Checking Correctness of Concurrents Objects: Tractable Reductions to Reachability

Ahmed Bouajjani
LIAFA, Univ Paris Diderot - Paris 7

Joint work with

Michael Emmi        Constantin Enea        Jad Hamza
IMDEA                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

T1
Push(0)
Pop(1)

... 

Tn
Push(1)
Pop(0)
Empty(true)

Low Level Representation

Methods Implementation

Push
Pop
Empty
Abstract (Client) View

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

```
Push(1)  Push(0)  Pop(0)  Pop(1)  Empty(true)
```

Abstract (Client) View

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


<table>
<thead>
<tr>
<th>Push(1)</th>
<th>Push(0)</th>
<th>Pop(0)</th>
<th>Pop(1)</th>
<th>Empty(true)</th>
</tr>
</thead>
</table>

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

\[
\begin{align*}
\text{Push(0)} & \quad \text{Pop(1)} \\
\text{Push(1)} & \quad \text{Pop(0)} & \text{Empty(true)}
\end{align*}
\]

• 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

- Reorder call/return events, while preserving returns $\rightarrow$ calls
- Find “linearization points” within execution time intervals
- s.t. match some sequential execution

Valid sequence in the sequential specification

Linearizability $\iff$ Observational Refinement

[Herlihy, Wing, 1990], [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 $L \times M$ 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 $e$:

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

where

- $O = \text{Operations}(e)$
- label: $O \rightarrow M \times V \times V$
- $<$ is a partial order s.t.

$O_1 < O_2$ iff $\text{Return}(O_1)$ is before $\text{Call}(O_2)$ in $e$

$c(\text{push}, 1) \ r(\text{push}, \text{tt}) \ c(\text{pop}, -) \ c(\text{pop}, -) \ r(\text{pop}, 1) \ c(\text{push}, 2) \ r(\text{push}, \text{tt}) \ r(\text{pop}, 2)$
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
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 \( S \).

\[ L_c \text{ reference concurrent implementation} = L_s + \text{lock/unlock at beginning/end of each method} \]

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

\[ h_1 \leq h_2 \ (h_1 \text{ is weaker than } h_2) \]

iff

\[ h_1 \text{ has less constraints than } h_2 \]

Lemma:

\[ (h_1 \leq h_2 \text{ and } h_2 \text{ is in } H(L)) \implies h_1 \text{ is in } H(L) \]
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 $H(S) \Rightarrow h$ is not in $H(S) —$ Violation!
Histories are Interval Orders

Interval Orders = partial order \((O, <)\) such that
\((o_1 < o_1' \text{ and } o_2 < o_2')\) implies \((o_1 < o_2' \text{ or } o_2 < o_1')\)

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: $\text{past}(o) = \{o' : o' < o\}$
- Lemma [Rabinovitch'78]:
  The set $\{\text{past}(o) : o \text{ in } O\}$ is linearly ordered
- The *length* of the order = number of pasts - 1
Canonical Representation of Interval Orders

- Mapping $I : O \rightarrow [n]^2$ where $n = \text{length}(h)$ [Greenough '76]
- $I(o) = [i, j]$, with $i, j \leq n$, such that
  
  
  $$i = |\{\text{past}(o') : o' < o\}| \quad \text{and} \quad j = |\{\text{past}(o') : \neg (o < o')\}| - 1$$

![Diagram of interval orders and operations]

- $I(\text{push}(1)) = [0, 0]$
- $I(\text{pop}(1)) = [1, 1]$
- $I(\text{push}(2)) = [2, 2]$
- $I(\text{push}(3)) = [3, 3]$
- $I(\text{pop}(3)) = [1, 3]$
- $I(\text{pop}(2)) = [4, 4]$

Length = 4
Let $A_k$ maps each $h$ to some $h' \leq h$ of length $k$

$\Rightarrow$ Keep precise the information about the $k$ last intervals
Counting Representation of Interval Orders

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

• $h = (O, \prec)$ an IO with canonical representation $I:O \rightarrow [k]^2$
• Associate a counter with each operation type and interval
• $\prod(h)$ is the Parikh image of $h$
• It represents the multi-set $\{ [\text{label}(o), I(o)] : o \in O \}$

Prop: $H_k(e)$ is in $H_k(L)$ iff $\prod(H_k(e))$ is in $\prod(H_k(L))$
Reduction to Reachability with Counters

\[ H_k(L) \text{ subset of } H_k(S) \iff \Pi(H_k(L)) \text{ subset of } \Pi(H_k(S)) \]

- Consider **k-bounded-length abstract histories**
- Track histories of L using a **finite number of counters**
- Use an **arithmetic-based representation of** \( \Pi(H_k(S)) \)
- \( \Pi(H_k(S)) \) can be either computed, or given manually
- Check that **\( \Pi(H_k(S)) \text{ is an invariant} \)**
Experimental Results: Coverage

Comparison of violations covered with $k \leq 4$

- Data point: Counts in logarithmic scale over all executions (up to 5 preemptions) on Scal’s nonblocking bounded-reordering queue with $\leq 4$ enqueue and $\leq 4$ 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

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

- Static detection of injected refinement violations with CSeq & CBMC.
- Program $P_{ij}$ 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 $A_k$.
- 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

• \( u \cdot v : Q \& u : ENQ^* \rightarrow Enq(x) \cdot u \cdot Deq(x) \cdot v : Q \)

• \( u \cdot v : Q \& \text{ no unmatched } Enq \text{ in } u \rightarrow u \cdot Emp \cdot v : Q \)

Stack

• \( u \cdot v : S \& \text{ no unmatched } Push \text{ in } u \rightarrow Push(x) \cdot u \cdot Pop(x) \cdot v : S \)

• \( u \cdot v : S \& \text{ no unmatched } Push \text{ in } u \rightarrow u \cdot Emp \cdot v : S \)
Order Violation

FIFO violation:

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

\[
\text{Push}_1 \\
\hline
\text{EMP} \\
\hline
\text{Pop}_1
\]
Empty Violation

Recognizing Bad Patterns using Regular Automata

EMP

Push

Push

Push

Push

 EMP

Pop

Pop

Pop

Pop

Push

Push

Push

Push

Pop

Pop

Pop

Pop

Push

Push

Push

Push

Pop

Pop

Pop

Pop

Call

Call

Call
Order Violation cont.

\[ \text{Push}_2 \]

\[ \text{Push}_1 \]

\[ \text{Pop}_2 \]

\[ \text{Push}_1 \]

\[ \text{Pop}_1 \]

\[ \text{Push}_1 \]

\[ \text{Pop}_1 \]

\[ \text{Push}_1 \]
Automaton for Empty Violation

Recognized by:

$q_0$ 

$\Sigma_2$ 

$\text{call } Push(1)$ 

$q_1$ 

$\Sigma_2$ 

$\text{ret } Push(1)$ 

$q_2$ 

$\Sigma_2$ 

$\text{call } EMP()$ 

$q_3$ 

$\Sigma_2$ 

$\text{call } Pop(1)$ 

$q_4$ 

$\Sigma_2$ 

$\text{ret } Push(1)$ 

$\text{ret } EMP()$
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