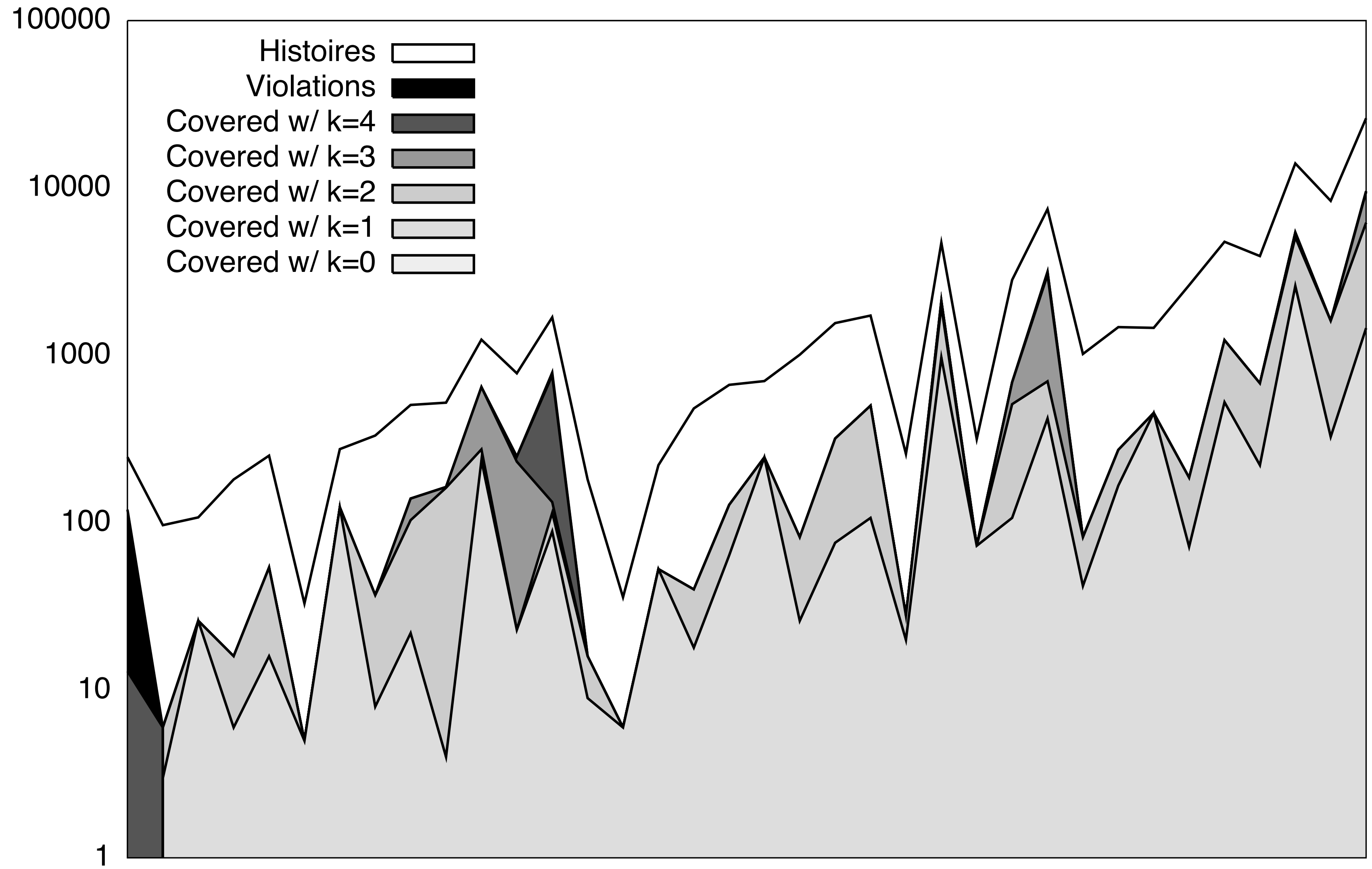
Violation of Remove property (RV):

$$\exists\, x,y.\ \mathrm{Match}(x, y)\ \&\ \mathrm{Before}_K(y, x)$$

Violation of Empty property (EV):

$$\exists\, x,y,z.\ \mathrm{Match}(x, z)\ \&\ \mathrm{EmptyPop}(y)\ \&\ \mathrm{Before}_K(y, z)\ \&\ \mathrm{Before}_K(y, z)$$

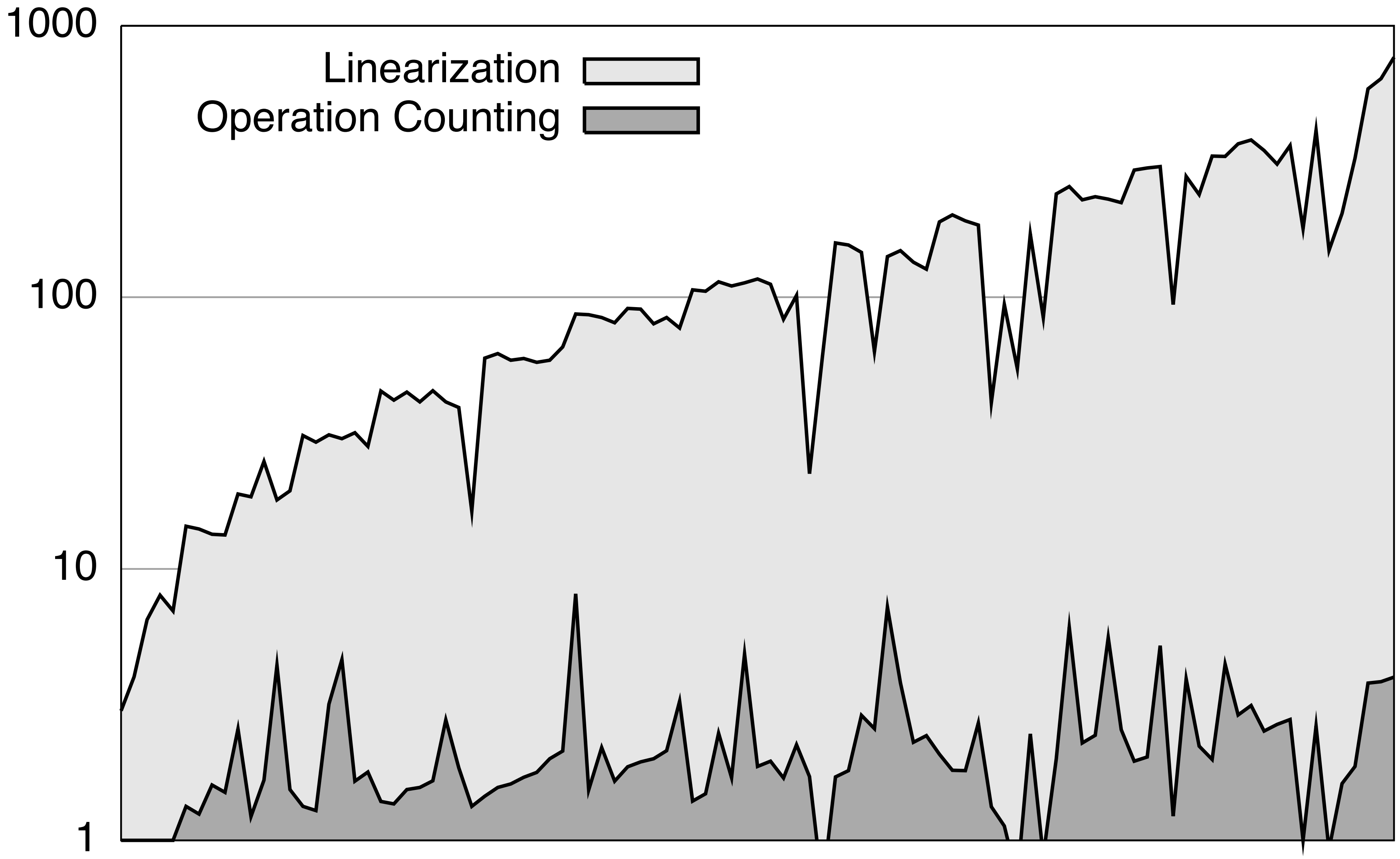
Experimental Results: Coverage



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”
- 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

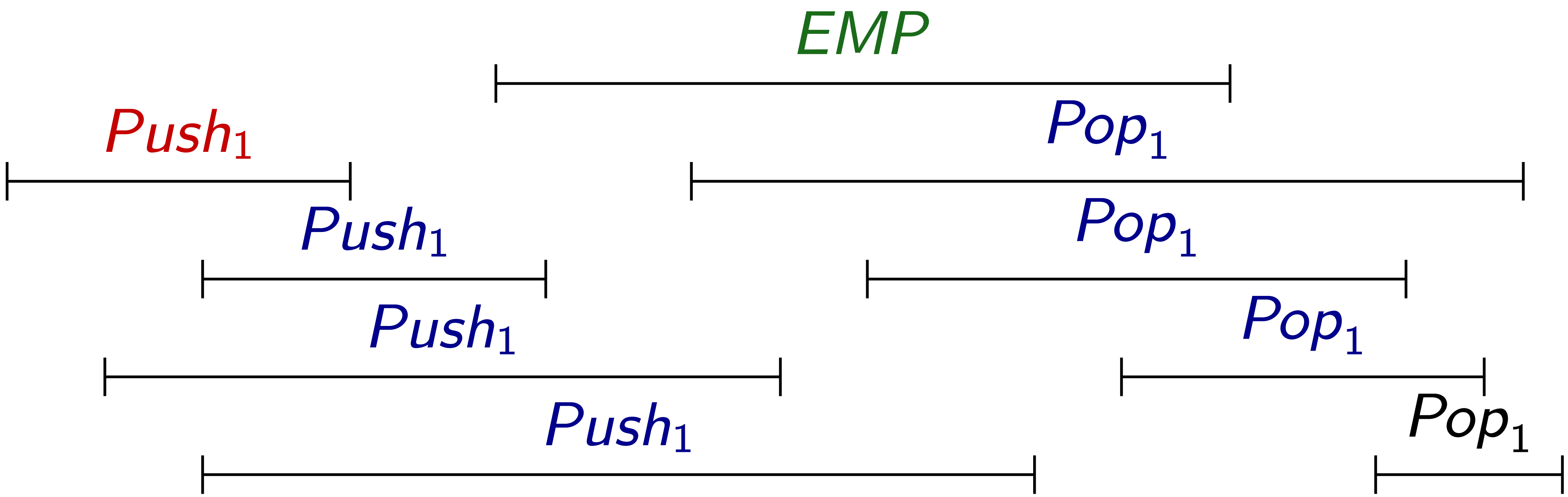
Order Violation

Enq(1) Deq(2)

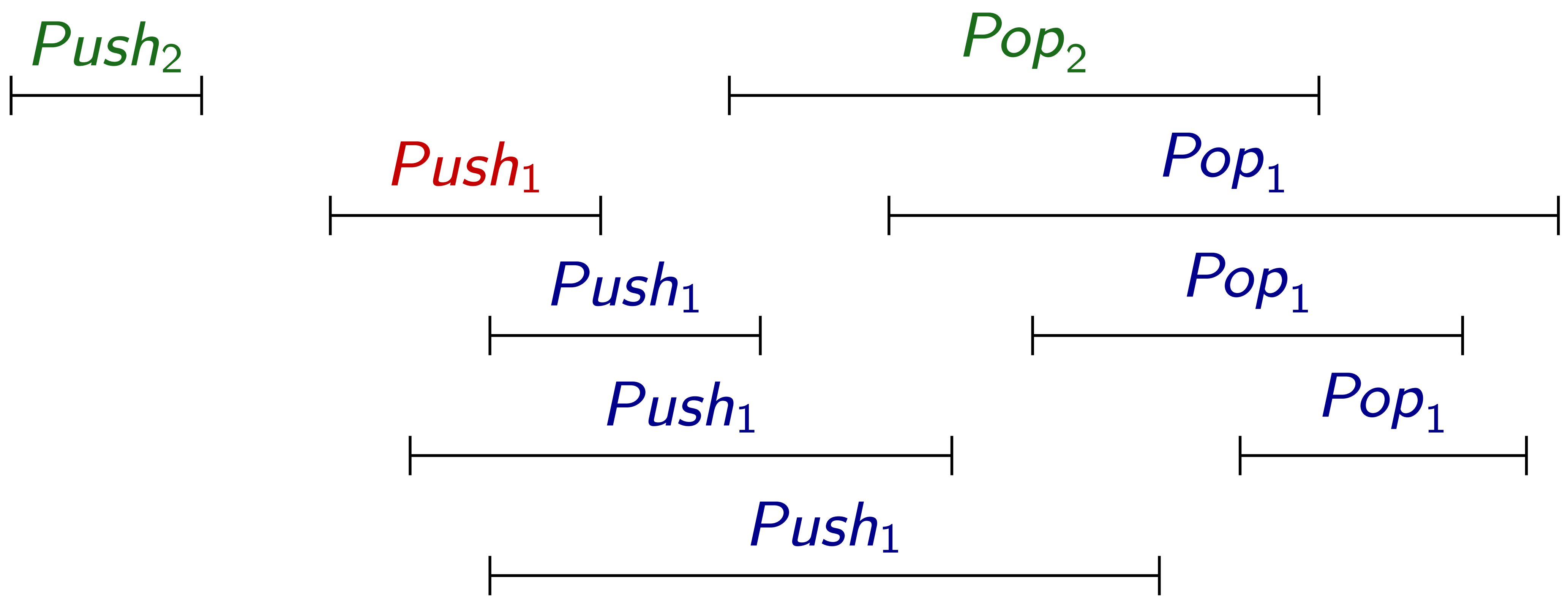
Enq(2) Deq(1)

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

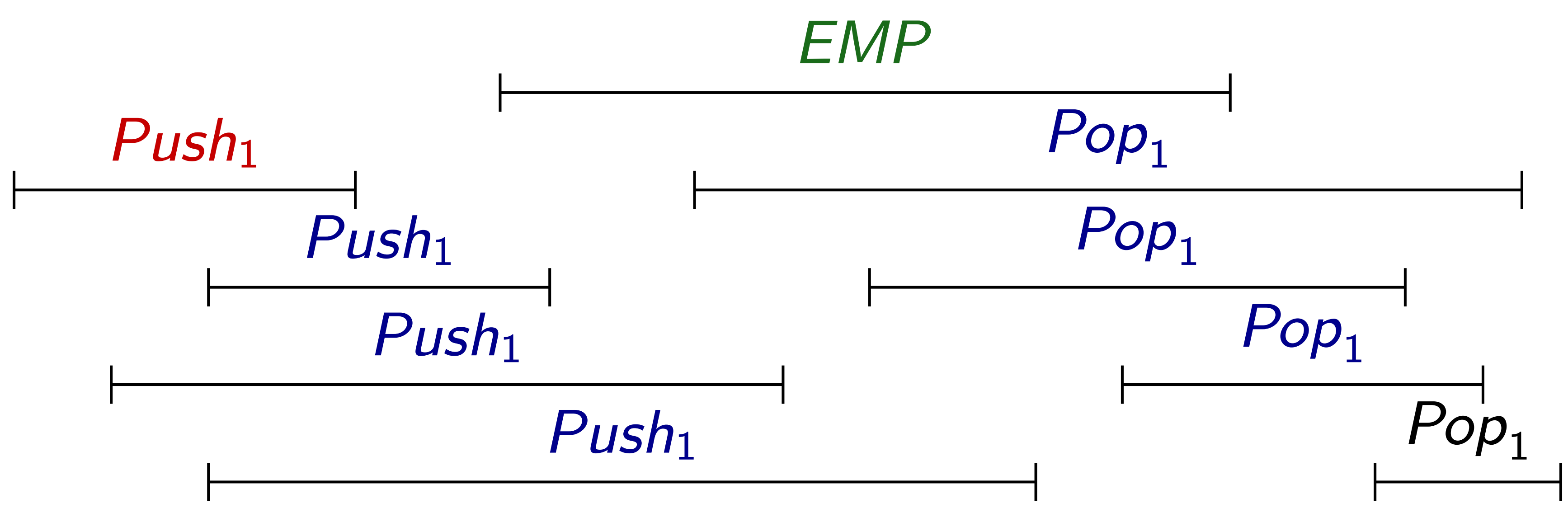
Empty Violation



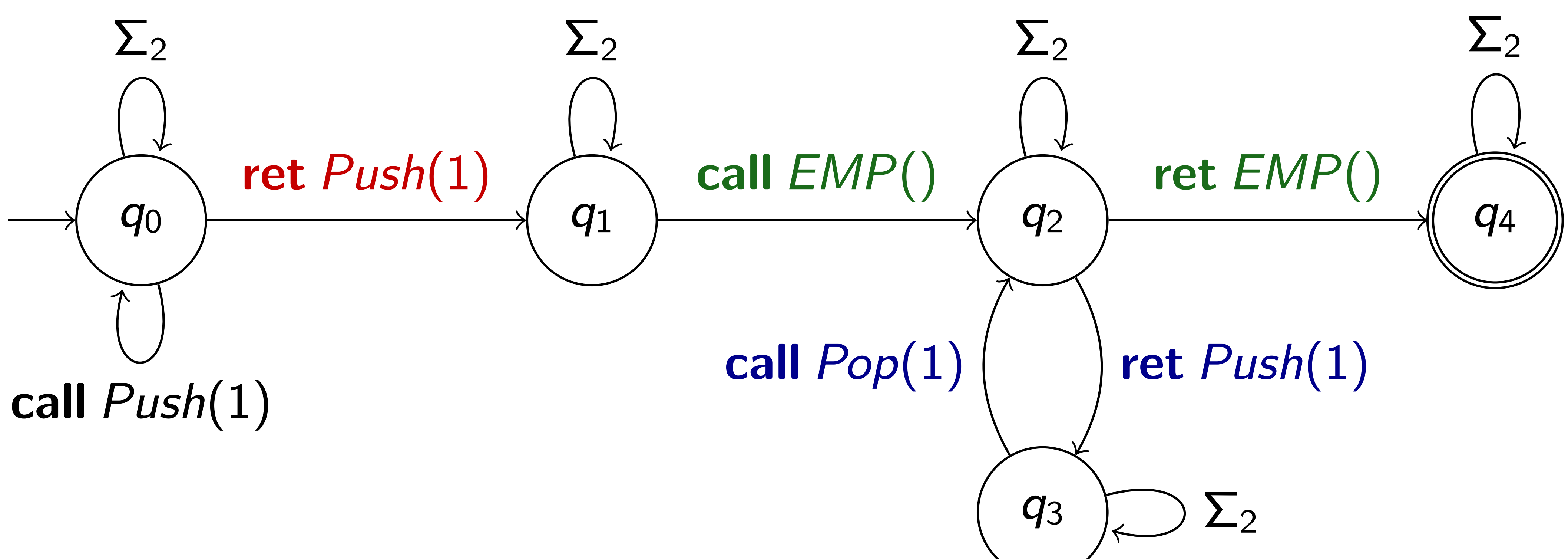
Order Violation cont.



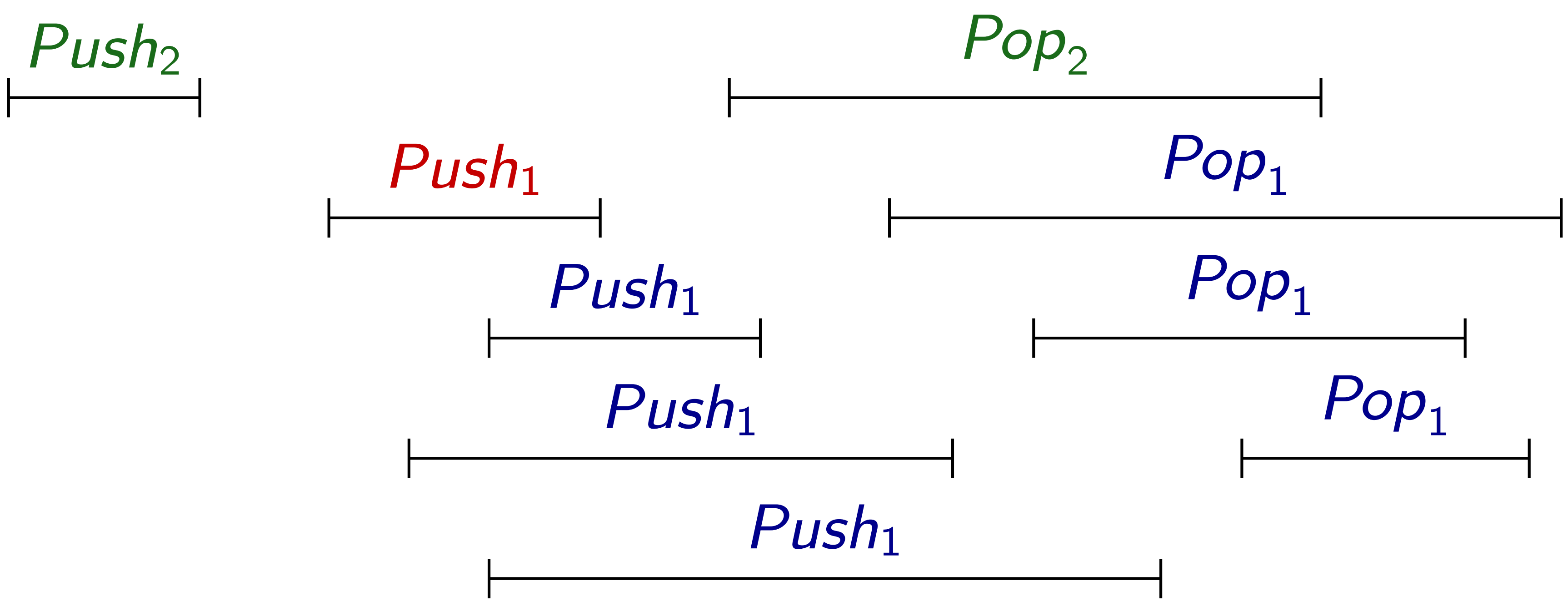
Automaton for Empty Violation



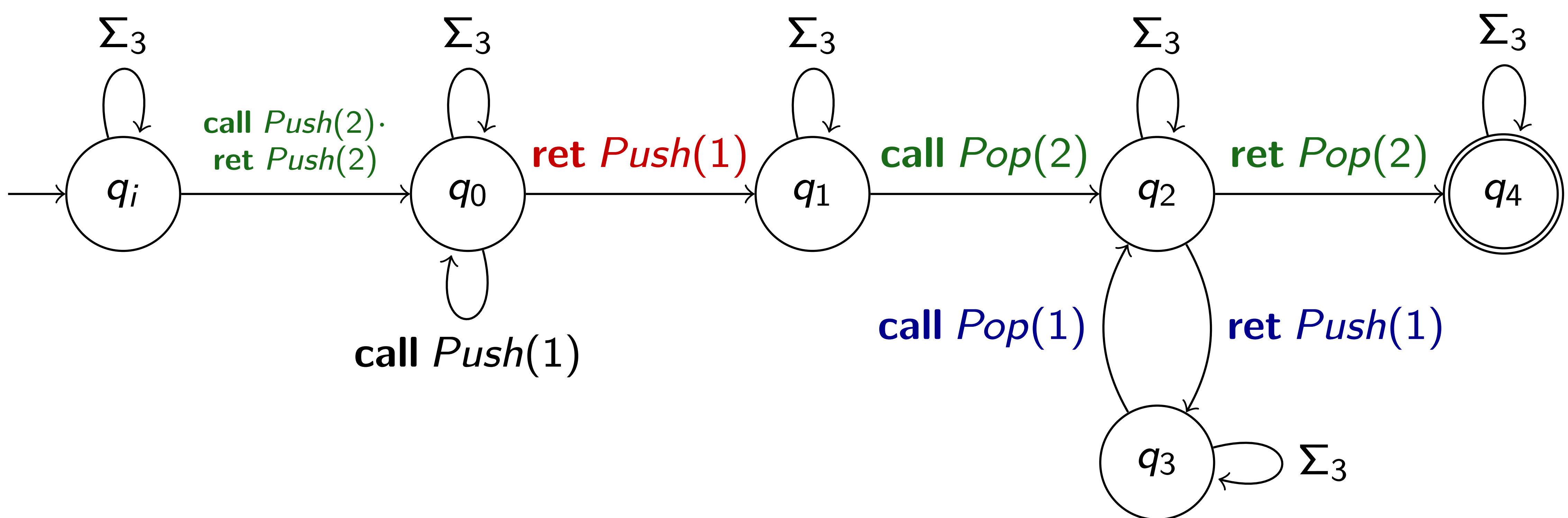
Recognized by:



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 object C, C is linearisable wrt S iff the intersection of H(C) wrt H(A(S)) is empty.

Same complexity as state reachability

Conclusion/Future work

- Bounding concept based on the notion of interval-length
- OR checking —> Reachability problem, using counting
- Suitable bounding concept: low complexity, small bounds
- Application in Dynamic and Static Analysis
- => Automatic synthesis of “specification”

- Reduction to SR for a common concurrent objects
- Reuse of existing verification technology
- => Extension to other kind of objects, e.g., sets
- => Abstractions / under-approximate analysis