Tractable Refinement Checking for Concurrent Objects

Constantin Enea
LIAFA, CNRS & University Paris Diderot - Paris 7

joint work with Ahmed Bouajjani, Michael Emmi, Jad Hamza
Concurrent objects

- Abstracting shared data: concurrent collections (queue, stack, ...)
- Synchronization objects (mutex, semaphore, ...)

Diagram:

- Concurrent Object
  - Call (push, pop, ...)
  - Return value

Threads:
- Thread 1
- Thread 2
- ...
Atomic objects

Ensure an atomic view of the method calls

Thread 1

push(1)  pop ⇒ 2

Thread 2

push(2)  pop ⇒ 1

• Obvious solution: global locks
• Performance requirements ⇒ optimistic concurrency (fine-grain locking, CAS, …)
Refinement

Reference implementation

class AtomicStack {
    cell* top;
    Lock l;

    void push (int v) {
        l.lock();
        top->next = malloc(sizeof *x);
        top = top->next;
        top->data = v;
        l.unlock();
    }

    int pop () {
        ...
    }
}

Efficient implementation

class TreiberStack {
    cell* top;

    void push (int v) {
        cell* t;
        cell* x = malloc(sizeof *x);
        x->data = v;
        do {
            t = top;
            x->next = top;
            top = top->next;
            t->data = v;
        } while (!CAS(&top,t,x));
    }

    int pop () {
        ...
    }
}

For every Client,
Client x Impl included in Client x Spec
Violating Refinement

Thread 1

Thread 2

pushed: 1, 2, 3
popped: 1, 3, EMPTY

PROBLEM
not admitted by atomic stack
Insidious Errors

HARD TO REPRODUCE
• memory management
• specific thread interleaving

HARD TO DIAGNOSE
program assertions don’t suffice

DEMANDS AUTOMATION
Automating Refinement
Checking

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

CHALLENGES

• enumeration of programs
• enumeration of executions
• checking inclusion
Linearizability [Herlihy & Wing 1990]

- Find “linearization points” within execution time intervals
- The order defined by the linearization points is admitted by the specification
- Impl is linearizable w.r.t. Spec iff Impl refines Spec (when Spec is atomic) [Filipovic et al. 2009, Bouajjani et al. 2015]
Enumeration

Exponentially-many linearizations

AtomicStack?

push(1) pop⇒3 pop⇒EMPTY

push(1) pop⇒1 push(2) push(3) pop⇒EMPTY

push(1) pop⇒1 push(2) push(3) pop⇒EMPTY

push(1) pop⇒1 push(2) push(3) pop⇒EMPTY

push(1) pop⇒1 push(2) push(3) pop⇒EMPTY

push(1) pop⇒1 push(2) push(3) pop⇒EMPTY

push(1) pop⇒1 push(2) push(3) pop⇒EMPTY
## Complexity

<table>
<thead>
<tr>
<th></th>
<th>Reachability</th>
<th>Linearizability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single trace</td>
<td>NL-complete</td>
<td>NP-complete&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>n threads</td>
<td>PSPACE-complete</td>
<td>EXPSPACE-complete&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>∞ threads</td>
<td>EXPSPACE-complete</td>
<td>Undecidable&lt;sup&gt;c&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

- Demands approximation analyses
- In this talk: focus on bug-finding

---

<sup>a</sup> Testing Shared Memory. Gibbons et al. 1996  
<sup>b</sup> Linearizability is EXPSPACE-complete Hamza 2015  
<sup>c</sup> Verifying Concurrent Programs Against Sequential Specifications Bouajjani et al. 2013
Parametrized under-approximations
[Bouajjani, Emmi, E, Hamza, POPL’15]

- Characterization of refinement using histories (partial orders)
  - reduce refinement to a history-set inclusion
- Parametrized under-approximation to solve the inclusion
  - histories are interval orders
  - parametrized by length
  - efficient representation using counters
  - efficient reduction to reachability (dynamic/static analysis)
- Experiments
  - Scalability: Efficient in practice
  - Coverage: Small length needed to catch violations
Parametrized under-approximations
[Bouajjani, Emmi, E, Hamza, POPL’15]

- **Characterization** of refinement using histories (partial orders)
  - reduce refinement to a history-set inclusion
- Parametrized under-approximation to solve the inclusion
  - histories are interval orders
  - parametrized by length
  - efficient representation using counters
  - efficient reduction to reachability (dynamic/static analysis)
- Experiments
  - **Scalability**: Efficient in practice
  - **Coverage**: Small length needed to catch violations
Histories

push(1) \rightarrow pop \Rightarrow 3

pop \Rightarrow 1 \rightarrow push(2) \rightarrow push(3) \rightarrow pop \Rightarrow \text{EMPTY}

\text{happens-before partial order}
History Inclusion

\[ \text{Hist}(L) = \text{the histories of all executions of } L \]
\[ \quad \text{(arbitrary calls with arbitrary many threads)} \]

**THEOREM**

\[ L \text{ refines } S \iff \text{Hist}(L) \subseteq \text{Hist}(S) \]

- \((=>)\) Given \( h \) in \( \text{Hist}(L) \), construct a client \( P_h \) that imposes all the happen-before constraints of \( h \).
- \((<=)\) Clients cannot distinguish executions with the same history.
Parametrized under-approximations
[Bouajjani, Emmi, E, Hamza, POPL’15]

- Characterization of refinement using histories (partial orders)
  - reduce refinement to a history-set inclusion

- Parametrized under-approximation to solve the inclusion
  - histories are interval orders
  - parametrized by length
  - efficient representation using counters
  - efficient reduction to reachability (dynamic/static analysis)

- Experiments
  - Scalability: Efficient in practice
  - Coverage: Small length needed to catch violations
Approximating

**GOAL**
parameterized approximation $A_k$
- complete as $k \to \infty$
- tractable for fixed $k$
- violations with small $k$

**HYPOTHESIS**
violations surface in histories with **low-complexity orderings**
Ordering complexity

execution histories are *interval orders*

INTERVAL LENGTH

smallest maximum integral interval bound
History Abstraction

**LEMMA**
Libraries closed under weakening

LENGTH 4

push(1) → pop⇒3 → pop⇒EMPTY
pop⇒1 push(2) push(3)

LENGTH 1

push(1) → push⇒3 → push⇒EMPTY
pop⇒1 push(2) push(3)

STILL A VIOLATION
Interval-Length Bounding

- $A_k(h) = \text{history of length } k \text{ (keeping last } k \text{ intervals precise and merge the remaining ones)}$

Checking $A_k(\text{Hist} (L)) \subseteq \text{Hist}_k (S)$ instead of $\text{Hist}(L) \subseteq \text{Hist} (S)$
  - construct a formula $\Psi_S$ representing $\text{Hist}_k (S)$
  - check $A_k(h) \models \Psi_S$ for every $h \in \text{Hist}(L)$

Counter-based representations of histories
Using counter-based representations

Checking $A_k(\text{Hist}(L)) \subseteq \text{Hist}_k(S)$

- Generating $A_k(\text{Hist}(L)) = \text{instrumenting the most general client of L with counter increments/decrements}$
- adding the assertion $\Psi_S$ representing $\text{Hist}_k(S)$
  - obtained manually for common objects (stack, queue, …)
  - automatically for context-free specifications
Parametrized under-approximations
[Bouajjani, Emmi, E, Hamza, POPL’15]

- Characterization of refinement using histories (partial orders)
  - reduce refinement to a history-set inclusion
- Parametrized under-approximation to solve the inclusion
  - histories are interval orders
  - parametrized by length
  - efficient representation using counters
  - efficient reduction to reachability (dynamic/static analysis)

- Experiments
  - **Scalability:** Efficient in practice
  - **Coverage:** Small length needed to catch violations
Empirically (dynamic analysis)

- Small bounds suffice as sample-size increases.
- Exponentially-lower monitoring overhead.

Execution sample vs. violations covered:
- Missed violations due to small sample.

Execution size vs. monitoring overhead:
- ~1000x
- ~2x

1x10^3 executions — 2.4x10^6 executions

Linearization vs. approximation k=2

Operation counting vs. execution size.

Histoires
Violations
Covered w/ k=4
Covered w/ k=3
Covered w/ k=2
Covered w/ k=1
Empirically (static analysis)

- Static analysis for finding refinement violations
- Reduction to existing tools
- CSeq, with CBMC backend (bounded model checking)

<table>
<thead>
<tr>
<th>Library</th>
<th>P</th>
<th>k</th>
<th>Unrolling</th>
<th>Rounds</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Michael-Scott Queue (Head)</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>24.76s</td>
</tr>
<tr>
<td>Michael-Scott Queue (Tail)</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>45.44s</td>
</tr>
<tr>
<td>Treiber Stack (ABA)</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>52.59s</td>
</tr>
<tr>
<td>Treiber Stack (push)</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>24.46s</td>
</tr>
<tr>
<td>Treiber Stack (pop)</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>15.16s</td>
</tr>
<tr>
<td>Elimination Stack</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>317.79s</td>
</tr>
<tr>
<td>Elimination Stack</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>222.04s</td>
</tr>
<tr>
<td>Elimination Stack</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>434.84s</td>
</tr>
<tr>
<td>Lock-coupling Set</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>11.27s</td>
</tr>
<tr>
<td>LFDS Queue</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>77.00s</td>
</tr>
</tbody>
</table>
Conclusion

- equivalence between refinement and history inclusion
- parametrized under-approximation schema to solve \( \text{Hist}(L) \subseteq \text{Hist}(S) \)
  - abstracting histories with low-complexity partial orders (bounded interval length)

Future work:

- **Complete verification**: Leverage insights on violations?
- **Compiler optimizations**: Refinement checking *without fixed reference impls*?
- **Weaker abstractions**: e.g., *causal consistency* in place of atomicity?
THE END