ReLinChe: Automatically Checking Linearizability under Relaxed Memory Consistency

Authors: P. Golovin, M. Kokologiannakis, V. Vafeiadis

Concurrent programming is challenging

Concurrent programming is challenging

- Synchronization

Concurrent programming is challenging

- Synchronization
- Memory reclamation

Concurrent programming is challenging

- Synchronization
- Memory reclamation
- Performance

Concurrent Libraries

Concurrent Libraries

libcds Public Fork 359 Starred 2.6k

Concurrent Libraries

- Concurrent Queue, Stack, Set etc

Concurrent Libraries

- Concurrent Queue, Stack, Set etc

When is such a library correct?

Standard notion: Linearizability

```
ConcurrentQueue q

q.enq(1)   ||   q.deq()
q.enq(2)   ||
```

what values deq() can return ?

Standard notion: Linearizability

```
ConcurrentQueue q
q.enq(1)   ||   q.deq()
q.enq(2)   ||
```

what values deq() can return ?

it can return ⊥ or 1, but not 2 or 42

Standard notion: Linearizability
= atomicity + respecting runtime order

```
ConcurrentQueue q

q.enq(1)   ||   q.deq()
q.enq(2)   ||
```

what values deq() can return ?

it can return ⊥ or 1, but not 2 or 42

Standard notion: Linearizability
= atomicity + respecting runtime order

```
ConcurrentQueue q

q.enq(1)  ||  q.deq()
q.enq(2)  ||
```

what values deq() can return ?

it can return ⊥ or 1, but not 2 or 42

deq():⊥ →

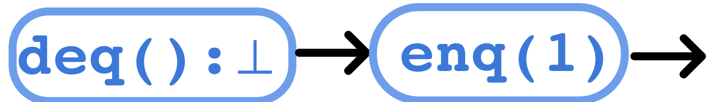
Standard notion: Linearizability
= atomicity + respecting runtime order

```
ConcurrentQueue q

q.enq(1)   ||   q.deq()
q.enq(2)   ||
```

what values deq() can return ?

it can return ⊥ or 1, but not 2 or 42



Standard notion: Linearizability
= atomicity + respecting runtime order

```
ConcurrentQueue q
q.enq(1)  ||  q.deq()
q.enq(2)  ||
```

what values deq() can return ?

it can return ⊥ or 1, but not 2 or 42

Standard notion: Linearizability
= atomicity + respecting runtime order

```
ConcurrentQueue q

q.enq(1)   ||   q.deq()
q.enq(2)   ||
```

what values deq() can return ?

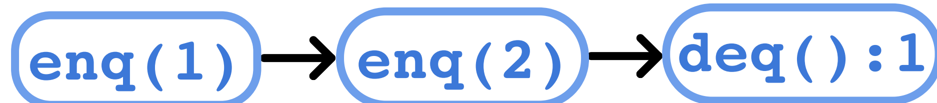
it can return ⊥ or 1, but not 2 or 42

Standard notion: Linearizability
= atomicity + respecting runtime order

```
ConcurrentQueue q
q.enq(1)    ||   q.deq()
q.enq(2)    ||
```

what values deq() can return ?

it can return ⊥ or 1, but not 2 or 42



Linearizability is unsuitable for weak memory

```
ConcurrentQueue q
ConcurrentStack s
```

(SB)

```
s.push(1)              ||      q.enq(2)
q.deq() // ⊥           ||      s.pop() // ⊥
```

Linearizability is unsuitable for weak memory

```
ConcurrentQueue q
ConcurrentStack s
```

(SB)

```
s.push(1)          ||    q.enq(2)
q.deq() // ⊥       ||    s.pop() // ⊥
```

- There is no total order over operations for the outcome

Linearizability is unsuitable for weak memory

```
ConcurrentQueue q
ConcurrentStack s
```

(SB)

```
s.push(1)           ||    q.enq(2)
q.deq() // ⊥        ||    s.pop() // ⊥
```

- There is no total order over operations for the outcome
- Yet, this outcome is intended

Linearizability is unsuitable for weak memory

```
ConcurrentQueue q
ConcurrentStack s
```

(SB)

```
s.push(1)          ||    q.enq(2)
q.deq() // ⊥       ||    s.pop() // ⊥
```

- There is no total order over operations for the outcome
- Yet, this outcome is intended
- But still, we have the total order per object (atomicity)

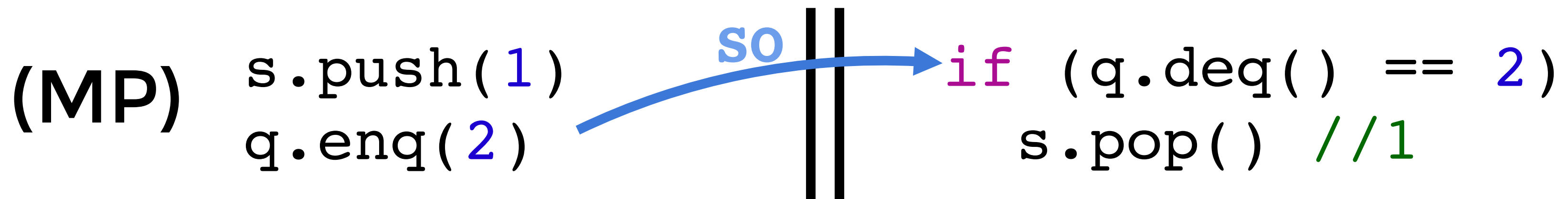
We also need to specify synchronization!

```
ConcurrentQueue q
ConcurrentStack s
```

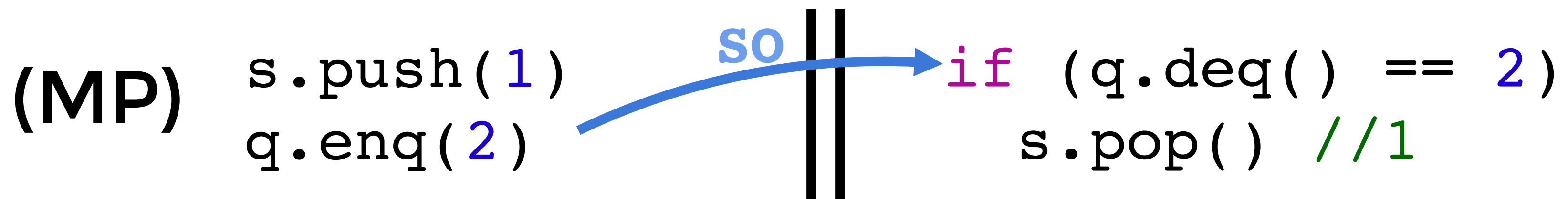
(MP)

```
s.push(1)          ||    if (q.deq() == 2)
q.enq(2)           ||      s.pop() //1
```

We also need to specify synchronization!



We also need to specify synchronization!



Relaxed Linearizability (per data structure):

atomicity + custom synchronization

Our contribution

ReLinChe (Relaxed Linearizability Checker):

Our contribution

ReLinChe (Relaxed Linearizability Checker):

First tool for verifying **relaxed linearizability** of concurrent libraries automatically

Our contribution

ReLinChe (Relaxed Linearizability Checker):

First tool for verifying **relaxed linearizability** of concurrent libraries automatically

- Model checking

Our contribution

ReLinChe (Relaxed Linearizability Checker):

First tool for verifying **relaxed linearizability** of concurrent libraries automatically

- Model checking
- Automatically explore tests of bounded size

Our contribution

ReLinChe (Relaxed Linearizability Checker):

First tool for verifying **relaxed linearizability** of concurrent libraries automatically

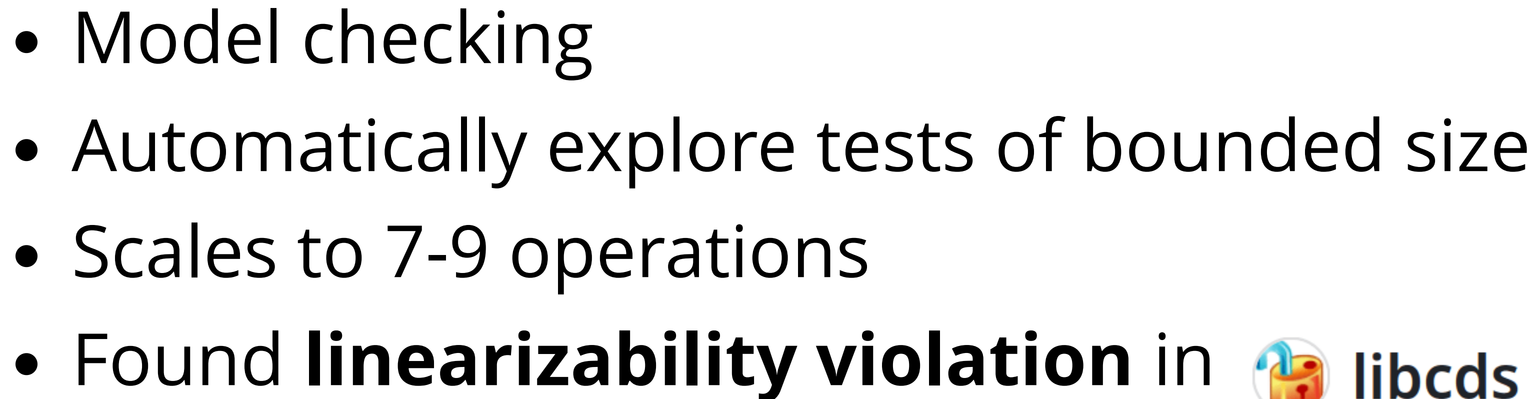
- Model checking
- Automatically explore tests of bounded size
- Scales to 7-9 operations

Our contribution

ReLinChe (Relaxed Linearizability Checker):

First tool for verifying **relaxed linearizability** of concurrent libraries automatically

- Model checking
- Automatically explore tests of bounded size
- Scales to 7-9 operations
- Found **linearizability violation** in



Two main questions:

- How to write a relaxed linearizability spec?
- How to verify a library against the spec?

Two main questions:

- How to write a relaxed linearizability spec?
- How to verify a library against the spec?

How to write such a spec?

- Sequential implementation as a core

How to write such a spec?

```
void enq(v) {

  nodes[back]:= v
  back++

}
```

- Sequential implementation as a core

How to write such a spec?

```
void enq(v) {

  nodes[back]:= v
  back++

}
```

```
int deq() {

  if (front == back)
    res := ⊥ // empty
  else
    front++
    res := nodes[front-1]

  return res
}
```

- Sequential implementation as a core

How to write such a spec?

```
void enq(v) {
  lock()
  nodes[back]:= v
  back++
  unlock()
}
```

```
int deq() {
  lock()
  if (front == back)
    res := ⊥ // empty
  else
    front++
    res := nodes[front-1]
  unlock()
  return res
}
```

- Sequential implementation as a core

How to write such a spec?

```
void enq(v) {
  lock()
  nodes[back]:= v
  back++
  unlock()
}
```

```
int deq() {
  lock()
  if (front == back)
    res := ⊥ // empty
  else
    front++
    res := nodes[front-1]
  unlock()
  return res
}
```

- Sequential implementation as a core
- Global lock induces too much synchronization

How to write such a spec?

```
void enq(v) {
  plock()
  nodes[back]:= v
  back++
  punlock()
}
```

```
int deq() {
  plock()
  if (front == back)
    res := ⊥ // empty
  else
    front++
    res := nodes[front-1]
  punlock()
  return res
}
```

- Sequential implementation as a core
- Partial lock (Singh&Lahav'23) which doesn't expose any synchronization, but ensure atomicity

How to write such a spec?

```
void enq(v) {
    plock()
rel nodes[back]:= v
    back++
    punlock()
}
```

so

```
int deq() {
    plock()
    if (front == back)
        res := ⊥ // empty
    else
        front++
acq     res := nodes[front-1]
    punlock()
    return res
}
```

- Sequential implementation as a core
- Partial lock (Singh&Lahav'23) which doesn't expose any synchronization, but ensure atomicity
- C11 atomics declare synchronization

Two main questions

- How to write a relaxed linearizability spec?
- How to verify a library against the spec?

Two main questions

- How to write a relaxed linearizability spec?
- How to verify a library against the spec?
  - for a specific test client?

Two main questions

- How to write a relaxed linearizability spec?
- How to verify a library against the spec?
  - for a specific test client?
  - for all test clients (up to a certain bound)?

Verifying a library for a given client

Phase 1. Analyze the specification:

Phase 2. Check the implementation:

Verifying a library for a given client

Phase 1. Analyze the specification:

- Run model checker on **Test[Spec]**
- Collect all behaviors of the spec

Phase 2. Check the implementation:

Verifying a library for a given client

Phase 1. Analyze the specification:

- Run model checker on **Test[Spec]**
- Collect all behaviors of the spec

Phase 2. Check the implementation:

Test client:

```
q.enq(1) || q.deq()
q.enq(2) ||
```

Verifying a library for a given client

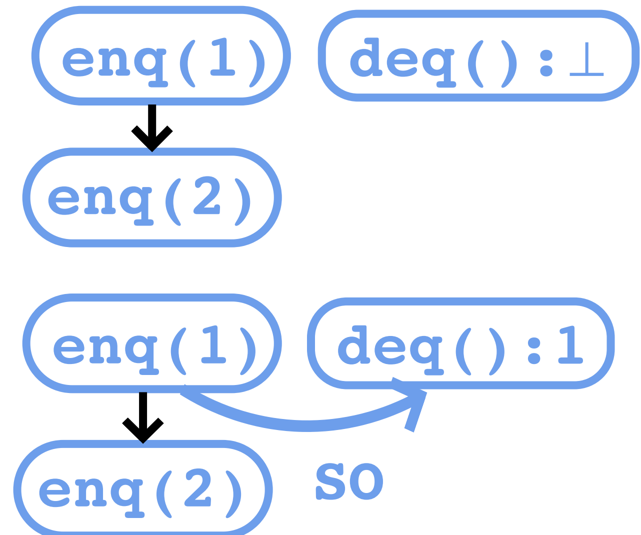
Phase 1. Analyze the specification:

- Run model checker on **Test[Spec]**
- Collect all behaviors of the spec

Phase 2. Check the implementation:

Test client:

```
q.enq(1) || q.deq()
q.enq(2) ||
```



Verifying a library for a given client

Phase 1. Analyze the specification:

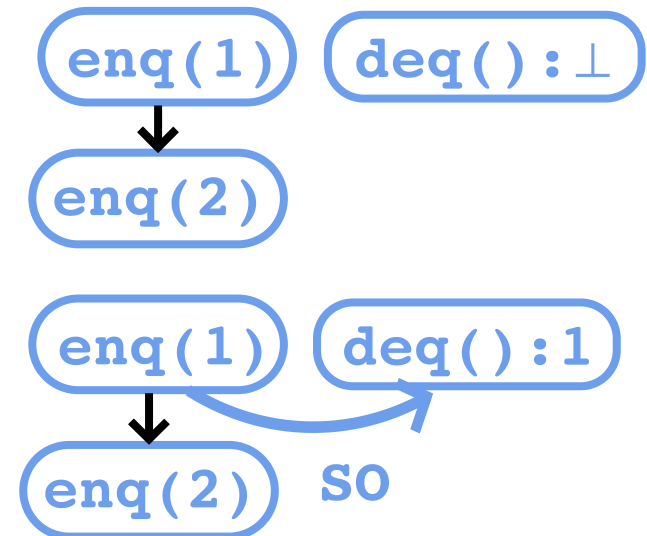
- Run model checker on **Test[Spec]**
- Collect all behaviors of the spec

Phase 2. Check the implementation:

- Run model checker on **Test[Impl]**
- For each G_{impl}, check that exists G_{spec} :

Test client:

```
q.enq(1) || q.deq()
q.enq(2) ||
```



Verifying a library for a given client

Test client:

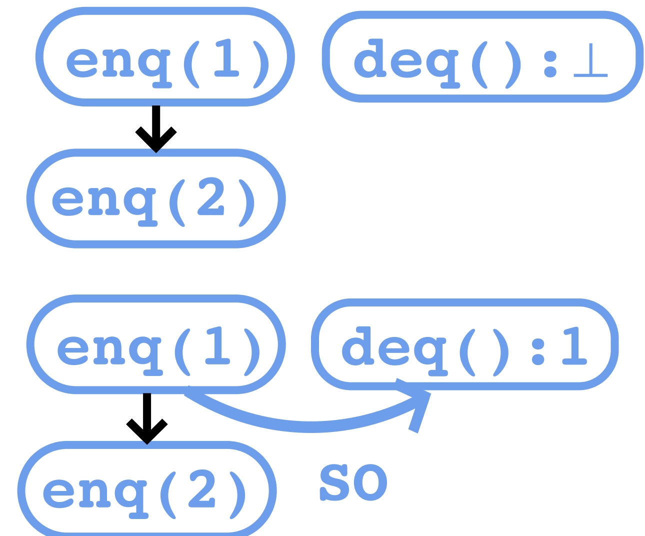
```
q.enq(1) || q.deq()
q.enq(2) ||
```

Phase 1. Analyze the specification:

- Run model checker on **Test[Spec]**
- Collect all behaviors of the spec

Phase 2. Check the implementation:

- Run model checker on **Test[Impl]**
- For each G_{impl}, check that exists G_{spec} :
  - same return values



Verifying a library for a given client

Phase 1. Analyze the specification:

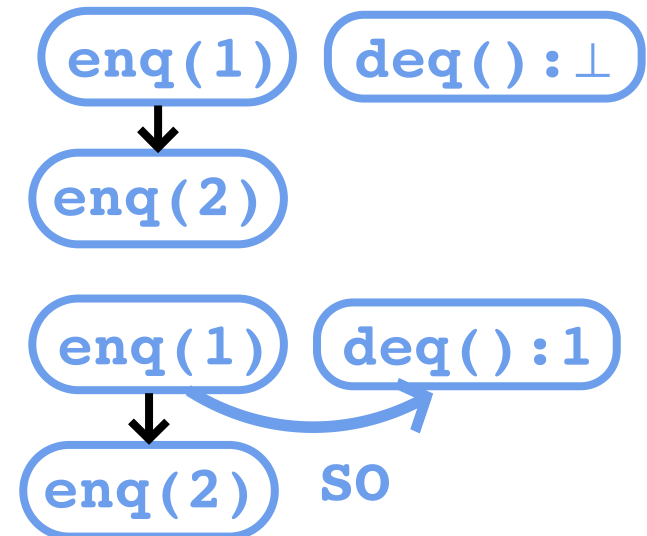
- Run model checker on **Test[Spec]**
- Collect all behaviors of the spec

Phase 2. Check the implementation:

- Run model checker on **Test[Impl]**
- For each G_{impl}, check that exists G_{spec} :
  - same return values
  - less or equal synchronization

Test client:

```
q.enq(1) || q.deq()
q.enq(2) ||
```



Verifying a library for a given client

Phase 1. Analyze the specification:

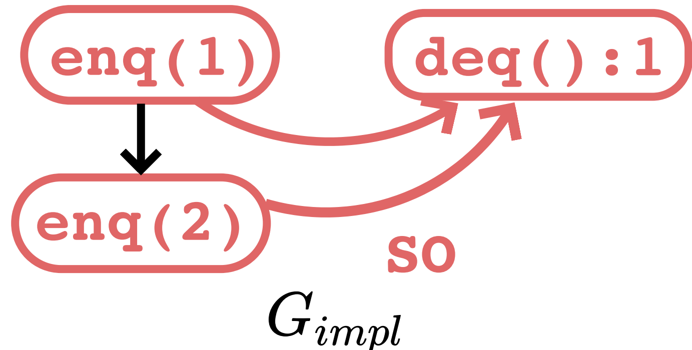
- Run model checker on **Test[Spec]**
- Collect all behaviors of the spec

Phase 2. Check the implementation:

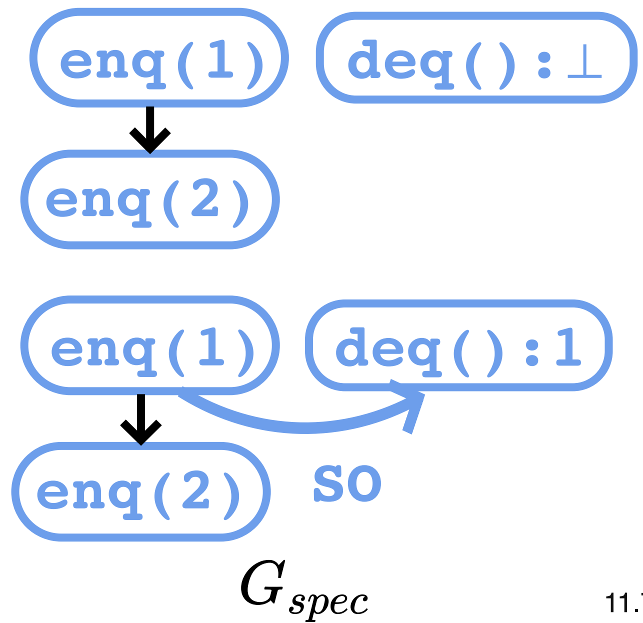
- Run model checker on **Test[Impl]**
- For each G_{impl}, check that exists G_{spec} :
  - same return values
  - less or equal synchronization

Test client:

```
q.enq(1) || q.deq()
q.enq(2) ||
```



$\in$



Two main questions:

- How to write a relaxed linearizability spec?
- How to verify a library against the spec?
  - for a specific test client?
  - for all test clients (up to a certain bound)?

Two main questions:

- How to write a relaxed linearizability spec?
- How to verify a library against the spec?
  - for a specific test client?
  - for all test clients (up to a certain bound)?

Verifying a library for all test clients

Verifying a library for all test clients

```
q.enq(1)    || q.deq()
q.enq(2)    ||
```

Verifying a library for all test clients

```
q.enq(1)      ||  q.enq(2)
q.deq()       ||
```

```
q.enq(1)      ||  q.deq()
q.enq(2)      ||
```

Verifying a library for all test clients

```
q.enq(1) || q.enq(2) || q.deq()
```

```
q.enq(1)   || q.enq(2)
q.deq()    ||
```

```
q.enq(1)   || q.deq()
q.enq(2)   ||
```

... & many more clients

Verifying a library for all test clients

```
q.enq(1) || q.enq(2) || q.deq()
```

```
q.enq(1)   ||  q.enq(2)
q.deq()    ||
```

```
q.enq(1)   ||  q.deq()
q.enq(2)   ||
```

... & many more clients

- **Naive approach doesn't scale**

Verifying a library for all test clients

`q.enq(1) || q.enq(2) || q.deq()` could produce `⊥, 1, 2`

```
q.enq(1)    || q.enq(2)
q.deq()     ||
```

```
q.enq(1)    || q.deq()
q.enq(2)    ||
```

... & many more clients

- Naive approach doesn't scale

Verifying a library for all test clients

`q.enq(1) || q.enq(2) || q.deq()` could produce `⊥, 1, 2`

`q.enq(1)` `q.deq()`	‖	`q.enq(2)`	could produces `1, 2`
`q.enq(1)` `q.enq(2)`	‖	`q.deq()`	

... & many more clients

- Naive approach doesn't scale

Verifying a library for all test clients

`q.enq(1) || q.enq(2) || q.deq()` could produce ⊥, 1, 2

`q.enq(1)` `q.deq()`	`q.enq(2)`	could produces	1, 2
`q.enq(1)` `q.enq(2)`	`q.deq()`	could produces	⊥, 1

... & many more clients

- Naive approach doesn't scale

Verifying a library for all test clients

Most parallel client:

`q.enq(1) || q.enq(2) || q.deq()` could produce `⊥, 1, 2`

`q.enq(1)` `q.deq()`	`q.enq(2)`	could produces	`1, 2`
`q.enq(1)` `q.enq(2)`	`q.deq()`	could produces	`⊥, 1`

... & many more clients

- **Naive approach doesn't scale**

Verifying a library for all test clients

Most parallel client:

`q.enq(1) || q.enq(2) || q.deq()` could produce ⊥, 1, 2

"extension"-clients of MPC:

```
q.enq(1)    ||  q.enq(2)
q.deq()
```

could produces 1, 2

```
q.enq(1)    ||  q.deq()
q.enq(2)
```

could produces ⊥, 1

... & many more clients

- Naive approach doesn't scale

Verifying a library for all test clients

Most parallel client:

`q.enq(1) || q.enq(2) || q.deq()` could produce ⊥, 1, 2

"extension"-clients of MPC:

```
q.enq(1)   ||  q.enq(2)
q.deq()
```

could produces 1, 2

```
q.enq(1)   ||  q.deq()
q.enq(2)
```

could produces ⊥, 1

... & many more clients

- Naive approach doesn't scale
- MPC can explore all executions of other clients

Checking just MPC is not enough

BUG : Queue implementation that always returns $\perp$

Checking just MPC is not enough

BUG : Queue implementation that always returns ⊥

Running on MPC does not reveal the bug:

```
q.enq(1) || q.enq(2) || q.deq() //⊥
```

Checking just MPC is not enough

BUG : Queue implementation that always returns ⊥

Running on MPC does not reveal the bug:

```
q.enq(1) || q.enq(2) || q.deq() //⊥
```

Running on an extension of MPC does:

```
q.enq(1)        || q.enq(2)
q.deq() //⊥     ||
```

Checking just MPC is not enough

BUG : Queue implementation that always returns ⊥

Running on MPC does not reveal the bug:

```
q.enq(1) || q.enq(2) || q.deq()//⊥
```

Running on an extension of MPC does:

```
q.enq(1)        || q.enq(2)
q.deq()//⊥      ||
```

Checking extensions of MPC is necessary

Checking just MPC is not enough

BUG : Queue implementation that always returns ⊥

Running on MPC does not reveal the bug:

```
q.enq(1) || q.enq(2) || q.deq()//⊥
```

Running on an extension of MPC does:

```
q.enq(1)     || q.enq(2)
q.deq()//⊥   ||
```

Checking extensions of MPC is necessary

We still use MPC:

Checking just MPC is not enough

BUG : Queue implementation that always returns ⊥

Running on MPC does not reveal the bug:

```
q.enq(1) || q.enq(2) || q.deq()//⊥
```

Running on an extension of MPC does:

```
q.enq(1)       || q.enq(2)
q.deq()//⊥     ||
```

Checking extensions of MPC is necessary

We still use MPC:

- For each spec execution, we calculate a minimal set of extensions that render it erroneous

Checking just MPC is not enough

BUG : Queue implementation that always returns ⊥

Running on MPC does not reveal the bug:

```
q.enq(1) || q.enq(2) || q.deq()//⊥
```

Running on an extension of MPC does:

```
q.enq(1)       || q.enq(2)
q.deq()//⊥     ||
```

Checking extensions of MPC is necessary

We still use MPC:

- For each spec execution, we calculate a minimal set of extensions that render it erroneous
- We check whether they also invalidate the implementation

Verifying a library for a given client

Verifying a library for a given client

Phase 1. Analyze the specification:

- Run model checker on **MostParallelClient[Spec]**
- Collect all behaviors of the spec
- **Foreach output calculate minimal set of forbidding extensions**

Verifying a library for a given client

Phase 1. Analyze the specification:

- Run model checker on **MostParallelClient[Spec]**
- Collect all behaviors of the spec
- **Foreach output calculate minimal set of forbidding extensions**

Phase 2. Check the implementation:

- Run model checker on **MostParallelClient[Impl]**
- For each G_{impl}, check that exists matching G_{spec}
- **For each G_{impl}, check that all forbidding extensions make it inconsistent**

Verifying a library for a given client

Phase 1. Analyze the specification:

- Run model checker on **MostParallelClient[Spec]**
- Collect all behaviors of the spec
- **Foreach output calculate minimal set of forbidding extensions**

Phase 2. Check the implementation:

- Run model checker on **MostParallelClient[Impl]**
- For each G_{impl}, check that exists matching G_{spec}
- **For each G_{impl}, check that all forbidding extensions make it inconsistent**

`enq(1)` `enq(2)` `deq():⊥`

G_{impl}

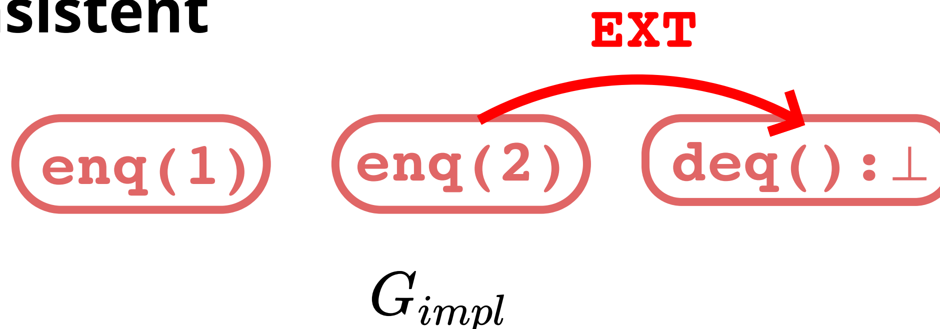
Verifying a library for a given client

Phase 1. Analyze the specification:

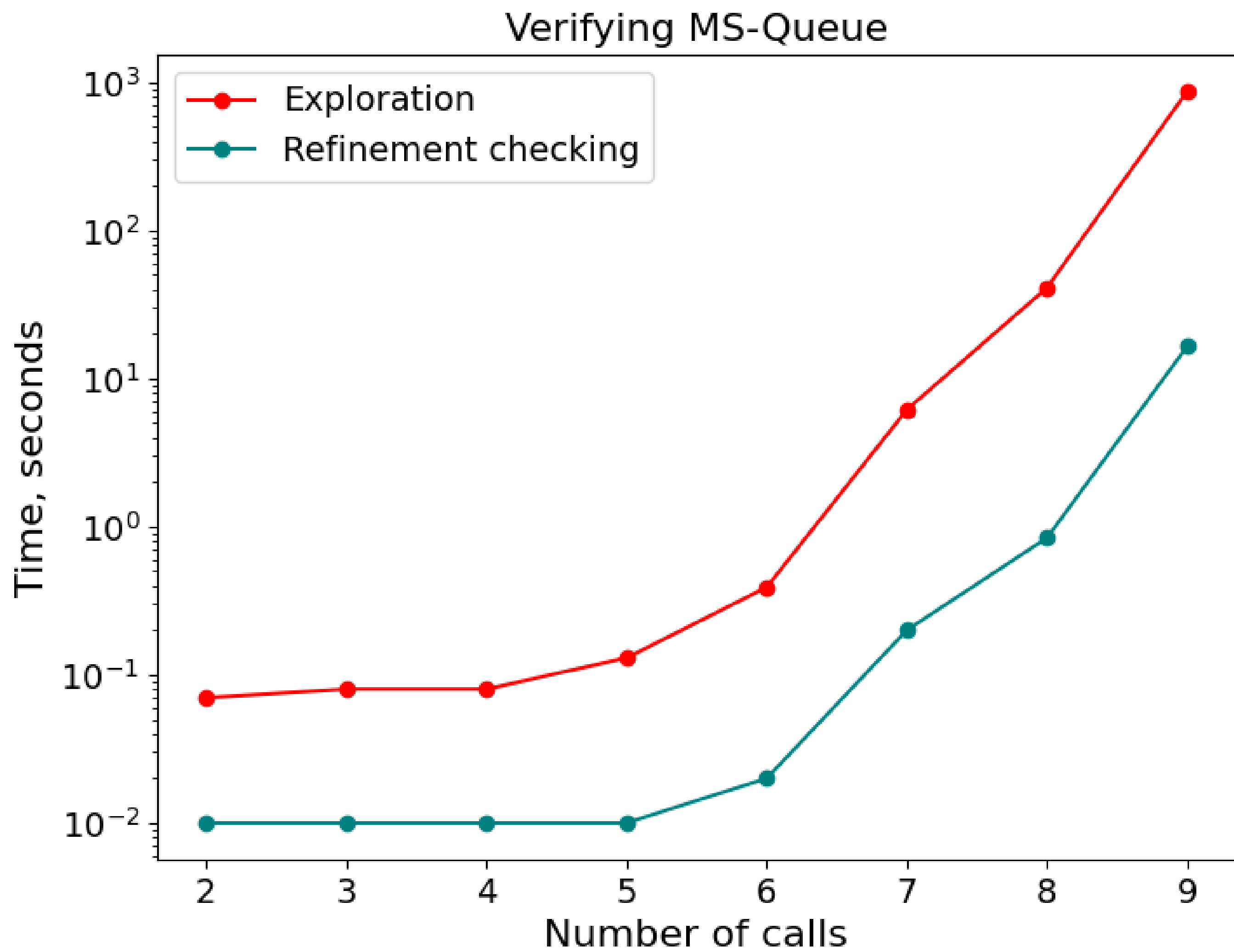
- Run model checker on **MostParallelClient[Spec]**
- Collect all behaviors of the spec
- **Foreach output calculate minimal set of forbidding extensions**

Phase 2. Check the implementation:

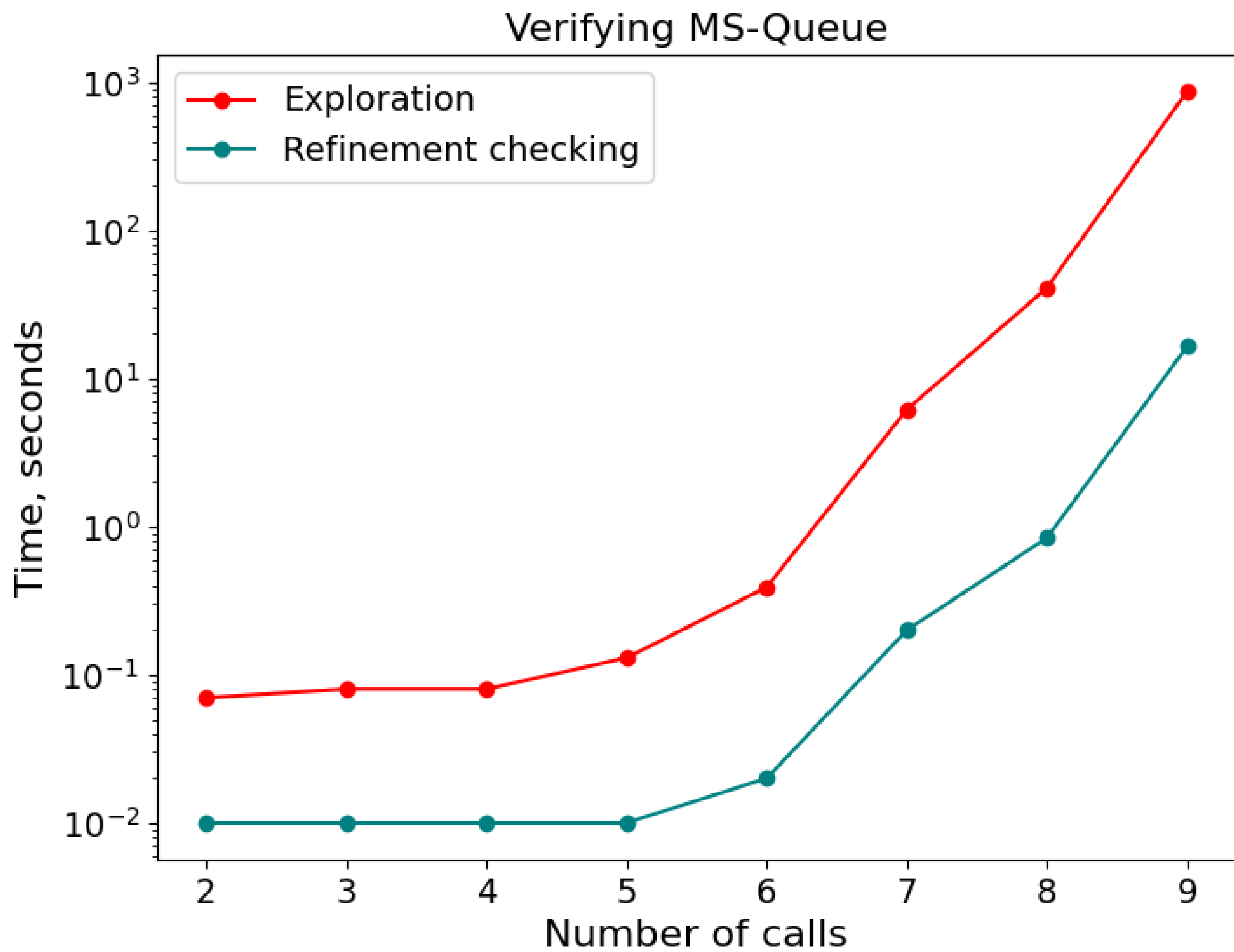
- Run model checker on **MostParallelClient[Impl]**
- For each G_{impl}, check that exists matching G_{spec}
- **For each G_{impl}, check that all forbidding extensions make it inconsistent**



How does it scale in the end?

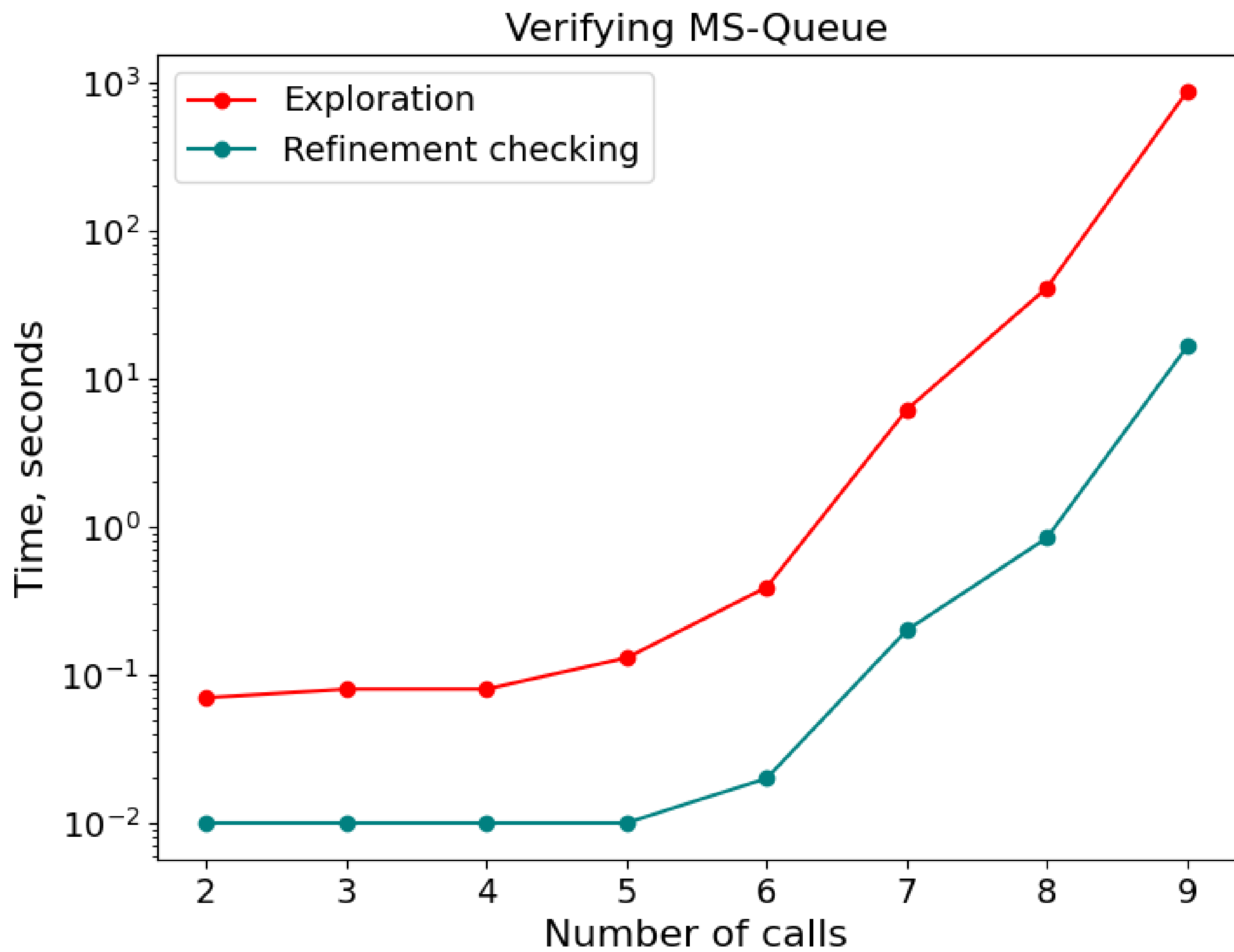


How does it scale in the end?



- Verifies MPCs with 7-9 threads

How does it scale in the end?



- Verifies MPCs with 7-9 threads
- Most time spent in enumeration (not refinement)

Our contribution

ReLinChe (Relaxed Linearizability Checker):

First tool for verifying **relaxed linearizability** of concurrent libraries automatically

Our contribution

ReLinChe (Relaxed Linearizability Checker):

First tool for verifying **relaxed linearizability** of concurrent libraries automatically

- Model checking

Our contribution

ReLinChe (Relaxed Linearizability Checker):

First tool for verifying **relaxed linearizability** of concurrent libraries automatically

- Model checking
- Partial lock for specification

Our contribution

ReLinChe (Relaxed Linearizability Checker):

First tool for verifying **relaxed linearizability** of concurrent libraries automatically

- Model checking
- Partial lock for specification
- MPC for verifying all clients

Our contribution

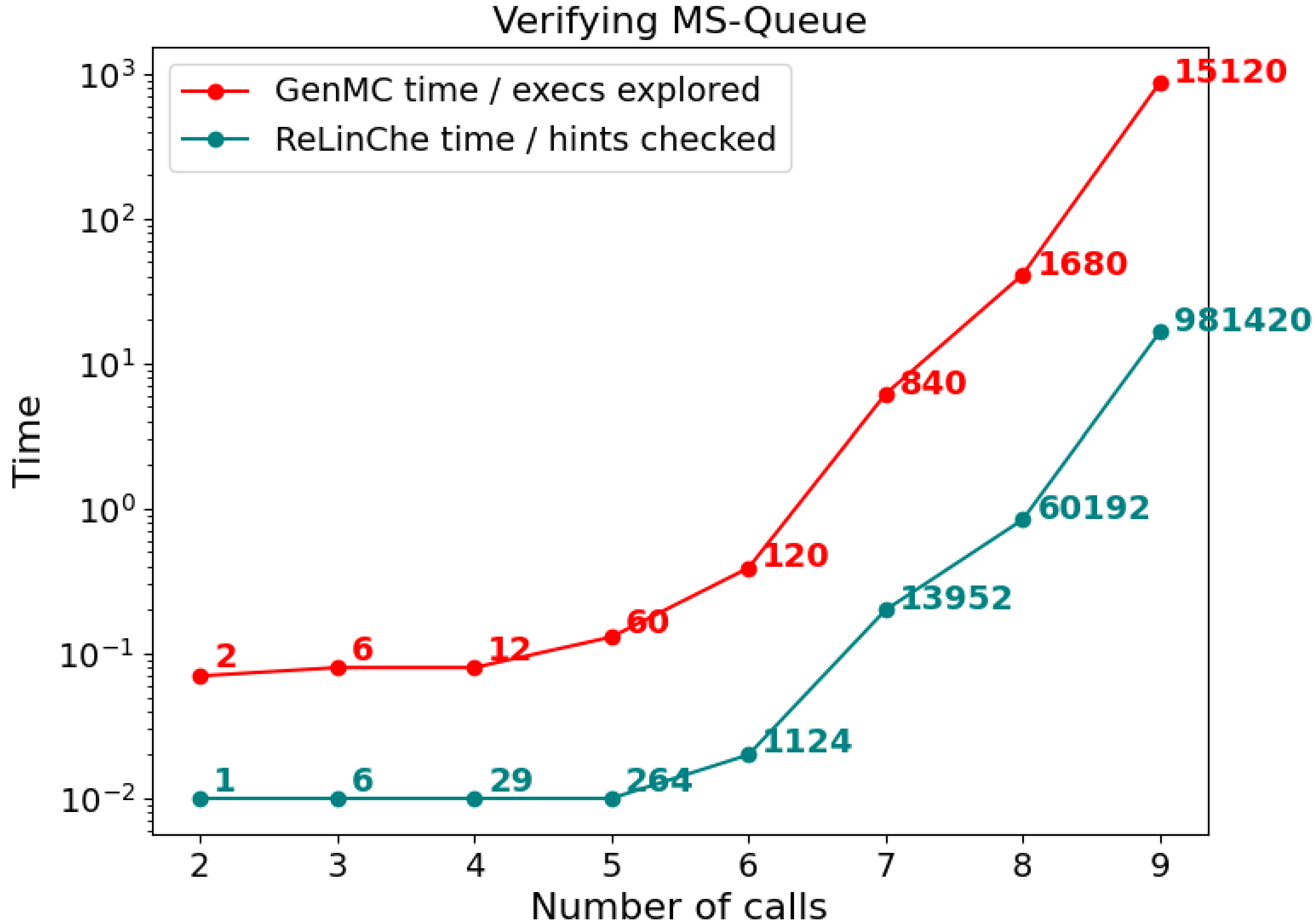
ReLinChe (Relaxed Linearizability Checker):

First tool for verifying **relaxed linearizability** of concurrent libraries automatically

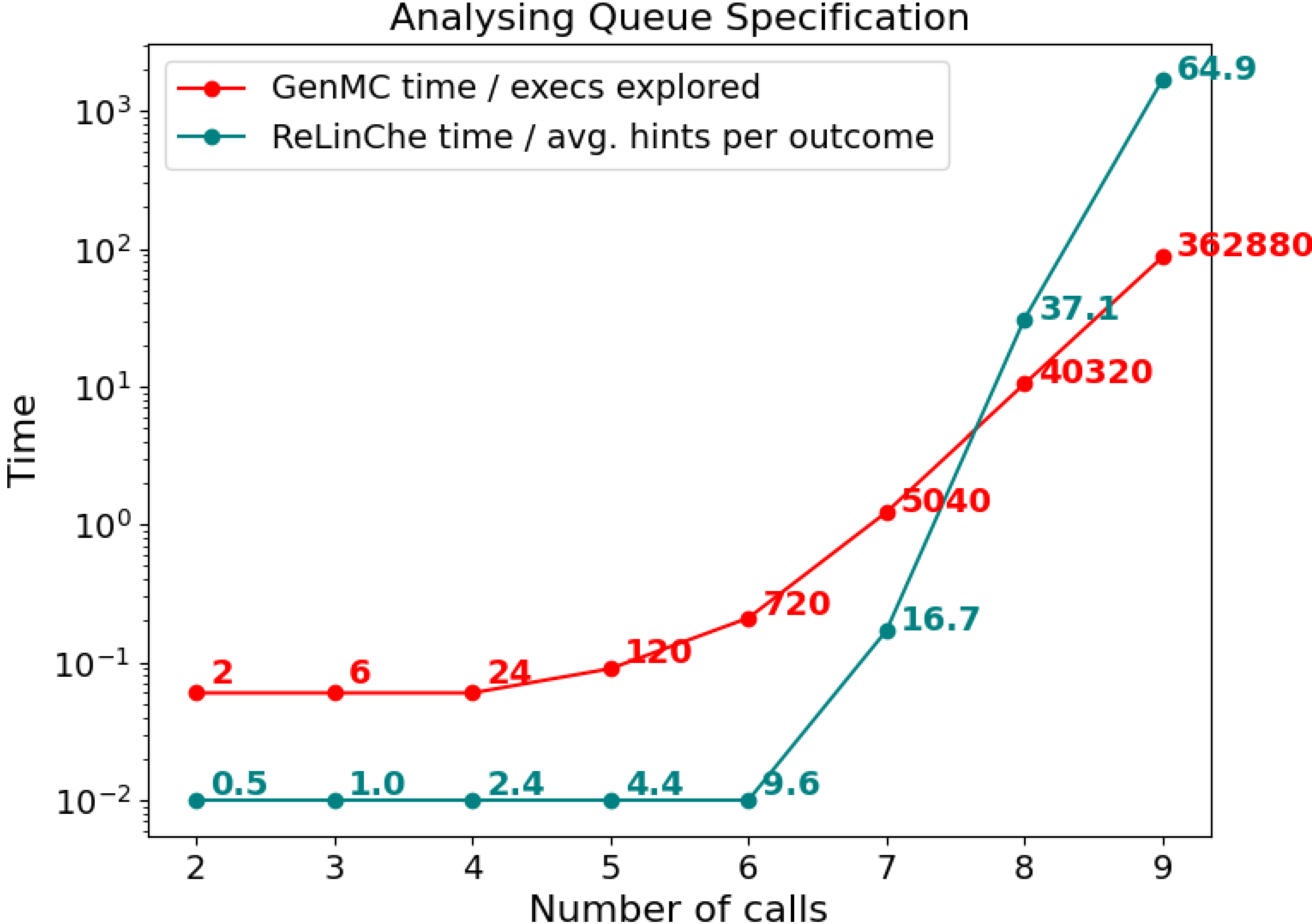
- Model checking
- Partial lock for specification
- MPC for verifying all clients
- Found **linearizability violation** in libcds

Spare slides

How does it scale in the end?



How does it scale in the end?



Subtle counter-examples that ReLinChe found:

Michael list from libcds:

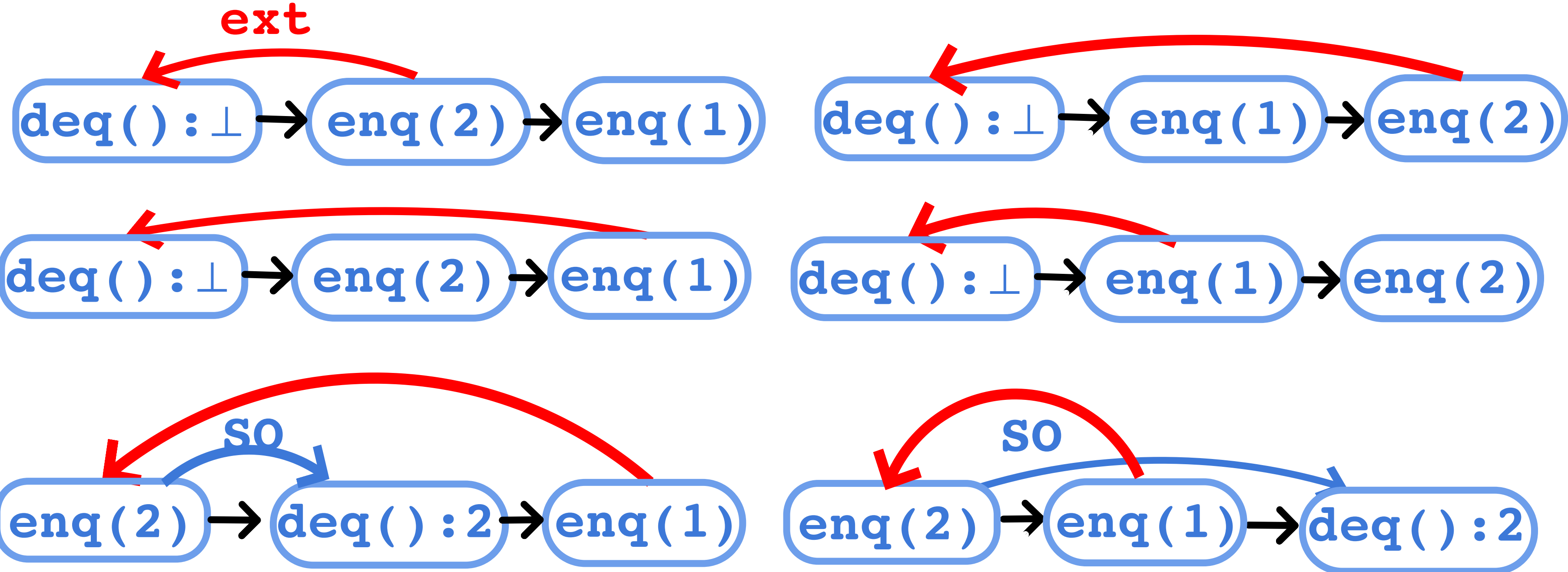
```
insert(1)         ||  insert(2)  ||  insert(3)
remove(3)//⊥      ||             ||  remove(1)//⊥
```

Subtle counter-examples that ReLinChe found:

Herlihy-Wing queue with wrong access mode:

```
q.enq(1) || q.enq(3)    || q.deq()//2 || q.deq()//4
q.enq(2) || q.deq()//1  || q.enq(4)   ||
```

Example of minimal extension set



only 3 / 19 extensions are useful