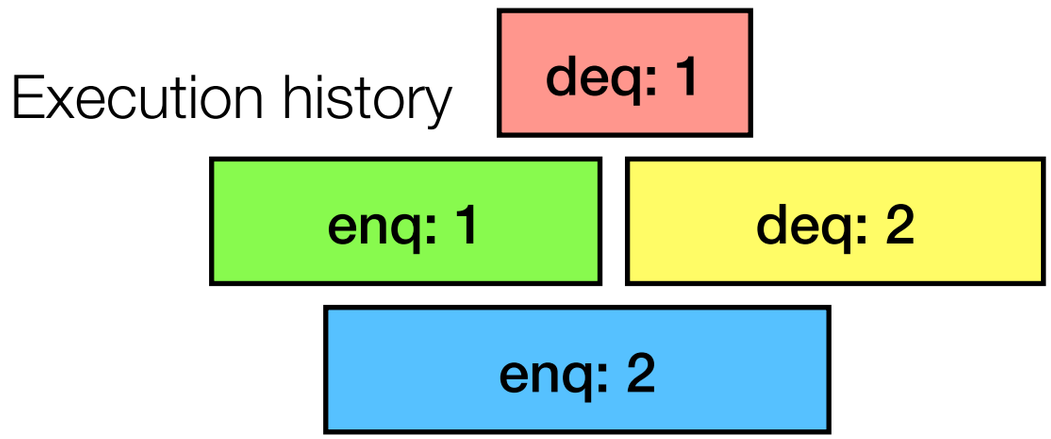
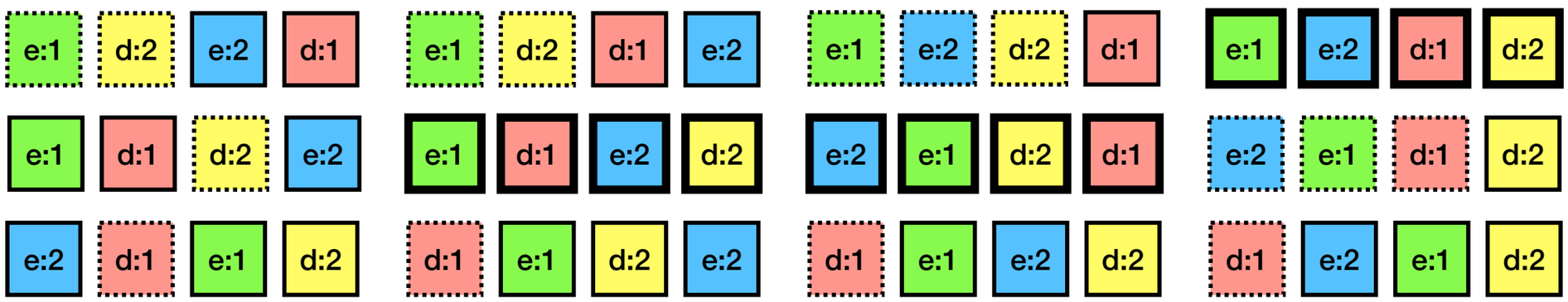
Checking Linearizability: Theoretical Limits

Constantin Enea
Ecole Polytechnique

Complexity of Testing Linearizability



Exponentially many linearizations to consider



Theorem [Gibbons.et.al.'97]

Checking linearizability for a fixed execution is NP-hard

Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]

Alur et al. 1996: Rajeev Alur, Kenneth L. McMillan, Doron A. Peled: Model-Checking of Correctness Conditions for Concurrent Objects. LICS 1996

Bouajjani et al., 2013: Ahmed Bouajjani, Michael Emmi, Constantin Enea, Jad Hamza: Verifying Concurrent Programs against Sequential Specifications. ESOP 2013

Hamza 2015: Jad Hamza: On the Complexity of Linearizability. NETYS 2015

Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]

Concurrent Languages

- Concurrent language = (Σ,D)
  - Σ an alphabet
  - $D \subseteq \Sigma \times \Sigma$

  (Mazurkiewicz traces - D is symmetric)

- a and b are called independent when $(a,b) \notin D$
- $\Rightarrow_D$ a relation that permutes independent symbols:
  - for all $(a,b) \notin D$, $\sigma\ ab\ \sigma' \Rightarrow_D \sigma'\ ba\ \sigma$ (and trans. closure)
- $cl_D(L)$ = all strings σ' such that $\sigma' \Rightarrow_D \sigma$ for some $\sigma \in L$
- Ex: Σ = {a,b}, L=(ab)*, D=∅ and D={(b,a)}

Specifications, Implementations

- Specification = a language over an alphabet containing symbols p:m(a)⇒b
- Example: bounded-value register, bounded size queue
- Implementation = a language over an alphabet containing symbols p:call m(a) and p:ret m(a)⇒b where returns “match” previous calls

- $\Sigma_p = (\Sigma_{call}(p) \cup \Sigma_{ret}(p))$
- $\Sigma = \cup_p \Sigma_p$
- the projection of every sequence in the implementation over Σ_p must belong to a language L(p) where there is a return between every two calls

Example: Treiber Stack

```
class Node {
  Node tl;
  int val;
}
```

```
class NodePtr {
  Node val;
} TOP;
```

```
void push(int e) {
  Node y, n;
  y = new();
  y->val = e;
  while(true) {
    n = TOP->val;
    y->tl = n;
    if (cas(TOP->val, n, y))
      break;
  }
}
```

```
int pop() {
  Node y,z;
  while(true) {
    y = TOP->val;
    if (y==0) return EMPTY;
    z = y->tl;
    if (cas(TOP->val, y, z))
      break;
  }
  return y->val;
}
```

What is the specification ?

Defining Linearizability

- lin = $\cup_p (\Sigma_p \times \Sigma_p) \cup (\Sigma_{ret} \times \Sigma_{call})$
- Spec* = replacing p:m(a)⇒b with call/ret actions
- an execution σ is **linearizable** iff $\sigma \in cl_{lin}(\text{Spec}^*)$
- Impl is linearizable iff Impl $\subseteq cl_{lin}(\text{Spec}^*)$
  - this inclusion check is undecidable in general (for regular languages)

Defining Linearizability

- Linearizability:
  - an execution σ is linearizable iff there exists a sequence τ that contains σ and linearization points (symbols p:m(a)⇒b) such that:
    - every projection over “actions” of the same process is “sequential”
    - the projection over linearization point actions is included in the specification

Defining Linearizability

- lin = $\cup_p$ (Σ_p x Σ_p) $\cup$ (Σ_{ret} x Σ_{call})
- Spec* = replacing p:m(a)⇒b with call/ret actions
- an execution σ is **linearizable** iff $\sigma \in cl_{lin}(Spec^*)$
- Impl is linearizable iff Impl $\subseteq cl_{lin}(Spec^*)$
  - this inclusion check is undecidable in general (for regular languages)
- $cl_{lin}(Spec^*) = (\|_p L_{lin\_points}(p) \| Spec) \downarrow (\Sigma_{call} \cup \Sigma_{ret})$

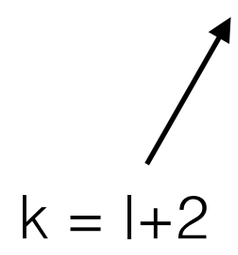
EXPSPACE-hardness

Problem 2 (Letter Insertion). Input: A set of *insertable* letters $A = \{a_1, \ldots, a_l\}$. An NFA N over an alphabet $\Gamma \uplus A$.

Question: For all words $w \in \Gamma^*$, does there exist a decomposition $w = w_0 \cdots w_l$, and a permutation p of $\{1, \ldots, l\}$, such that $w_0 a_{p[1]} w_1 \ldots a_{p[l]} w_l$ is accepted by N?

Reducing Letter Insertion to Linearizability:

1. there exists a word w in Γ^*, such that there is no way to insert the letters from A in order to obtain a word accepted by N
2. there exists an execution of Lib with k threads which is not linearizable w.r.t. S_N



EXPSPACE-hardness

Define k, the number of threads, to be $l + 2$.
We will define a library Lib composed of

- methods $M_1, \ldots, M_l$, one for each letter of A
- methods M_γ, one for each letter of Γ
- a method M_{Tick}.

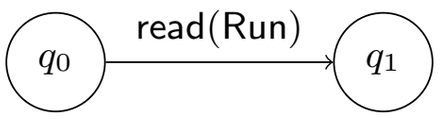


Fig. 4. Description of M_γ, $\gamma \in \Gamma$

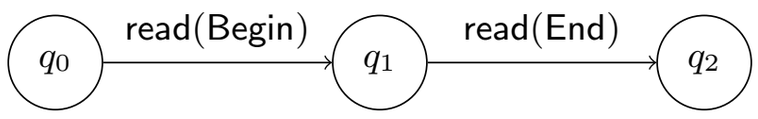


Fig. 5. Description of $M_1, \ldots, M_l$

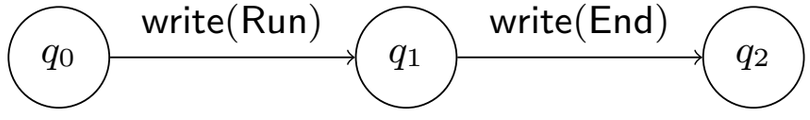


Fig. 6. Description of M_{Tick}

The specification S_N is defined as the set of words w over the alphabet $\{M_1, \ldots, M_l\} \cup \{M_{\mathsf{Tick}}\} \cup \{M_\gamma | \gamma \in \Gamma\}$ such that one the following condition holds:

- w contains 0 letter M_{Tick}, or more than 1, or
- for a letter M_i, $i \in \{1, \ldots, l\}$, w contains 0 such letter, or more than 1, or
- when projecting over the letters M_γ, $\gamma \in \Gamma$ and M_i, $i \in \{1, \ldots, l\}$, w is in N_M, where N_M is N where each letter γ is replaced by the letter M_γ, and where each letter a_i is replaced by the letter M_i.

EXPSPACE-hardness

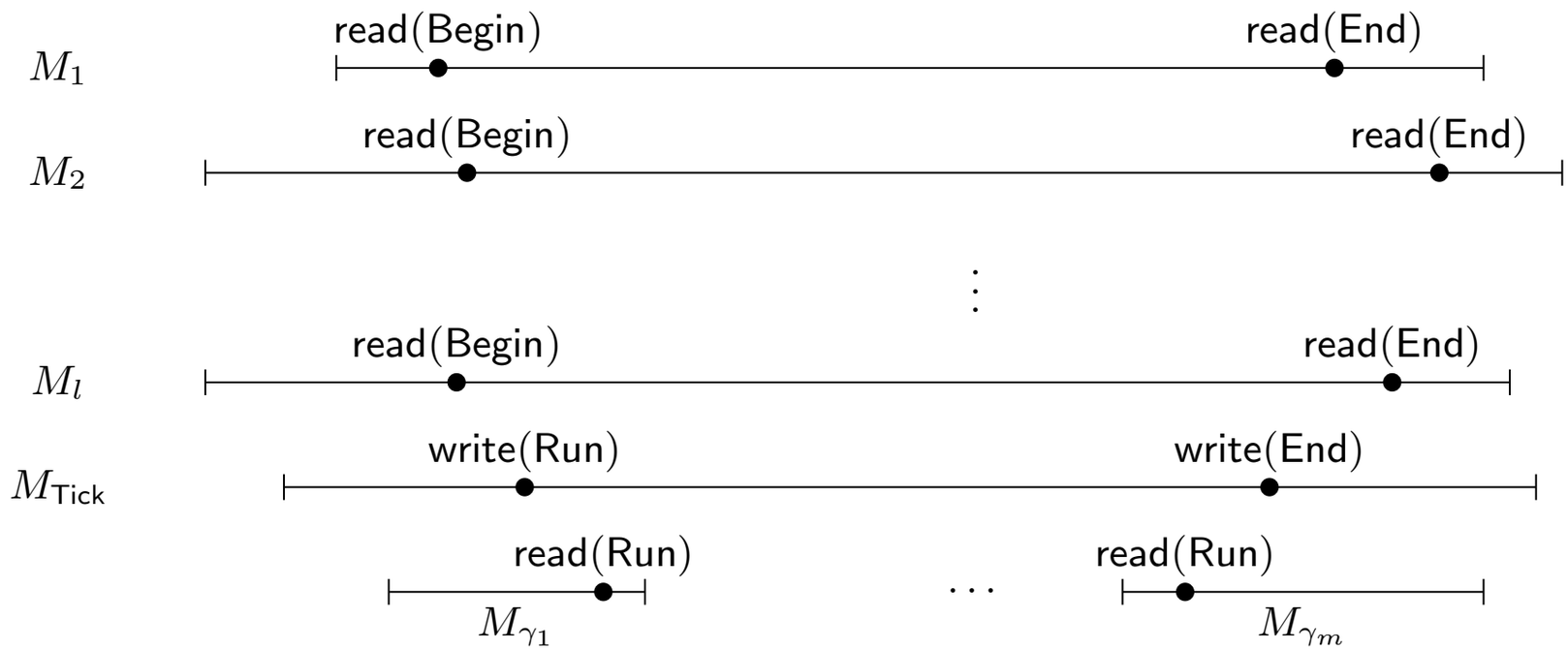


Fig. 7. Non-linearizable execution corresponding to a word $\gamma_1 \dots \gamma_m$ in which we cannot insert the letters from $A = \{a_1, \dots, a_l\}$ to make it accepted by N. The points represent steps in the automata.

Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

- Undecidable [Bouajjani et al., 2013]

- Decidable with “fixed linearization points” [Bouajjani et al. 2013]

Undecidability

- Reduction from reachability in counter machines
- Given a counter machine A, we construct a library L_A and a specification S_A such that L_A is **not** linearizable w.r.t. S_A iff A reaches the target state
- L_A = transition methods T[t], increments I[c_i], decrements D[c_i] and zero-tests Z[c_i]
- L_A allows only valid sequences of transitions
- S_A allows executions which don't reach the target state, or which erroneously pass some zero-test

  – it doesn't contain $\mathtt{M}[q_f]$,
  – it ends in $\mathtt{M}[q_f]$ and it contains a prefix of the form

$$(\mathtt{M\_inc}[i]\,\mathtt{M\_dec}[i])^*(\mathtt{M\_inc}[i]^+ + \mathtt{M\_dec}[i]^+)\mathtt{M\_zero}[i]$$

  – it ends in M_f and it contains a subword of the form

$$\mathtt{M\_zero}[i](\mathtt{M\_inc}[i]\,\mathtt{M\_dec}[i])^*(\mathtt{M\_inc}[i]^+ + \mathtt{M\_dec}[i]^+)\mathtt{M\_zero}[i].$$

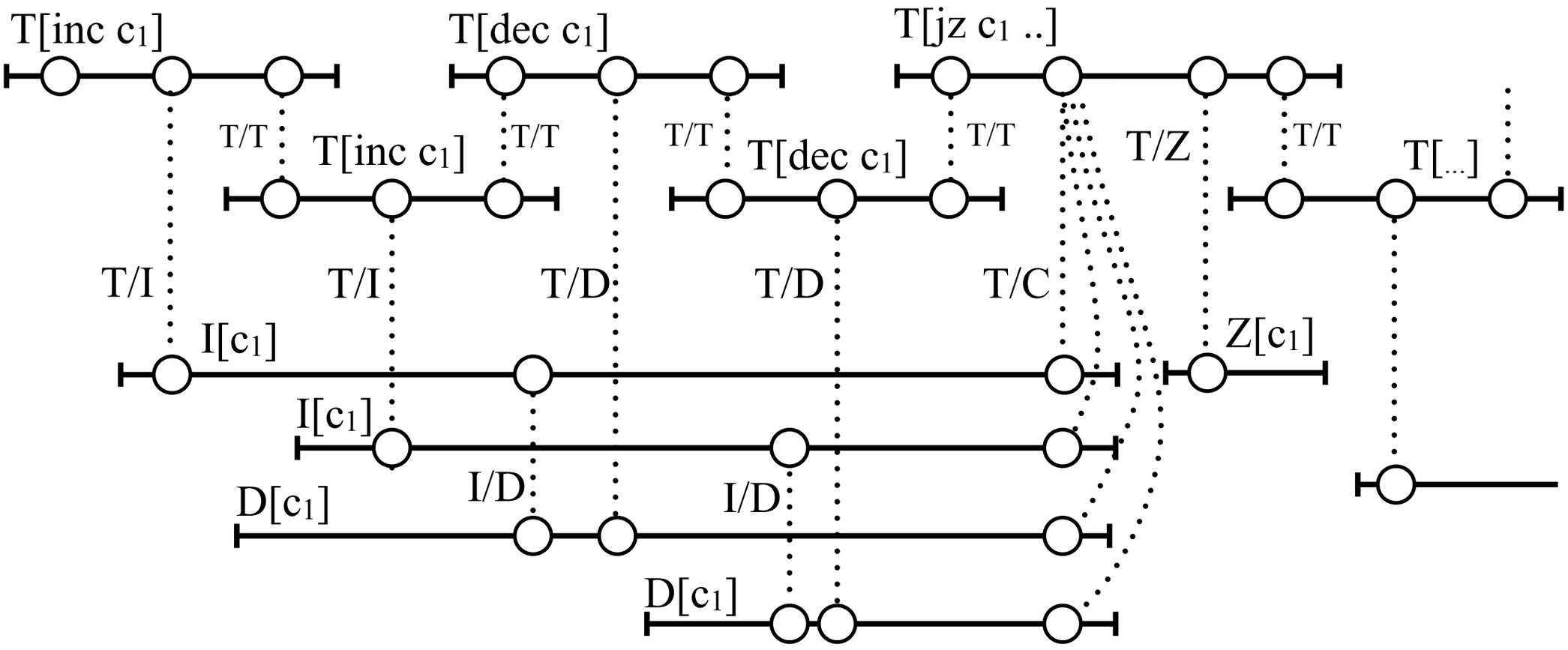
Undecidability

1. A sequence $t_1 t_2 \ldots t_i$ of $\mathcal{A}$-transitions is modeled by a pairwise-overlapping sequence of $\mathrm{T}[t_1] \cdot \mathrm{T}[t_2] \cdots \mathrm{T}[t_i]$ operations.
2. Each $\mathrm{T}[t]$-operation has a corresponding $\mathrm{I}[c_i]$, $\mathrm{D}[c_i]$, or $\mathrm{Z}[ci]$ operation, depending on whether t is, resp., an increment, decrement, or zero-test transition with counter c_i.
3. Each $\mathrm{I}[c_i]$ operation has a corresponding $\mathrm{D}[c_i]$ operation.
4. For each counter c_i, all $\mathrm{I}[c_i]$ and $\mathrm{D}[c_i]$ between $\mathrm{Z}[c_i]$ operations overlap.
5. For each counter c_i, no $\mathrm{I}[c_i]$ nor $\mathrm{D}[c_i]$ operations overlap with a $\mathrm{Z}[c_i]$ operation.
6. The number of $\mathrm{I}[c_i]$ operations between two $\mathrm{Z}[c_i]$ operations matches the number of $\mathrm{D}[c_i]$ operations.

Undecidability

- a T/T signal between T[*] operations
- for each counter c, a T/I, T/D, T/Z between T[*] operations and, resp., $I[c_i]$, $D[c_i]$ and $Z[c_i]$ operations
- an I/D signal between $I[c_i]$ and $D[c_i]$ operations
- a T/C signal between T[t] operations and $I[c_i]$, $D[c_i]$ operations, for zero-testing transitions t

Undecidability



Undecidability

```
1  var q ∈ Q: T
2  var req[U]: T
3  var ack[U]: T
4  var dec[i ∈ ℕ : i < d]: T
5  var zero[i ∈ ℕ : i < d]: 𝔹
6
7  // for each transition ⟨q, n, q'⟩
8  method M[q, n, q']()
9      atomic
10         wait(q);
11         signal(req[n]);
12     atomic
13         wait(ack[n]);
14         signal(q');
15     return ()
16
17 // for each transition ⟨q, i, q'⟩
18 method M[q, i, q'] ()
19     atomic
20         wait(q);
21         zero[i] := true;
22     atomic
23         if !zero[i] then
24             signal(q');
25     return ()
26
27 // for each final state q_f
28 method M[q_f] ()
29     wait(q_f);
30     return
31 method M_inc[i] ()
32     atomic
33         if !zero[i] then
34             wait(req[u_i]);
35             signal(ack[u_i]);
36             signal(dec[i])
37     assume zero[i];
38     return ()
39
40 method M_dec[i] ()
41     atomic
42         if !zero[i] then
43             wait(dec[i]);
44     atomic
45         wait(req[−u_i]);
46         signal(ack[−u_i]);
47     assume zero[i];
48     return ()
49
50 method M_zero[i] ()
51     atomic
52         if zero[i] then
53             zero[i] := false;
54     return ()
```

Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]

Libraries

A *method* is a finite automaton $M = \langle Q, \Sigma, I, F, \hookrightarrow \rangle$ with labeled transitions $\langle m_1, v_1 \rangle \xhookrightarrow{a} \langle m_2, v_2 \rangle$ between method-local states $m_1, m_2 \in Q$ paired with finite-domain shared-state valuations $v_1, v_2 \in V$. The initial and final states $I, F \subseteq Q$ represent the method-local states passed to, and returned from, M.

A *client* of a library L is a finite automaton $C = \langle Q, \Sigma, \ell_0, \hookrightarrow \rangle$ with initial state $\ell_0 \in Q$ and transitions $\hookrightarrow \subseteq Q \times \Sigma \times Q$ labeled by the alphabet $\Sigma = \{M(m_0, m_f) : M \in L, m_0, m_f \in Q_M\}$ of library method calls

most general client $C^\star = \langle Q, \Sigma, \ell_0, \hookrightarrow \rangle$ of a library L nondeterministically calls L's methods in any order: $Q = \{\ell_0\}$ and $\hookrightarrow = Q \times \Sigma \times Q$.

Example

```
class Node {
  Node tl;
  int val;
}
```

```
class NodePtr {
  Node val;
} TOP;
```

```
void push(int e) {
  Node y, n;
  y = new();
  y->val = e;
  while(true) {
    n = TOP->val;
    y->tl = n;
    if (cas(TOP->val, n, y))
      break;
  }
}
```

```
int pop() {
  Node y,z;
  while(true) {
    y = TOP->val;
    if (y==0) return EMPTY;
    z = y->tl;
    if (cas(TOP->val, y, z))
      break;
  }
  return y->val;
}
```

Libraries

A *configuration* $c = \langle v, u \rangle$ of $L[C]$ is a shared memory valuation $v \in V$, along with a map u mapping each thread $t \in \mathbb{N}$ to a tuple $u(t) = \langle \ell, m_0, m \rangle$, composed of a client-local state $\ell \in Q_C$, along with initial and current method states $m_0, m \in Q_L \cup \{\bot\}$; $m_0 = m = \bot$ when thread t is not executing a library

$$\textsc{Internal}\quad \frac{u_1(t) = \langle \ell, m_0, m_1 \rangle \quad \langle m_1, v_1 \rangle \xhookrightarrow{a} \langle m_2, v_2 \rangle \quad u_2 = u_1\,(t \mapsto \langle \ell, m_0, m_2 \rangle)}{\langle v_1, u_1 \rangle \xrightarrow[L[C]]{\langle a, t \rangle} \langle v_2, u_2 \rangle}$$

$$\textsc{Call}\quad \frac{u_1(t) = \langle \ell_1, \bot, \bot \rangle \quad m_0 \in I_M \quad \ell_1 \xhookrightarrow{M(m_0, m_f)}_{\mathrm{C}} \ell_2 \quad u_2 = u_1\,(t \mapsto \langle \ell_1, m_0, m_0 \rangle)}{\langle v, u_1 \rangle \xrightarrow[L[C]]{\mathsf{call}(M, m_0, t)} \langle v, u_2 \rangle}$$

$$\textsc{Return}\quad \frac{u_1(t) = \langle \ell_1, m_0, m_f \rangle \quad m_f \in F_M \quad \ell_1 \xhookrightarrow{M(m_0, m_f)}_{\mathrm{C}} \ell_2 \quad u_2 = u_1\,(t \mapsto \langle \ell_2, \bot, \bot \rangle)}{\langle v, u_1 \rangle \xrightarrow[L[C]]{\mathsf{ret}(M, m_f, t)} \langle v, u_2 \rangle}$$

Fig. 1. The transition relation $\rightarrow_{L[C]}$ for the library-client composition $L[C]$.

VASS model

We associate to each concurrent system $L[C]$ a *canonical VASS*,[2] denoted $\mathcal{A}_{L[C]}$, whose states are the set of shared-memory valuations, and whose vector components count the number of threads in each thread-local state; a transition of $\mathcal{A}_{L[C]}$ from $\langle v_1, \boldsymbol{n}_1 \rangle$ to $\langle v_2, \boldsymbol{n}_2 \rangle$ updates the shared-memory valuation from v_1 to v_2 and the local state of some thread t from $u_1(t)$ to $u_2(t)$ by decrementing the $u_1(t)$-component of $\boldsymbol{n}_1$, and incrementing the $u_2(t)$-component, to derive $\boldsymbol{n}_2$.

Specifications

A *specification* S of a library L is a language over the *specification alphabet*

$$\Sigma_S \stackrel{\text{def}}{=} \{M[m_0, m_f] : M \in L, m_0, m_f \in Q_M\}.$$

Definition 2 (Linearizability [20]). *A trace τ is* S-linearizable *when there exists a completion*[4] *π of a strict, serial permutation of τ such that $(\pi \mid S) \in S$.*

Specifications

The *pending closure* of a specification S, denoted $\overline{S}$ is the set of S-images of serial sequences which have completions whose S-images are in S:

$$\overline{S} \stackrel{\text{def}}{=} \{(\sigma \mid S) \in \overline{\Sigma}_S^* : \exists \sigma' \in \Sigma_S^*.\ (\sigma' \mid S) \in S \text{ and } \sigma' \text{ is a completion of } \sigma\}.$$

Specifications

The *pending closure* of a specification S, denoted $\overline{S}$ is the set of S-images of serial sequences which have completions whose S-images are in S:

$$\overline{S} \stackrel{\text{def}}{=} \{(\sigma \mid S) \in \overline{\Sigma}_S^* : \exists \sigma' \in \Sigma_S^*.\ (\sigma' \mid S) \in S \text{ and } \sigma' \text{ is a completion of } \sigma\}.$$

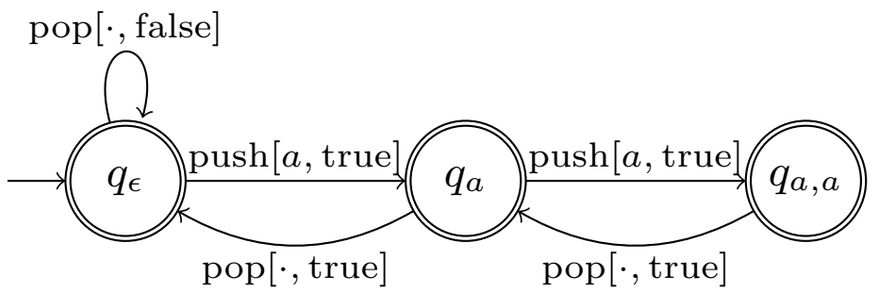


Fig. 2. The sequential specification of two-element stacks containing the (abstract) value a, given as the language of a finite automaton, whose operation alphabet indicates both the argument and return values.

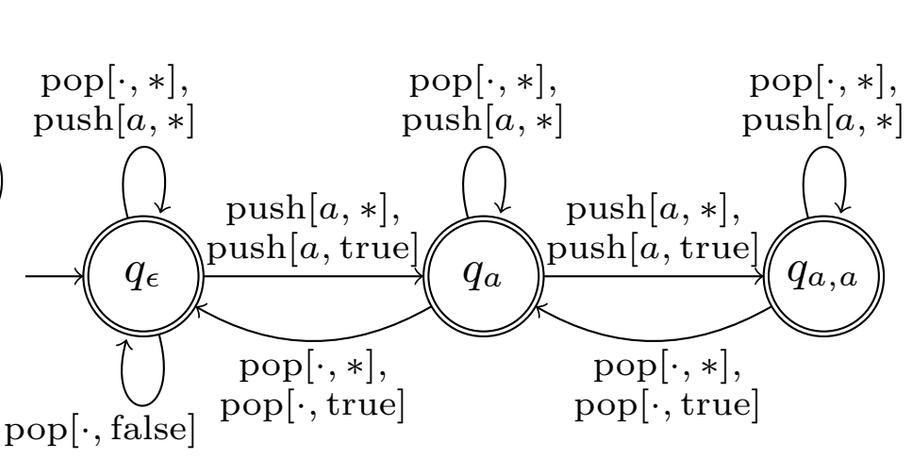


Fig. 3. The pending closure of the stack specification from Figure 2.

Specifications

The *pending closure* of a specification S, denoted $\overline{S}$ is the set of S-images of serial sequences which have completions whose S-images are in S:

$$\overline{S} \stackrel{\text{def}}{=} \{(\sigma \mid S) \in \overline{\Sigma}_S^* : \exists \sigma' \in \Sigma_S^*.\ (\sigma' \mid S) \in S \text{ and } \sigma' \text{ is a completion of } \sigma\}.$$

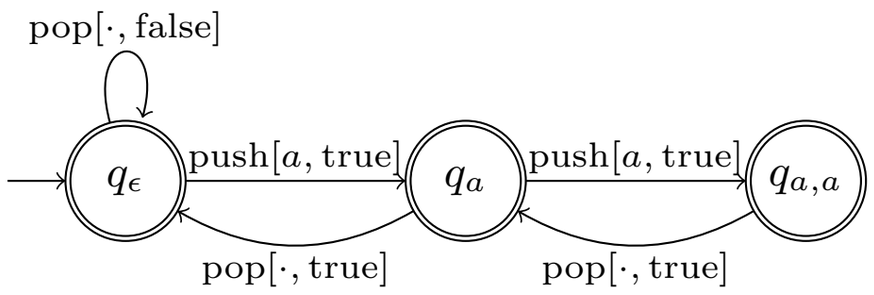


Fig. 2. The sequential specification of two-element stacks containing the (abstract) value a, given as the language of a finite automaton, whose operation alphabet indicates both the argument and return values.

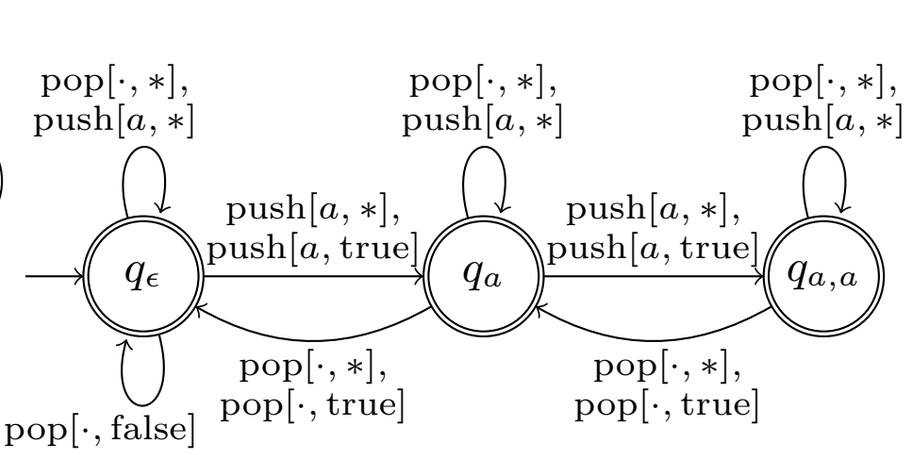


Fig. 3. The pending closure of the stack specification from Figure 2.

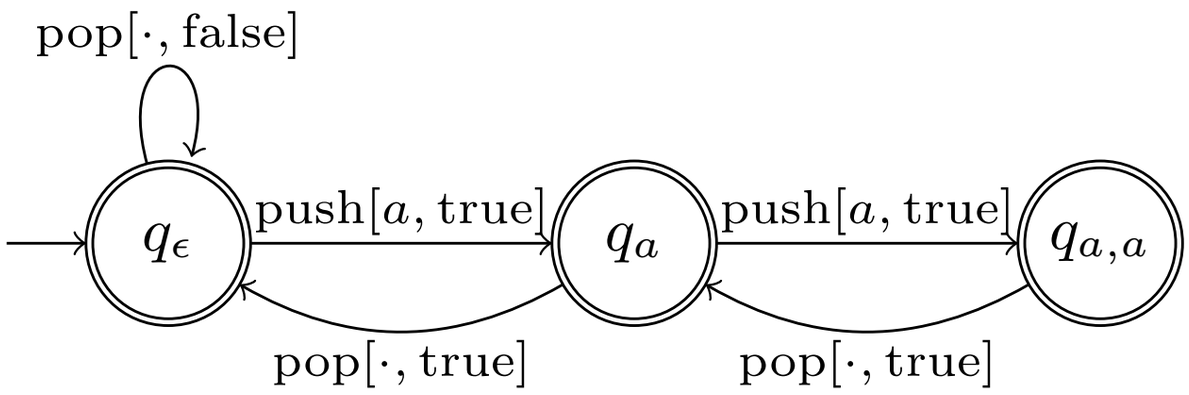
Lemma 1. *The pending closure $\overline{S}$ of a regular specification S is regular.*

Lemma 2. *A trace τ is S-linearizable if and only if there exists a strict, serial permutation π of τ such that $(\pi \mid S) \in \overline{S}$.*

Read-only operations

Given a method M of a library L and $m_0, m_f \in Q_M$, an $M[m_0, m_f]$-operation θ is *read-only* for a specification S if and only if for all $w_1, w_2, w_3 \in \Sigma_S^*$,

1. If $w_1 \cdot M[m_0, m_f] \cdot w_2 \in S$ then $w_1 \cdot M[m_0, m_f]^k \cdot w_2 \in S$ for all $k \geq 0$, and
2. If $w_1 \cdot M[m_0, m_f] \cdot w_2 \in S$ and $w_1 \cdot w_3 \in S$ then $w_1 \cdot M[m_0, m_f] \cdot w_3 \in S$.



Linearization points

The *control graph* $G_M = \langle Q_M, E \rangle$ is the quotient of a method M's transition system by shared-state valuations V: $\langle m_1, a, m_2 \rangle \in E$ iff $\langle m_1, v_1 \rangle \hookrightarrow^{\mathrm{a}}_{\mathrm{M}} \langle m_2, v_2 \rangle$ for some $v_1, v_2 \in V$. A function $\mathsf{LP} : L \to \wp(\Sigma_L)$ is called a *linearization-point mapping* when for each $M \in L$:

1. each symbol $a \in \mathsf{LP}(M)$ labels at most one transition of M,
2. any directed path in G_M contains at most one symbol of $\mathsf{LP}(M)$, and
3. all directed paths in G_M containing $a \in \mathsf{LP}(M)$ reach the same $m_a \in F_M$.

An action $\langle a, i \rangle$ of an M-operation is called a *linearization point* when $a \in \mathsf{LP}(M)$, and operations containing linearization points are said to be *effectuated*; $\mathsf{LP}(\theta)$ denotes the unique linearization point of an effectuated operation θ. A *read-points mapping* $\mathsf{RP} : \Theta \to \mathbb{N}$ for an action sequence σ with operations Θ maps each read-only operation θ to the index $\mathsf{RP}(\theta)$ of an internal θ-action in σ.

Fixed Linearization Points

- **Fixed** linearization points: the linearization point is fixed to a particular statement in the code

```
class Node {
  Node tl;
  int val;
}
```

```
class NodePtr {
  Node val;
} TOP
```

Treiber Stack

```
void push(int e){
  Node y, n;
  y = new();
  y->val = e;
  while(true) {
    y->tl = n;
    if (cas(TOP->val, n, y))
      break;
  }
}
```

```
int pop(){
  Node y,z;
  while(true) {
    y = TOP->val;
    if (y==0) return EMPTY;
    z = y->tl;
    if (cas(TOP->val, y, z))
      break;
  }
  return y->val;
}
```

Exercices (1)

- Does the Herlihy & Wing queue admit **fixed** linearization points ?

```
void enq(int x) {
  i = back++; items[i] = x;
}
int deq() {
  while (1) {
    range = back - 1;
    for (int i = 0; i <= range; i++) {
      x = swap(items[i],null);
      if (x != null) return x;
} } }
```

Static linearizability

An action sequence σ is called *effectuated* when every completed operation of σ is either effectuated or read-only, and an effectuated completion σ' of σ is *effect preserving* when each effectuated operation of σ also appears in σ'. Given a linearization-point mapping LP, and a read-points mapping RP of an action sequence σ, we say a permutation π of σ is *point preserving* when every two operations of π are ordered by their linearization/read points in σ.

Definition 4. *A trace τ is* $\langle S, \mathsf{LP}\rangle$-linearizable *when τ is effectuated, and there exists a read-points mapping* RP *of τ, along with an effect-preserving completion π of a strict, point-preserving, and serial permutation of τ such that $(\pi \mid S) \in S$.*

Definition 5 (Static Linearizability). *The system $L[C]$ is* S-static linearizable *when $L[C]$ is $\langle S, \mathsf{LP}\rangle$-linearizable for some mapping* LP.

Checking Static Linerizability

- A_S = a deterministic automaton recognizing the Specification
- we define a monitor to be composed with L[C] that simulates the Specification
  - methods have a new local variable RO which is initially ∅ (records return values of read-only operations)
    - if mf ∈ RO in an invocation of M, then M[m0,mf] is read-only and a state of A_S in which $M[m_0,m_f]$ is enabled has been observed
  - L[C] executes a linearization point => the state of the Specification is advanced to the $M[m_0,m_f]$ successor (m_0 is the initial state of the current operation and m_f is the unique final state reachable from this lin. point)
  - L[C] executes an internal action from an $M[m_0,*]$ operation => RO is enriched with every m_f such that $M[m_0,m_f]$ is read-only and enabled in the current specification state
  - L[C] executes the return of an $M[m_0,m_f]$ read-only operation => if $m_f \notin$ RO then the monitor goes to an error state

EXPSPACE-hardness

- Reduce control state reachability in VASS (which is EXPSPACE-complete) to static linearizability
  - Use the library from the undecidability proof without the zero-test method (the specification excludes only executions not reaching the target state)

Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]

Alur et al. 1996: Rajeev Alur, Kenneth L. McMillan, Doron A. Peled: Model-Checking of Correctness Conditions for Concurrent Objects. LICS 1996

Bouajjani et al., 2013: Ahmed Bouajjani, Michael Emmi, Constantin Enea, Jad Hamza: Verifying Concurrent Programs against Sequential Specifications. ESOP 2013

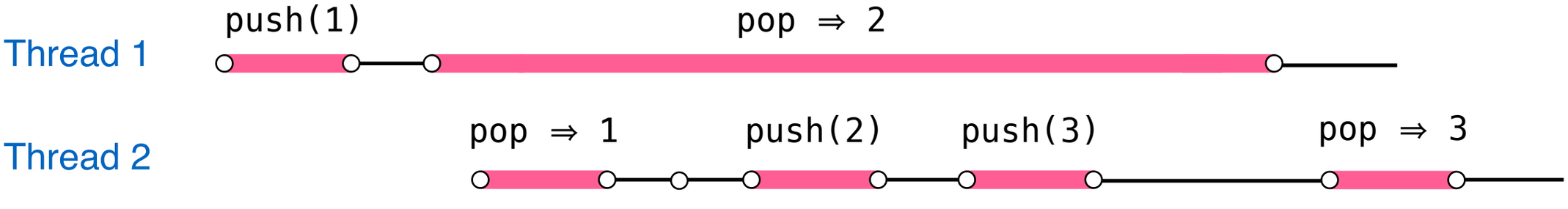
Hamza 2015: Jad Hamza: On the Complexity of Linearizability. NETYS 2015

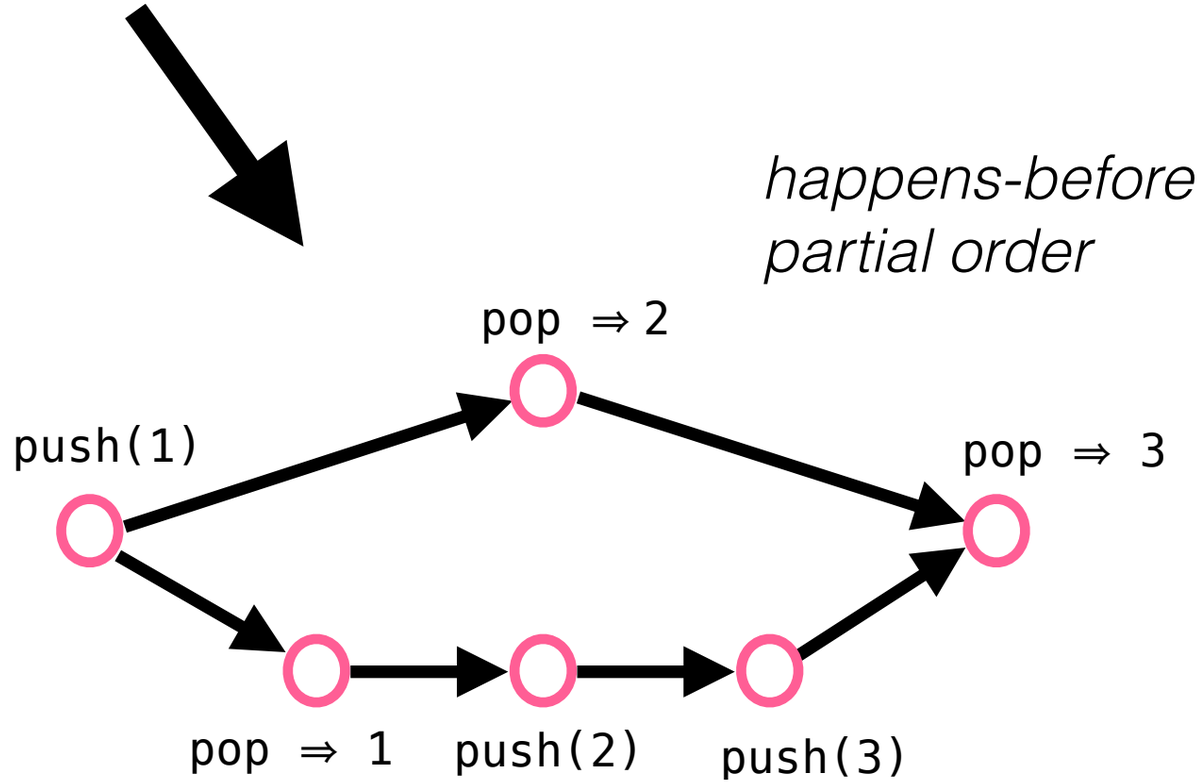
Reducing Linearizability to Reachability

Checking Lin. using “bad patterns”

- Reduce linearizability checking to reachability (EXPSPACE-complete):
  - Define (sequential) data-structure S using inductive rules
  - S is data independent and closed under projection
  - Characterize sequential executions of S using bad patterns
  - Characterize concurrent executions, linearizable w.r.t. S using bad patterns (one per rule)
  - Define a regular automaton A_i for each bad pattern
  - Reduce“L is linearizable w.r.t. S” to: for all i, $L \cap A_i = \varnothing$

Histories = Posets of events





Concurrent Queues [ICALP'15]

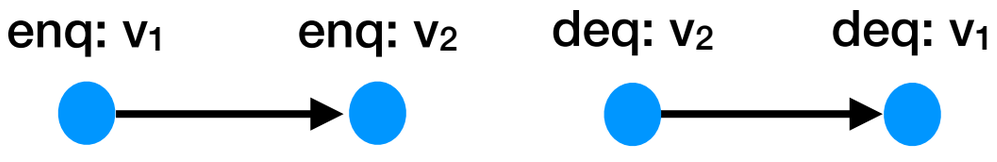
Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

"Value v dequeued without being enqueued"

"Value v dequeued before being enqueued"

"Value v dequeued twice"

"Values dequeued in the wrong order"

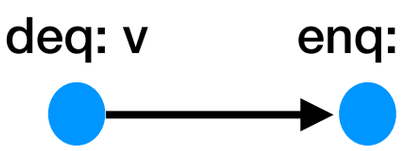


Concurrent Queues [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

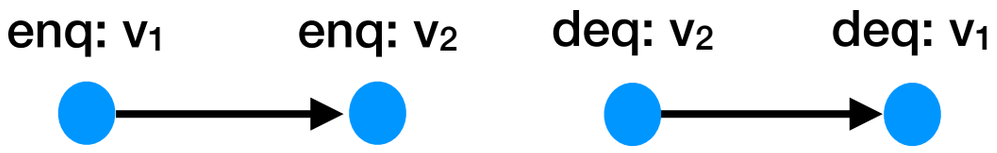
"Value v dequeued without being enqueued"

"Value v dequeued before being enqueued"

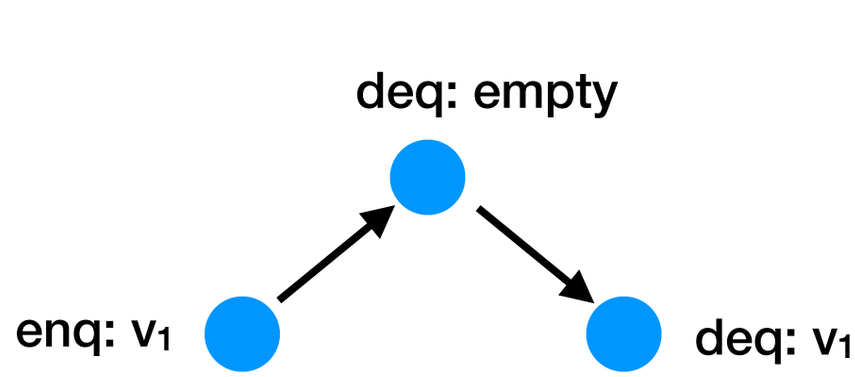


"Value v dequeued twice"

"Values dequeued in the wrong order"



"Dequeue wrongfully returns empty"



Concurrent Queues [ICALP'15]

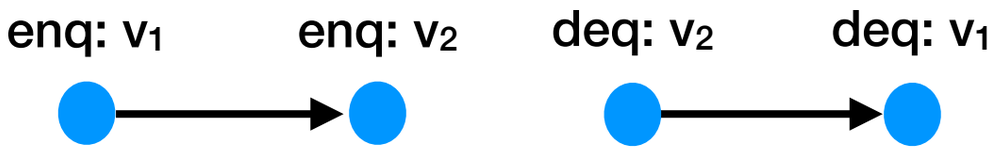
Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

"Value v dequeued without being enqueued"

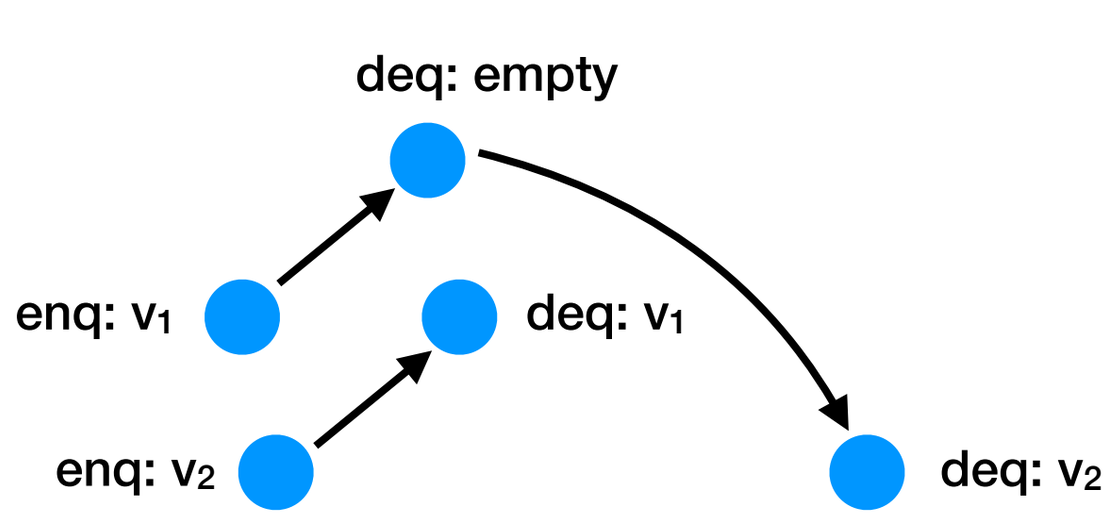
"Value v dequeued before being enqueued"

"Value v dequeued twice"

"Values dequeued in the wrong order"



"Dequeue wrongfully returns empty"



Concurrent Queues [ICALP'15]

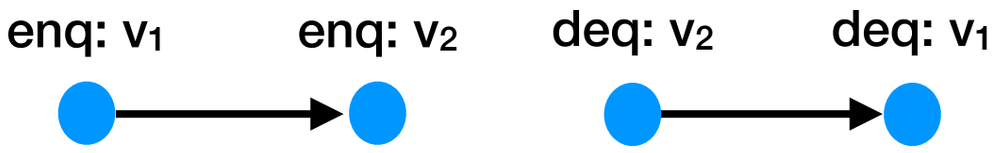
Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

"Value v dequeued without being enqueued"

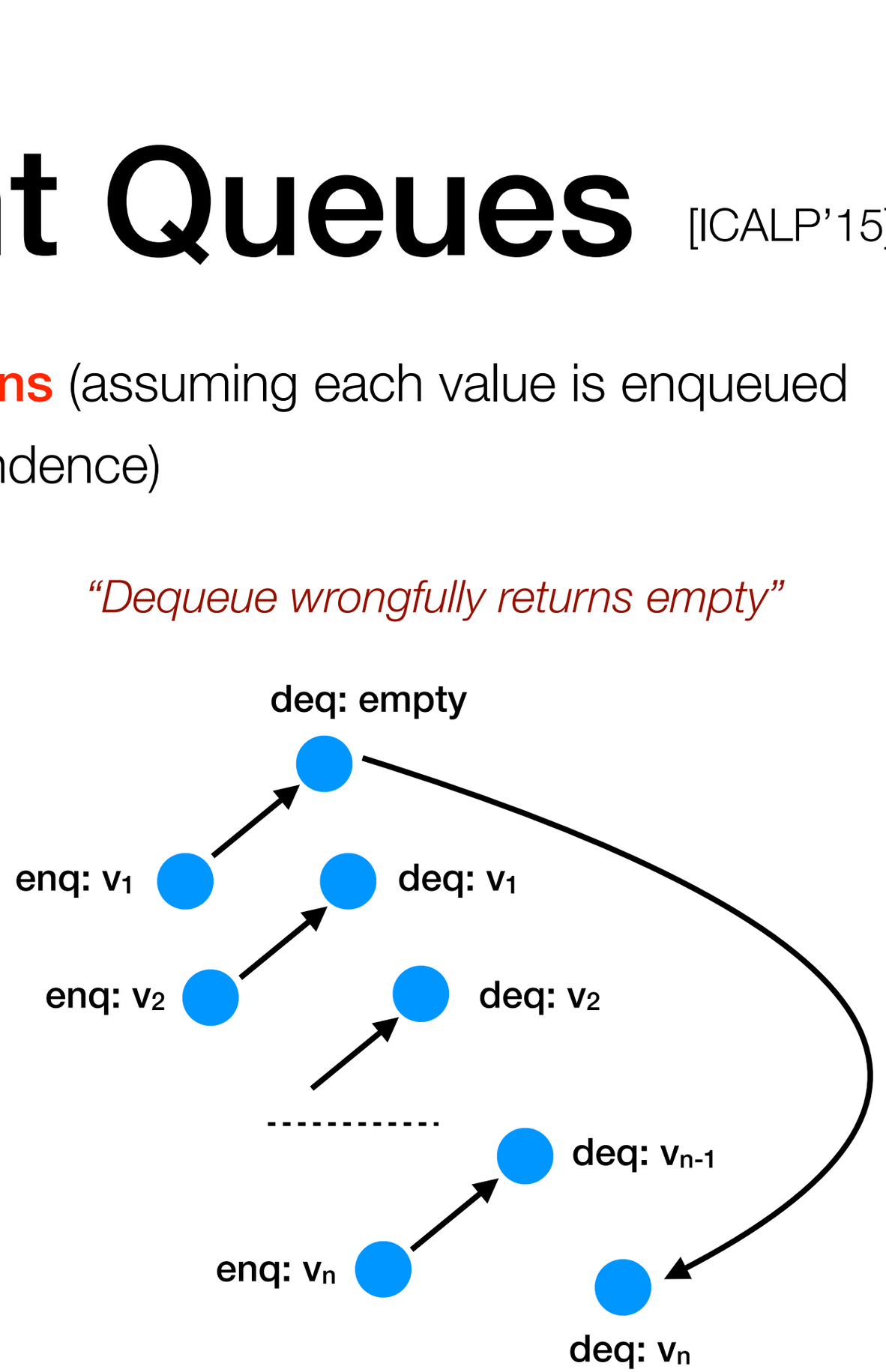
"Value v dequeued before being enqueued"

"Value v dequeued twice"

"Values dequeued in the wrong order"



"Dequeue wrongfully returns empty"



Concurrent Stacks [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once, which is sound under data independence)

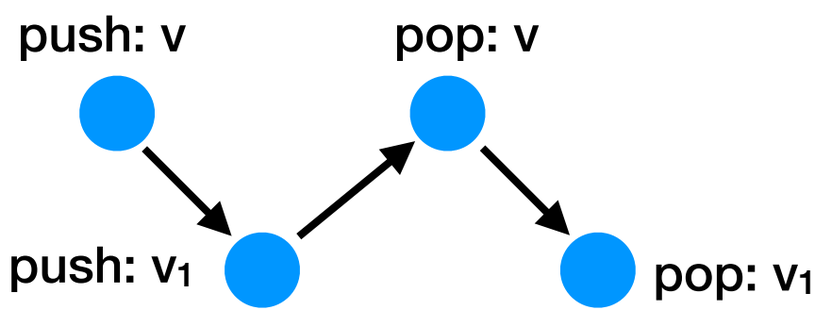
"Pop doesn't return the top of the stack"

"Value v popped without being pushed"

"Value v popped before being pushed"

"Value v popped twice"

"Pop wrongfully returns empty"



Concurrent Stacks [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once, which is sound under data independence)

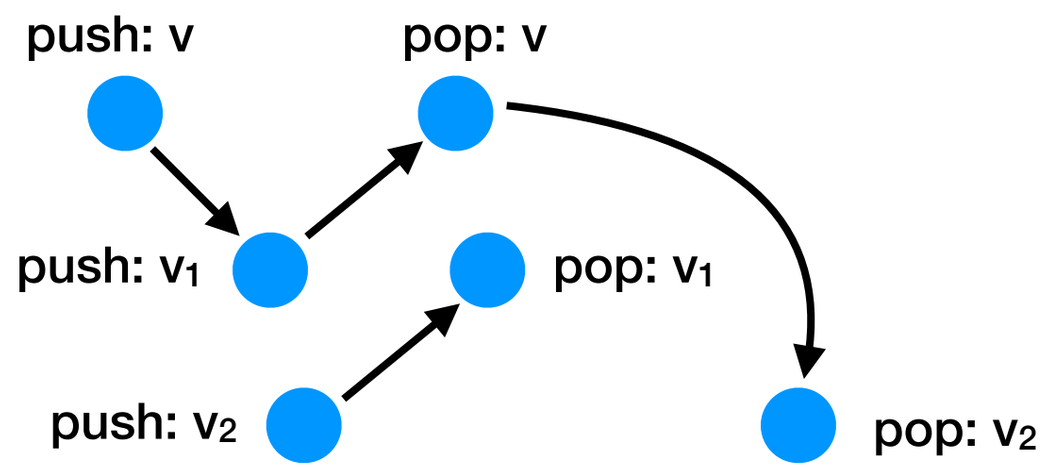
"Pop doesn't return the top of the stack"

"Value v popped without being pushed"

"Value v popped before being pushed"

"Value v popped twice"

"Pop wrongfully returns empty"



Concurrent Stacks [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once, which is sound under data independence)

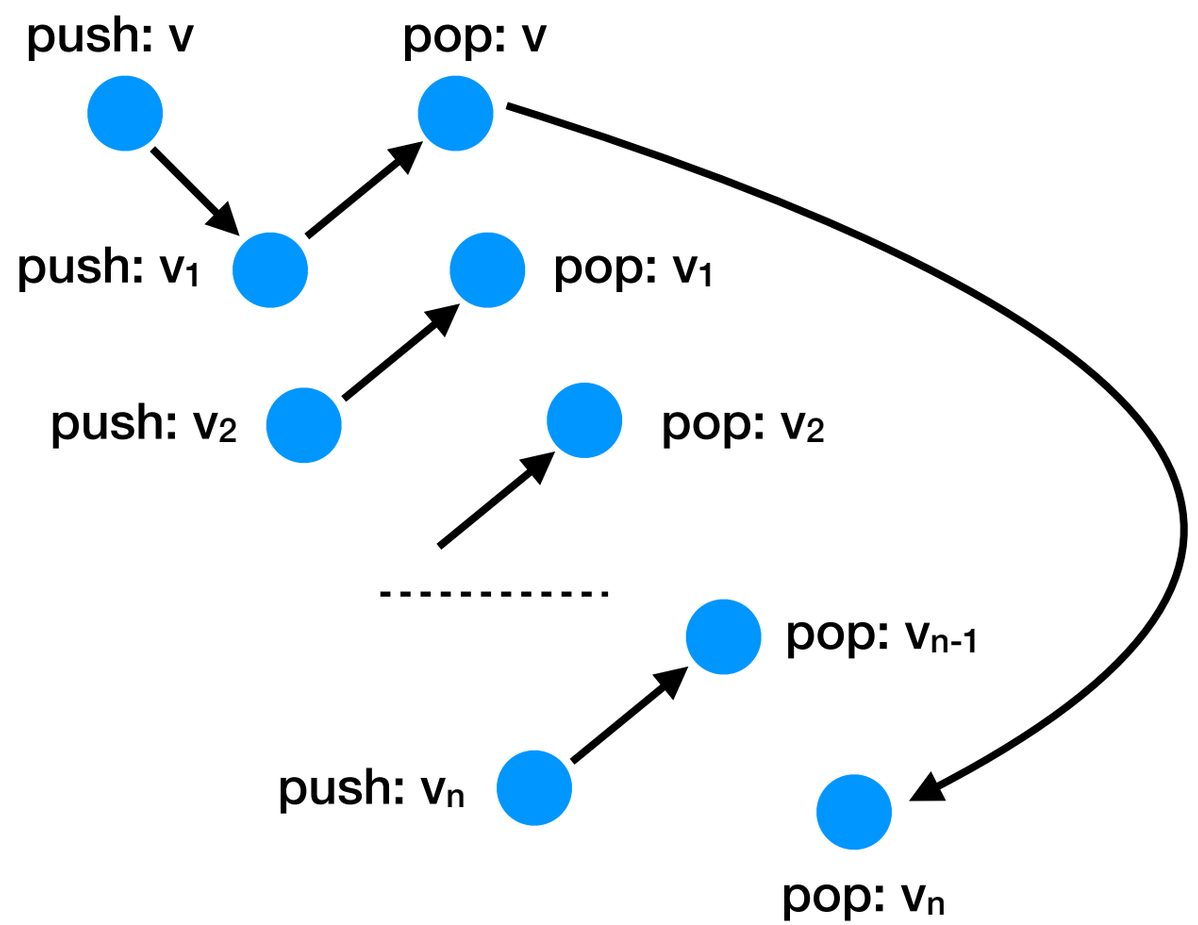
"Value v popped without being pushed"

"Value v popped before being pushed"

"Value v popped twice"

"Pop wrongfully returns empty"

"Pop doesn't return the top of the stack"



Checking Lin. using “bad patterns”

- Reduce linearizability checking to reachability (EXPSPACE-complete):
  - Define (sequential) data-structure S using inductive rules
  - S is data independent and closed under projection
  - Characterize sequential executions of S using bad patterns
  - Characterize concurrent executions, linearizable w.r.t. S using bad patterns (one per rule)
  - Define a regular automaton A_i for each bad pattern
  - Reduce“L is linearizable w.r.t. S” to: ⍰for all i, $L \cap A_i = \varnothing$

Inductive definition of the Register

$$R_{wr}: \ u \in R \implies Write_x \cdot (Read_x)^* \cdot u \in R$$

- including the empty sequence

Inductive definition of the Queue

Two rules to build the **sequences** belonging to the Queue such as

$$Enq_4\, Enq_3\, Deq_4\, Deq_3\, \mathbf{EMP}\, Enq_2\, Enq_1\, Deq_2\, Deq_1 \in Q$$

$$R_{Enq}: \quad u \in Q \wedge u \in Enq^* \Rightarrow u \cdot \mathbf{Enq_x} \in Q$$

$$R_{EnqDeq}: \quad u \cdot v \in Q \wedge u \in Enq^* \Rightarrow \mathbf{Enq_x} \cdot u \cdot \mathbf{Deq_x} \cdot v \in Q$$

$$R_{EMP}: \quad u \cdot v \in Q \wedge \text{no unmatched } Enq \text{ in } u \Rightarrow u \cdot \mathbf{EMP} \cdot v \in Q$$

Derivation:

$$\epsilon \in Q$$

$$\rightarrow \quad \mathbf{Enq_1\, Deq_1} \in Q$$

$$\rightarrow \quad \mathbf{Enq_2}\, Enq_1\, \mathbf{Deq_2}\, Deq_1 \in Q$$

$$\rightarrow \quad \mathbf{Enq_3 Deq_3}\, Enq_2\, Enq_1\, Deq_2\, Deq_1 \in Q$$

$$\rightarrow \quad \mathbf{Enq_4}\, Enq_3\, \mathbf{Deq_4}\, Deq_3\, Enq_2\, Enq_1\, Deq_2\, Deq_1 \in Q$$

$$\rightarrow \quad Enq_4\, Enq_3\, Deq_4\, Deq_3\, \mathbf{EMP}\, Enq_2\, Enq_1\, Deq_2\, Deq_1 \in Q$$

Inductive definition of the Stack

$$R_{PushPop}: \quad u \cdot v \in S \wedge \text{no unmatched } \mathit{Push} \text{ in } u, v \Rightarrow \mathbf{Push_x} \cdot u \cdot \mathbf{Pop_x} \cdot v \in S$$

$$R_{Push}: \quad u \cdot v \in S \wedge \text{no unmatched } \mathit{Push} \text{ in } u \Rightarrow u \cdot \mathbf{Push_x} \cdot v \in S$$

$$R_{EMP}: \quad u \cdot v \in S \wedge \text{no unmatched } \mathit{Push} \text{ in } u \Rightarrow u \cdot \mathbf{EMP} \cdot v \in S$$

Derivation for $\mathit{Push}_1\, \mathit{Push}_2\, \mathit{Pop}_2\, \mathit{Pop}_1\, \mathbf{EMP}\, \mathit{Push}_3\, \mathit{Pop}_3 \in S$

$$\begin{aligned}
& \epsilon \in S \\
\rightarrow \quad & \mathbf{Push_3\, Pop_3} \in S \\
\rightarrow \quad & \mathbf{Push_2\, Pop_2}\, \mathit{Push}_3\, \mathit{Pop}_3 \in S \\
\rightarrow \quad & \mathbf{Push_1}\, \mathit{Push}_2\, \mathit{Pop}_2\, \mathbf{Pop_1}\, \mathit{Push}_3\, \mathit{Pop}_3 \in S \\
\rightarrow \quad & \mathit{Push}_1\, \mathit{Push}_2\, \mathit{Pop}_2\, \mathit{Pop}_1\, \mathbf{EMP}\, \mathit{Push}_3\, \mathit{Pop}_3 \in S
\end{aligned}$$

Data Independence

- Input methods = methods taking an argument
- A sequential execution u is called *differentiated* if for all input methods m and every x, u contains at most one invocation m(x)
- $S_{\neq}$ is the set of differentiated executions in S

A *renaming* r is a function from $\mathbb{D}$ to $\mathbb{D}$. Given a sequential execution (resp., execution or history) u, we denote by $r(u)$ the sequential execution (resp., execution or history) obtained from u by replacing every data value x by $r(x)$.

Definition 6. *The set of sequential executions (resp., executions or histories)* S *is* data independent *if:*

- *for all* $u \in S$*, there exists* $u' \in S_{\neq}$*, and a renaming* r *such that* $u = r(u')$*,*
- *for all* $u \in S$ *and for all renaming* r*,* $r(u) \in S$*.*

Theorem: A data-independent implementation I is linearizable w.r.t. a data-independent specification S iff $I_{\neq}$ is linearizable w.r.t. $S_{\neq}$

Closure under projection

Projection: Subsequence consistent with the values

If

$$Enq_4\, Enq_3\, Deq_4\, Deq_3\, Enq_2\, Enq_1\, Deq_2\, Deq_1 \in Q$$

Then

$$Enq_4\, Deq_4\, Enq_2\, Enq_1\, Deq_2\, Deq_1 \in Q$$

Lemma

Any **data structure** *defined in our framework is* **closed under projection**

Proof.

The **predicates** used ($u \in Enq^*$ and "no unmatched Enq in u") are closed under projection $\square$

Characterization of sequential executions

We assume that the rules defining a data-structure are *well-formed*, that is:

- for all $u \in [\![S]\!]$, there exists a unique rule, denoted by $\texttt{last}(u)$, that can be used as the last step to derive u, i.e., for every sequence of rules $R_{i_1}, \ldots, R_{i_n}$ leading to u, $R_{i_n} = \texttt{last}(u)$. For $u \notin [\![S]\!]$, $\texttt{last}(u)$ is also defined but can be arbitrary, as there is no derivation for u.
- if $\texttt{last}(u) = R_i$, then for every permutation $u' \in [\![S]\!]$ of a projection of u, $\texttt{last}(u') = R_j$ with $j \leq i$. If u' is a permutation of u, then $\texttt{last}(u') = R_i$.

Example 6. *For* Queue*, we define* `last` *for a sequential execution* u *as follows:*

- *if* u *contains a DeqEmpty operation,* $\texttt{last}(u) = R_{DeqEmpty}$*,*
- *else if* u *contains a Deq operation,* $\texttt{last}(u) = R_{EnqDeq}$*,*
- *else if* u *contains only Enq's,* $\texttt{last}(u) = R_{Enq}$*,*
- *else (if* u *is empty),* $\texttt{last}(u) = R_0$*.*

Since the conditions we use to define `last` *are closed under permutations, we get that for any permutation* u_2 *of* u*,* $\texttt{last}(u) = \texttt{last}(u_2)$*, and* `last` *can be extended to histories. Therefore, the rules* $R_0, R_{EnqDeq}, R_{DeqEmpty}$ *are well-formed.*

Characterization of sequential executions

- MS(R) = the set of sequences “matching” a rule R

Lemma 3. *Let $S = R_1, \ldots, R_n$ be a data-structure and u be a differentiated sequential execution. Then,*

$$u \in S \iff \mathsf{proj}(u) \subseteq \bigcup_{i \in \{1,\ldots,n\}} \mathsf{MS}(R_i)$$

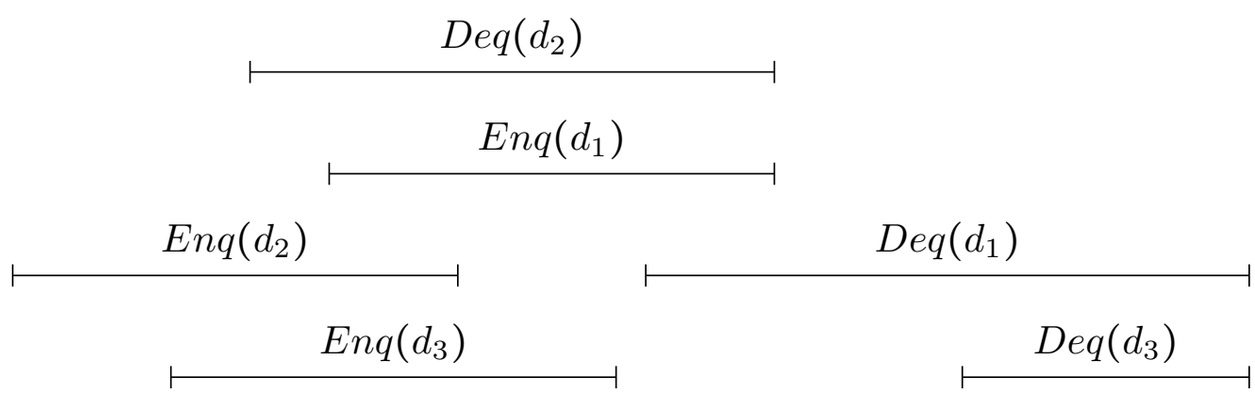
Lemma (Characterization of Queue Sequential Executions)

$w \in Q$ iff **every projection** *w' of w is either of the form*
$\mathbf{Enq_x} \cdot u \cdot \mathbf{Deq_x} \cdot v$ *(with $u \in Enq^*$) or*
$u \cdot$ **EMP** *$\cdot v$ (with no unmatched Enq in u)*

Characterization of concurrent executions

Definition 7. *A data-structure $S = R_1, \ldots, R_n$ is said to be* step-by-step linearizable *if for any differentiated execution e, any $i \in \{1, \ldots, n\}$ and $x \in \mathbb{D}$, if e is linearizable with respect to $\mathsf{MS}(R_i)$ with witness x, we have:*

$$e \setminus x \sqsubseteq [\![R_1, \ldots, R_i]\!] \implies e \sqsubseteq [\![R_1, \ldots, R_i]\!]$$



- the history linearizable MS(R_{EnqDeq}) with witness d_1
  - Enq(d_1) is minimal among all operations and Deq(d_1) minimal among all dequeues
- Excluding the operations on d1, the history is linearizable w.r.t. [R_{Enq},R_{EnqDeq}], i.e., Enq(d_2) Enq(d_3) Deq(d_2) Deq(d_3)
- The notion of step-by-step linearizable ensures that the history is linearizable w.r.t. Queue

Step-by-Step Lin. of Register

Lemma 9. Register *is step-by-step linearizable.*

Proof. Let h be a differentiated history, and u a sequential execution such that $h \sqsubseteq u$ and such that u matches the rule R_{WR} with witness x. Let a and $b_1, \ldots, b_s$ be respectively the $Write$ and $Read$'s operations of h corresponding to the witness.

Let $h' = h \setminus x$ and assume $h' \sqsubseteq [\![R_0, R_{WR}]\!]$. Let $u' \in [\![R_0, R_{WR}]\!]$ such that $h' \sqsubseteq u'$. Let $u_2 = a \cdot b_1 \cdot b_2 \cdots b_s \cdot u'$. By using rule R_{WR} on u', we have $u_2 \in [\![R_0, R_{WR}]\!]$. Moreover, we prove that $h \sqsubseteq u_2$ by contradiction. Assume that the total order imposed by u_2 doesn't respect the happens-before relation of h. All three cases are not possible:

- the violation is between two u' operations, contradicting $h' \sqsubseteq u'$,
- the violation is between a and another operation, i.e. there is an operation o which happens before a in h, contradicting $h \sqsubseteq u$,
- the violation is between some b_i and a u' operation, i.e. there is an operation o which happens before b_i in h, contradicting $h \sqsubseteq u$.

Thus, we have $h \sqsubseteq u_2$ and $h \sqsubseteq [\![R_0, R_{WR}]\!]$, which ends the proof. □

Characterization of concurrent executions

Lemma 4. *Let S be a data-structure with rules $R_1, \ldots, R_n$. Let e be a differentiated execution. If S is step-by-step linearizable, we have (for any j):*

$$e \sqsubseteq [\![R_1, \ldots, R_j]\!] \iff \mathsf{proj}(e) \sqsubseteq \bigcup_{i \leq j} \mathsf{MS}(R_i)$$

Proof ($\Leftarrow$) By induction on the size of e. We know $e \in \mathsf{proj}(e)$ so it can be linearized with respect to a sequential execution u matching some rule R_k $(k \leq j)$ with some witness x. Let $e' = e \setminus x$.

Since S is well-formed, we know that no projection of e can be linearized to a matching set $\mathsf{MS}(R_i)$ with $i > k$, and in particular no projection of e'. Thus, we deduce that $\mathsf{proj}(e') \sqsubseteq \bigcup_{i \leq k} \mathsf{MS}(R_i)$, and conclude by induction that $e' \sqsubseteq [\![R_1, \ldots, R_k]\!]$.

We finally use the fact that S is step-by-step linearizable to deduce that $e \sqsubseteq [\![R_1, \ldots, R_k]\!]$ and $e \sqsubseteq [\![R_1, \ldots, R_j]\!]$ because $k \leq j$. □

Lemma

E is linearizable to Q iff **every projection** *E' of E is linearizable to the form $\mathbf{Enq_x} \cdot u \cdot \mathbf{Deq_x} \cdot v$ (with $u \in Enq^*$) or to the form $u \cdot \mathbf{EMP} \cdot v$ (with no unmatched Enq in u)*

Characterization of concurrent executions

Lemma 5. *Let S be a data-structure with rules $R_1, \ldots, R_n$. Let e be a differentiated execution. If S is step-by-step linearizable, we have:*

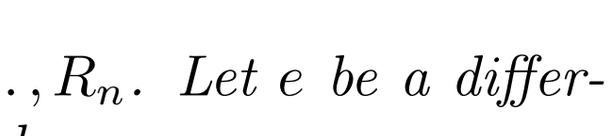
$$e \sqsubseteq S \iff \forall e' \in \mathsf{proj}(e).\ e' \sqsubseteq \mathsf{MS}(R) \text{ where } R = \texttt{last}(e')$$

$$e \not\sqsubseteq S \iff \exists e' \in \mathsf{proj}(e).\ e' \not\sqsubseteq \mathsf{MS}(R) \text{ (where } R = \texttt{last}(e'))$$

E is **non-linearizable** *wrt Queue iff it has a projection E′ of the form* **bad pattern 1**, *or* **bad pattern 2**.

Bad Pattern 1 (rule R_{EnqDeq}):

$Enq_1 < Enq_2$
$Deq_2 < Deq_1$



or Deq_1 before Enq_1

Characterization of concurrent executions

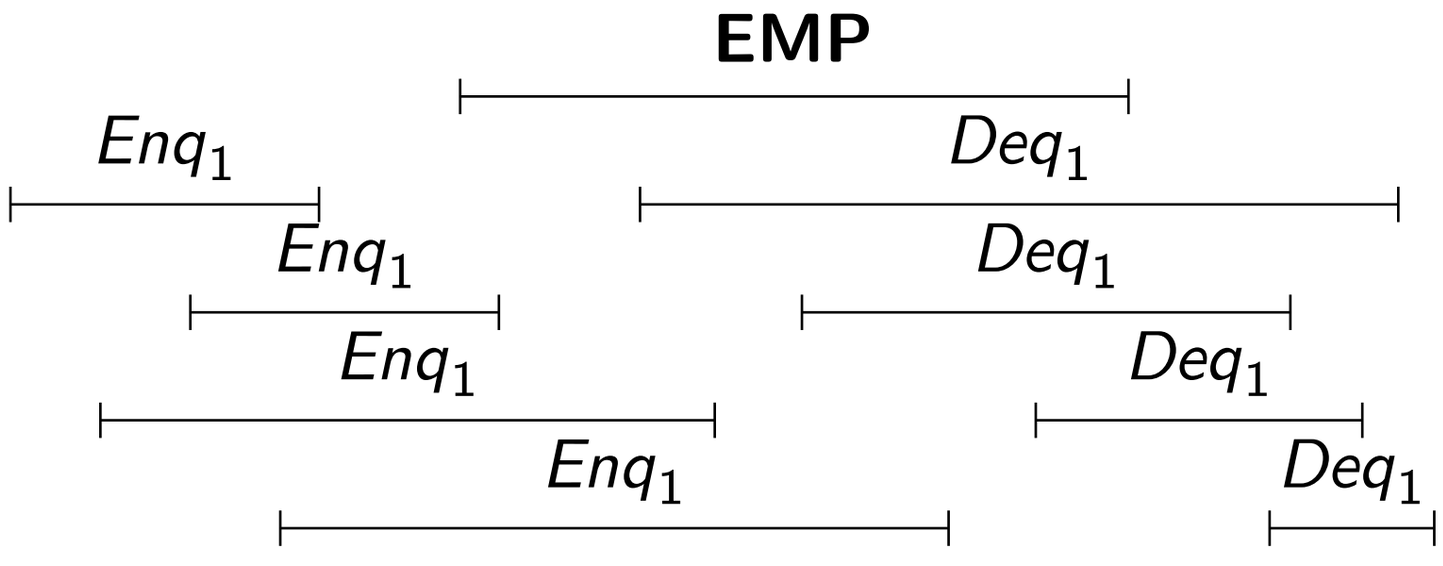
Lemma 5. *Let S be a data-structure with rules $R_1, \ldots, R_n$. Let e be a differentiated execution. If S is step-by-step linearizable, we have:*

$$e \sqsubseteq S \iff \forall e' \in \mathsf{proj}(e).\ e' \sqsubseteq \mathsf{MS}(R) \text{ where } R = \texttt{last}(e')$$

$$e \not\sqsubseteq S \iff \exists e' \in \mathsf{proj}(e).\ e' \not\sqsubseteq \mathsf{MS}(R) \text{ (where } R = \texttt{last}(e'))$$

E is **non-linearizable** *wrt Queue iff it has a projection E' of the form* **bad pattern 1**, *or* **bad pattern 2**.

Bad Pattern 2:
(rule R_{EMP})



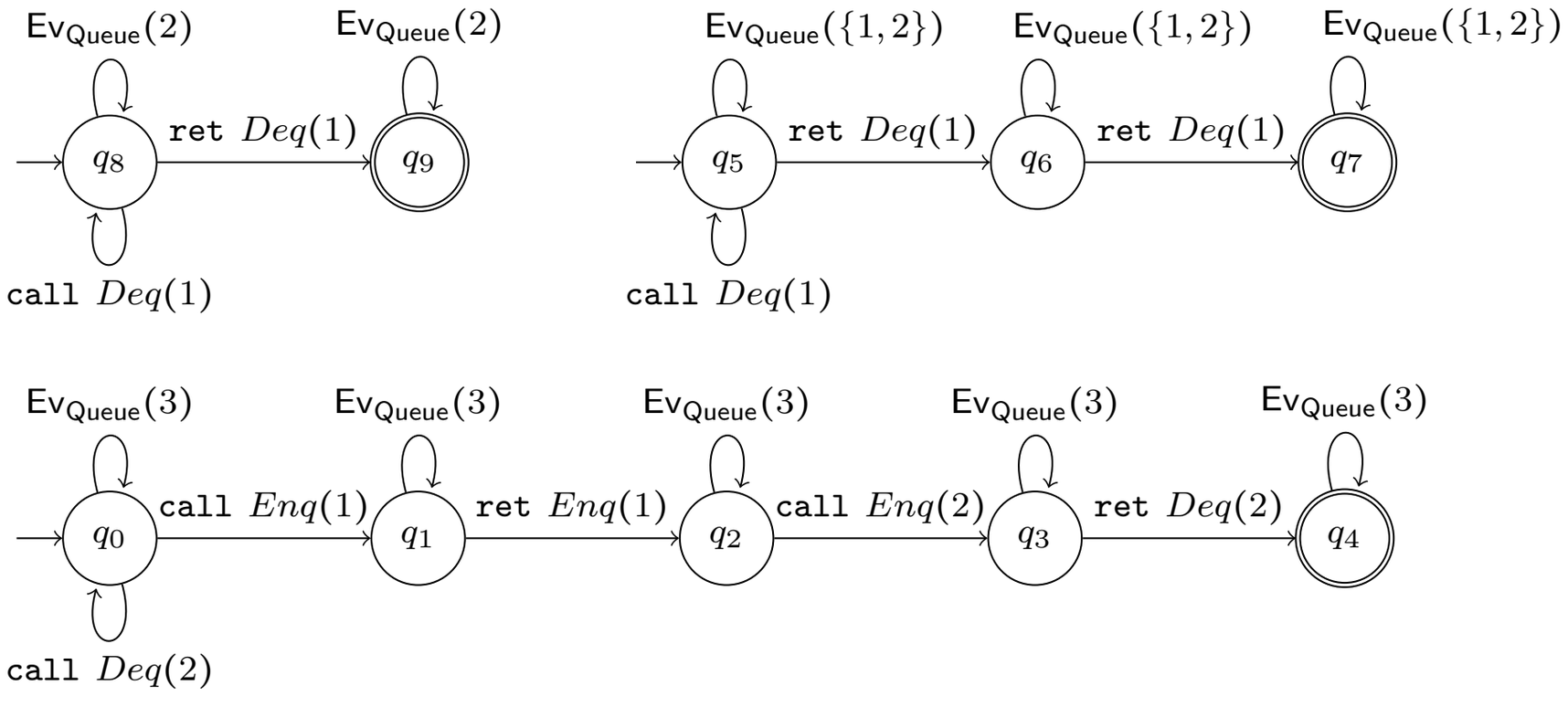
Characterization of concurrent executions

- define for each R, a finite state automaton A which recognizes (a subset of) the executions e which have a projection not linearizable w.r.t. MS(R)

Definition 8. *A rule R is said to be* co-regular *if we can build an automaton $\mathcal{A}$ such that, for any data-independent implementation $\mathcal{I}$, we have:*

$$\mathcal{I} \cap \mathcal{A} \neq \varnothing \iff \exists e \in \mathcal{I}_{\neq}, e' \in \mathsf{proj}(e).\ \mathtt{last}(e') = R \wedge e' \notin \mathsf{MS}(R)$$

R_{EnqDeq}



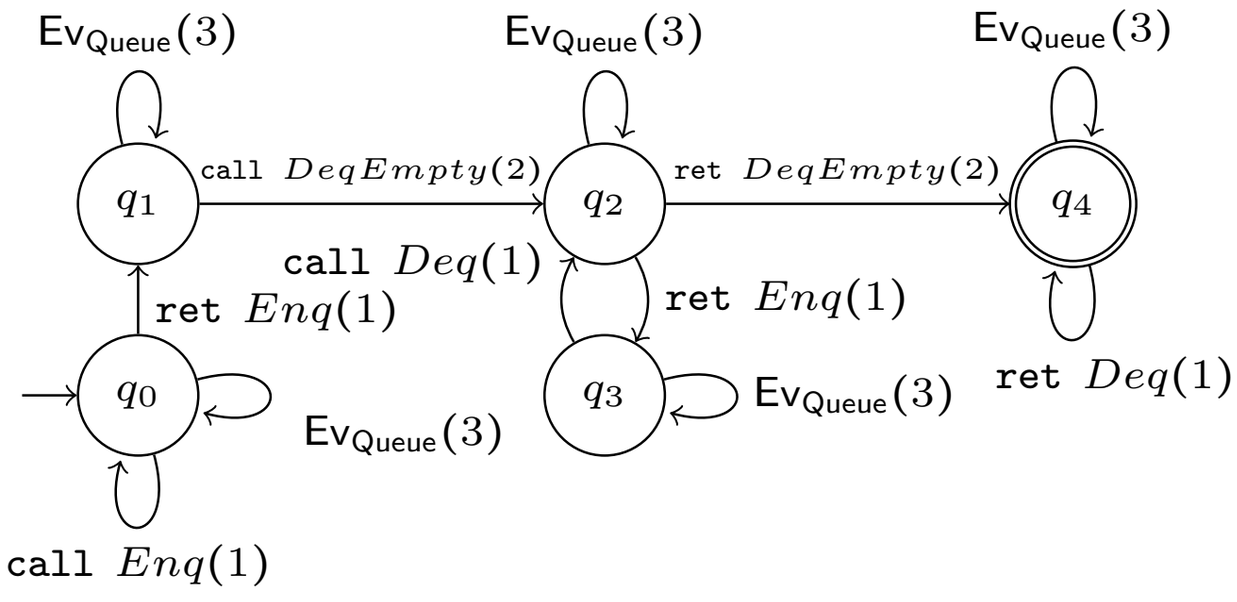
Characterization of concurrent executions

- define for each R, a finite state automaton A which recognizes (a subset of) the executions e which have a projection not linearizable w.r.t. MS(R)

Definition 8. *A rule R is said to be* co-regular *if we can build an automaton $\mathcal{A}$ such that, for any data-independent implementation $\mathcal{I}$, we have:*

$$\mathcal{I} \cap \mathcal{A} \neq \varnothing \iff \exists e \in \mathcal{I}_{\neq}, e' \in \mathsf{proj}(e).\ \mathtt{last}(e') = R \wedge e' \notin \mathsf{MS}(R)$$

REMP



we assume that all actions call Enq(1) occur at the beginning

Exercices (2)

We consider a sequential specification defined by the language $S = (\texttt{a()})^*(\texttt{b()})^*$ where all the invocations of `a()` occur before invocations of `b()`.

1. Describe a reduction of checking linearizability w.r.t. the specification S to a reachability problem. More precisely, describe a labeled transition system (monitor) that accepts exactly all the histories of a given implementation (sequences of call and return actions) that are *not* linearizable w.r.t. S. The synchronized product between a transition system representing an implementation and this monitor (where the synchronization actions are call and returns) reaches an accepting state of the monitor iff the implementation is not linearizable.

Exercices (3)

- What is the complexity of checking linearizability of a *differentiated* history *of a* concurrent queue?

Exercices (3)

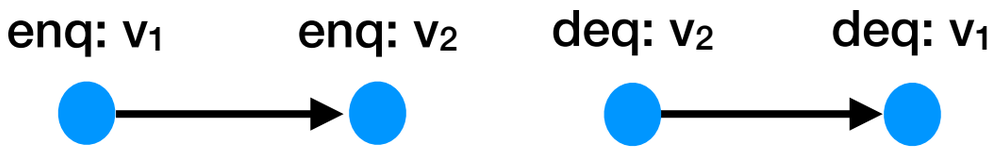
- What is the complexity of checking linearizability of a *differentiated* history *of a* concurrent queue?

"Value v dequeued without being enqueued"

"Value v dequeued before being enqueued"

"Value v dequeued twice"

"Values dequeued in the wrong order"



"Dequeue wrongfully returns empty"