Towards formal verification of imperative concurrent data structures

{Alejandro, César} Sánchez
IMDEA Software - Madrid - Spain

Workshop ReSeCo - FaMAF - Dec. 17, 2009
What are we interested in?
What are we interested in?

- Imperative programs
What are we interested in?

- Imperative programs
- Concurrent data structures

\[ P_1 \parallel \cdots \parallel P_n \]
What are we interested in?

- Imperative programs
- Concurrent data structures

$P_1 \parallel \cdots \parallel P_n$

data structures (heap)
What are we interested in?

- Imperative programs
- Concurrent data structures
- Temporal property (safety, liveness)

$P_1 \parallel \cdots \parallel P_n \models \varphi$

data structures
(heap)
What are we interested in?

- Imperative programs
- Concurrent data structures
- Temporal property (safety, liveness)
- Formal verification

\[
P_1 \parallel \cdots \parallel P_n \models \varphi
\]

- data structures
  (heap)
What are we interested in?

- Imperative programs
- Concurrent data structures
- Temporal property (safety, liveness)
- Formal verification

\[ P_1 \parallel \cdots \parallel P_n \models \varphi \]  \text{LTL (☐.☐.☐.☐.☐)}

data structures (heap)
What are we interested in?

- Imperative programs
- Concurrent data structures
- Temporal property (safety, liveness)
- Formal verification

\[ P_1 \parallel \cdots \parallel P_n \models \varphi \} \text{ LTL ( } \Box, \diamond, \circ, \mathcal{U} \text{ )} \]

Regional Logic \{ data structures (heap) \}
What are we interested in?

- Imperative programs
- Concurrent data structures
- Temporal property (safety, liveness)
- Formal verification

Regional Logic \{ data structures (heap) \}

\[ P_1 \parallel \cdots \parallel P_n \models \varphi \] \[ \text{LTL ( } \Box, \Diamond, \circ, \gamma) \]

Verification Diagram
Reasoning about the heap
Reasoning about the heap

- Separation Logic
  
  Hoare logic extension to reason about shared mutable data structure
Reasoning about the heap

- Separation Logic
  
  Hoare logic extension to reason about shared mutable data structure

emp
Reasoning about the heap

- Separation Logic
  Hoare logic extension to reason about shared mutable data structure
  emp , x \rightarrow 
  x \rightarrow 3
Reasoning about the heap

Separation Logic

Hoare logic extension to reason about shared mutable data structure

\[ [P_0 \ast P_1] s h \iff \exists h_0, h_1 \bullet h_0 \perp h_1 \land h_0 \cdot h_1 = h \land [P_0] s h_0 \land [P_1] s h_1 \]
Reasoning about the heap

**Separation Logic**

Hoare logic extension to reason about shared mutable data structure

\[ \text{emp} \rightarrow \top \]

\[
[P_0 \ast P_1] s h \iff \exists h_0, h_1 \cdot h_0 \uparrow h_1 \land h_0.h_1 = h \land [P_0] s h_0 \land [P_1] s h_1
\]

\[ P_1 \quad P_2 \]

\[ P_1 \quad P_2 \]
Reasoning about the heap

- Separation Logic

Hoare logic extension to reason about shared mutable data structure

\[ \text{emp} \mapsto *, -* \]

\[ [P_0 \ast P_1] s \ h \ \leftrightarrow \ \exists h_0, h_1 \cdot h_0 \perp h_1 \land h_0 . h_1 = h \land [P_0] s \ h_0 \land [P_1] s \ h_1 \]
Reasoning about the heap

Regional Logic
Reasoning about the heap

Regional Logic

Classical first order logic
Reasoning about the heap

Regional Logic

Classical first order logic

Based on Hoare logic
Reasoning about the heap

Regional Logic
  Classical first order logic
  Based on Hoare logic
  Ghost fields/variables
Reasoning about the heap

Regional Logic

Classical first order logic
Based on Hoare logic
Ghost fields/variables
Region manipulation language: $\text{emp}, \{\}, \cup, \cap, -$
Reasoning about the heap

Regional Logic

- Classical first order logic
- Based on Hoare logic
- Ghost fields/variables
- Region manipulation language: \( \text{emp}, \{ \}, \cup, \cap, \setminus \)
- Region assertion language: \( R_1 \subseteq R_2 \)
Reasoning about the heap

Regional Logic

Classical first order logic
Based on Hoare logic
Ghost fields/variables
Region manipulation language: \text{emp}, \langle \rangle, \cup, \cap, -
Region assertion language: \( R_1 \subseteq R_2, R_1 \# R_2 \)
Reasoning about the heap

Regional Logic

- Classical first order logic
- Based on Hoare logic
- Ghost fields/variables
- Region manipulation language: $\text{emp, } \{ \}, \cup, \cap, -$ 
- Region assertion language: $R_1 \subseteq R_2, R_1 \# R_2, R_1.f \subseteq R_2$
Reasoning about the heap

Regional Logic

Classical first order logic
Based on Hoare logic
Ghost fields/variables
Region manipulation language: $\text{emp}, \{\}, \cup, \cap, -$ 
Region assertion language: $R_1 \subseteq R_2, R_1 \# R_2, R_1.f \subseteq R_2, R_1.f \# R_2$
Reasoning about the heap

Regional Logic

Classical first order logic
Based on Hoare logic
Ghost fields/variables
Region manipulation language: emp, ∅, ∪, ∩, −
Region assertion language: $R_1 \subseteq R_2, R_1 \# R_2, R_1.f \subseteq R_2, R_1.f \# R_2$

$\forall x : K \in R \mid P$
Verification Diagrams
Verification Diagrams

\[ P \models \varphi \]
Verification Diagrams

\[ P \vdash \varphi \]

\[ \Psi \]
Verification Diagrams

Representation of a system by FTS

\[ P \models \varphi \]

\[ \Psi \]
Verification Diagrams

- Representation of a system by FTS
- Sound & complete

\[ P \models \varphi \quad \downarrow \quad \Psi \]
Verification Diagrams

- Representation of a system by FTS
- Sound & complete

\[ \Psi = \langle N, N_0, E, \mu, F, \eta, \Delta, f \rangle \]
Verification Diagrams

- Representation of a system by FTS
- Sound & complete

$$\Psi = \langle N, N_0, E, \mu, FA, \eta, \Delta, f \rangle$$

- $n_1$
- $n_2$
Verification Diagrams

- Representation of a system by FTS
- Sound & complete

\[ \Psi = \langle N, N_0, E, \mu, F, \eta, \Delta, f \rangle \]
Verification Diagrams

- Representation of a system by FTS
- Sound & complete

\[ \Psi = \langle N, N_0, E, \mu, F, \eta, \Delta, f \rangle \]
Verification Diagrams

- Representation of a system by FTS
- Sound & complete

\[ \Psi = \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle \]
Verification Diagrams

- Representation of a system by FTS
- Sound & complete

\[ \Psi = \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle \]
Main Idea
Main Idea

Concurrent Data Structure
Main Idea

Concurrent Data Structure
Main Idea

Concurrent Data Structure

Most General Client [N]
(extended with GV)
Main Idea

Concurrent Data Structure

Most General Client [N]
(extended with GV)
Main Idea

Concurrent Data Structure

<table>
<thead>
<tr>
<th>Most General Client [N]</th>
<th>( \Psi )</th>
<th>( \varphi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(extended with GV)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Verification conditions like: initialization, consecution, acceptance, fairness, satisfaction...
Main Idea

Concurrent Data Structure

Most General Client \([N]\) (extended with GV)

Verification conditions like: initialization, consecution, acceptance, fairness, satisfaction...
Skiplists
Skiplists

- Sorted list of elements
Skiplists

- Sorted list of elements
Skiplists

- Sorted list of elements
- Hierarchy of linked lists
Skiplists

- Sorted list of elements
- Hierarchy of linked lists
Skiplists

- Sorted list of elements
- Hierarchy of linked lists
Skiplists

- Sorted list of elements
- Hierarchy of linked lists
Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
Fine-grained lock-coupling concurrent skiplists
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[ \text{insert}[^k](9) \]
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[
\text{insert}^{[k]}(9) \quad \text{level} = 2
\]
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[ \text{insert}^{[k]}(9) \quad \text{level} = 2 \]
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[ \text{insert}_{[k]}(9) \quad \text{level} = 2 \]
Fine-grained lock-coupling concurrent skip lists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

$\text{insert}^k(9) \quad \text{level} = 2$
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[ \text{insert}^{[k]}(9) \quad \text{level} = 2 \]
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[ \text{insert}^k(9) \text{ level} = 2 \]
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[ \text{insert}^{[k]}(9) \quad \text{level} = 2 \]
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[ \text{insert}^{[k]}(9) \quad \text{level} = 2 \]
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

$\text{insert}^{[k]}(9)$

level = 2

Diagram showing a skiplist with nodes at positions $\infty$, 2, 4, 6, 8, 10, and $\infty$, with locks at positions 6 and 8.
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[ \text{insert}^k(9) \quad \text{level} = 2 \]
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

$insert[9](k)$

level = 2
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[ \text{insert}^{[k]}(9) \quad \text{level} = 2 \]
Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

\[ \text{insert}^{[k]}(9) \quad \text{level} = 2 \]
Fine-grained lock-coupling CSL
Fine-grained lock-coupling CSL

Algorithm 4 Insertion on a lock-coupling concurrent skip-list

1. procedure Insert(SkipList sl, Key key, Value v)
2.   Node *node = sl.search(key) // lang: unsafe
3.   bool - predecessor()
4.   Node *node = node
5.   node->lock(node) // mk := mk + 1 (pred, forward)[mk]
6.   node->cursor = node->forward[1]
7.   node->lock(node) // mk := mk + 1 (cursor, forward)[mk]
8.   for i = 0 to log(i) do
9.     if i < itl then
10.        predecessor(), lock() // mk := mk + 1 (pred, forward)[mk]
11.        cursor = node->forward[i]
12.        cursor.lock(node) // mk := mk + 1 (cursor, forward)[mk]
13.        end if
14.     while cursor.key < key do
15.        predecessor(), unlock() // mk := mk - 1 (pred, forward)[mk]
16.        cursor = node->forward[i]
17.        cursor = cursor->forward[i]
18.        cursor.lock(node) // mk := mk - 1 (cursor, forward)[mk]
19.     end while
20.     update[i] = pred
21. end for
22. if cursor.key = key then
23.   current = node
24. for i = 1 to log(i) do
25.     update[i].forward[i].unlock() // mk := mk - 1 (update[i], forward[i], forward)[mk]
26.     update[i].forward[i].unlock() // mk := mk - 1 (update[i], forward[i], forward)[mk]
27. end for
28. end if
29. x = CreateNode(key, v, normal)
30. for i = 1 to log(i) do
31.   x.forward[i] = update[i].forward[i]
32.   update[i].forward[i] = x // mk := mk - 1 (update[i], forward[i], forward)[mk]
33.   x.forward[i].lock(node) // mk := mk - 1 (x, forward[i], forward)[mk]
34.   update[i].lock(node) // mk := mk - 1 (update[i], forward[i], forward)[mk]
35. end for
36. end if
37. end procedure

insert(sl, v)
Fine-grained lock-coupling CSL

### Algorithm 4: Insertion on a lock-coupling concurrent skip list

```plaintext
insert(sl, v)
```

```plaintext
while curr.key < k do
    pred.locks[i].unlock()
    pred := curr
    curr := pred.forward[i]
    curr.locks[i].lock()

    //@ m_r := m_r - (pred, forward[i])
end while

//@ m_r := m_r ∪ (curr, forward[i])
```

```
1: procedure Insert(SkipList sl, Key k, Value v) {
2:     Node *node = node[sl, insert(k, v)] // π
3:     sl = randomNode()
4:     Node *newNode = newNode()
5:     pred.locks[hash[k]].lock() // m := m ∪ (pred, forward[i])
6:     newNode.key = pred.forward[i]
7:     newNode.lock() // m := m ∪ (newNode, forward[i])
8:     for i = 0 downto 0 do
9:         if i = 0 then
10:             pred.locks[i].unlock() //= m := m ∪ (pred, forward[i])
11:             curr = pred.forward[i]
12:             curr.locks[i].lock() //= m := m ∪ (curr, forward[i])
13:             end if
14:         while curr.key < k do
15:             pred.locks[i].unlock()
16:             pred = curr
17:             curr = pred.forward[i]
18:             curr.locks[i].lock() //= m := m ∪ (curr, forward[i])
19:         end while
20:         update[i] = pred
21:     end for
22: }
```
Fine-grained lock-coupling CSL

\[\text{insert}(sl, v) \quad \text{search}(sl, v) \quad \text{remove}(sl, v)\]

\begin{verbatim}
while curr.key < k do
  pred.locks[i].unlock()
  pred := curr
  curr := pred.forward[i]
  curr.locks[i].lock()

// @ m_r := m_r - (pred, forward[i])
// @ m_r := m_r \cup (curr, forward[i])
end while
\end{verbatim}
Fine-grained lock-coupling CSL

\[ \text{insert}(s_l, v) \quad \text{search}(s_l, v) \quad \text{remove}(s_l, v) \quad \text{decide}(s_l) \]

\[
\textbf{while} \quad \text{curr.key} < k \quad \textbf{do} \\
\quad \text{pred.locks[i].unlock()} \quad \text{//@ } m_r := m_r - (\text{pred, forward[i]}) \\
\quad \text{pred := curr} \\
\quad \text{curr := pred.forward[i]} \\
\quad \text{curr.locks[i].lock()} \quad \text{//@ } m_r := m_r \cup (\text{curr, forward[i]}) \\
\textbf{end while}
\]
Fine-grained lock-coupling CSL

\[ \text{insert}(sl,v) \quad \text{search}(sl,v) \quad \text{remove}(sl,v) \quad \text{decide}(sl) \]
Fine-grained lock-coupling CSL

\[ \text{insert}(sl, v) \quad \text{search}(sl, v) \quad \text{remove}(sl, v) \quad \text{decide}(sl) \]

\[ T_i \]
Fine-grained lock-coupling CSL

\[
\text{insert}(sl, v) \quad \text{search}(sl, v) \quad \text{remove}(sl, v) \quad \text{decide}(sl)
\]

\[
T_i \models \Box \varphi_{\text{insert}}(i)
\]

\[
\varphi_{\text{insert}}(i) \triangleq \text{at}_{\text{insert}}^{[i]}_{8..36} \rightarrow \text{at}_{\text{insert}}^{[i]}_{8..36} \cup \text{at}_{\text{insert}}^{[i]}_{37}
\]
Fine-grained lock-coupling CSL

\[ \text{insert}(sl, v) \quad \text{search}(sl, v) \quad \text{remove}(sl, v) \quad \text{decide}(sl) \]

\[ \|_{j \in T_{ID} \setminus \{i\}} T_j \| T_i \models \Box \varphi_{\text{insert}}(i) \]

\[ \varphi_{\text{insert}}(i) \models \text{at}_{\text{insert}}^{[i]}_{8..36} \rightarrow \text{at}_{\text{insert}}^{[i]}_{8..36} \cup \text{at}_{\text{insert}}^{[i]}_{37} \]
Fine-grained lock-coupling CSL

\[
\parallel_{j \in T_{ID} - \{i\}} T_j \parallel T_i \models \square \varphi_{insert}(i)
\]

\[
\varphi_{insert}(i) \doteq at\_insert_{8..36}^{[i]} \rightarrow at\_insert_{8..36}^{[i]} \cup at\_insert_{37}^{[i]}
\]
Fine-grained lock-coupling CSL
Fine-grained lock-coupling CSL
Fine-grained lock-coupling CSL
Fine-grained lock-coupling CSL

\[ n_3 : \text{at\_insert}_{14..16,18,19,20}^{[2]} \land I_{\text{insert}}^{[2]} \]

\[ n_4 : \text{at\_insert}_{17}^{[2]} \land I_{\text{insert}}^{[2]} \]
Fine-grained lock-coupling CSL

14: while curr.key < k do
15:     pred.locks[i].unlock()  // @ m_r := m_r - (pred.forward[i])
16:     pred := curr
17:     curr := pred.forward[i]
18:     curr.locks[i].lock()  // @ m_r := m_r ∪ (curr.forward[i])
19: end while
Fine-grained lock-coupling CSL

Verification conditions

\[ \left\{ \tau_{\text{insert}_{9,13}}^{[1]} \right\} \leftarrow \{ \tau_{\text{insert}_{14,15,18,19}, \tau_{\text{insert}}}^{[i]} \} \]

\[ n_3 : at_{\text{insert}}^{[i]}_{14..16,18,19,20} \wedge I^{[i]}_{\text{insert}} \]

\[ \left\{ \tau_{\text{insert}_{17}}^{[i]} \right\} \leftarrow \left\{ \tau_{\text{insert}_{16}}^{[1]} \right\} \leftarrow \left\{ \tau_{\text{insert}}^{[j]} \right\} \]

\[ n_4 : at_{\text{insert}}^{[i]}_{17} \wedge I^{[i]}_{\text{insert}} \]
Fine-grained lock-coupling CSL

Verification conditions ✓

\[
\begin{align*}
\{ \tau_{\text{insert}_{9,13}}^{[1]} \} & \quad \{ \tau_{\text{insert}_{14,15,18,19}}^{[i]}, \tau_{-}^{[j]} \} \\
\text{n}_3 : & \quad \text{at}\_\text{insert}_{14..16,18,19,20}^{[i]} \land I_{\text{insert}}^{[i]} \\
\{ \tau_{\text{insert}_{17}}^{[i]} \} & \quad \{ \tau_{\text{insert}_{16}}^{[1]} \} & \quad \{ \tau_{-}^{[j]} \} \\
\text{n}_4 : & \quad \text{at}\_\text{insert}_{17}^{[i]} \land I_{\text{insert}}^{[i]}
\end{align*}
\]
Fine-grained lock-coupling CSL

Verification conditions ✓

Ψ

\[ \parallel j \in T_{ID} - \{i\} \; T_j \parallel T_i \models \varphi_{\text{insert}}(i) \]
Fine-grained lock-coupling CSL

Verification conditions ✓

\[ \Psi \]

\[ \big\|_{j \in T_{ID} - \{i\}} T_j \parallel T_i \models \varphi_{\text{insert}(i)} \]
Conclusions
Conclusions

A method to formally verify temporal properties over concurrent data structures
Conclusions

- A method to formally verify temporal properties over concurrent data structures
- Not just limited to safety properties
Conclusions

- A method to formally verify temporal properties over concurrent data structures
- Not just limited to safety properties
- A different approach to Separation Logic
Conclusions

- A method to formally verify temporal properties over concurrent data structures
- Not just limited to safety properties
- A different approach to Separation Logic
- Good results over many mutable data structures
Conclusions

- A method to formally verify temporal properties over concurrent data structures
- Not just limited to safety properties
- A different approach to Separation Logic
- Good results over many mutable data structures
- Experience shows possibility of working with parameterized VD
Future work
Future work

- Extend the work over other concurrent data structures
Future work

- Extend the work over other concurrent data structures
- Enrich verifications diagrams
Future work

- Extend the work over other concurrent data structures
- Enrich verifications diagrams
- Automatic generation of verification conditions
Future work

- Extend the work over other concurrent data structures
- Enrich verifications diagrams
- Automatic generation of verification conditions
- Analyze decidability of involved logics
Future work

- Extend the work over other concurrent data structures
- Enrich verifications diagrams
- Automatic generation of verification conditions
- Analyze decidability of involved logics
- Development of assisted decision procedures
Future work

- Extend the work over other concurrent data structures
- Enrich verifications diagrams
- Automatic generation of verification conditions
- Analyze decidability of involved logics
- Development of assisted decision procedures
- This is just the beginning
Questions ?