On the Correctness of Optimistic Composable Data Structures

Ahmed Hassan, Sebastiano Peluso, Roberto Palmieri and Binoy Ravindran

Systems Software Research Group
Virginia Tech

7th Workshop on the Theory of Transactional Memory WTTM - 2015
Concurrent Data Structures

- Different Designs and Implementations
  - Different ad-hoc approaches for proving correctness.

- Is there a unified model for concurrent data structures?
  - General enough.
  - Easy to use.
SWMR Model (Lev-Ari et. Al, DISC'14)
Shared States

- Data Structure is represented as a set of shared variables.
- The values of those variables is the **shared state** of the data structure.
Local States

- Operation is represented as a set of steps.
- The values of the operation's local variables before any step is the **local state** of the step.
SWMR Scenario
Validity
Validity
Validity

- All $S_i$ are sequentially reachable, so all $UO_i$ are valid.
Validity

- All $S_i$ are sequentially reachable, so all $UO_i$ are valid.
- Step $i$ in RO is valid if there is $S_j$ such that a sequential execution of RO starting from $S_j$ reaches $L_i$. 
Validity

- All $S_i$ are sequentially reachable, so all $UO_i$ are valid.
- Step $j$ in $RO$ is valid if there is $S_i$ such that a sequential execution of $RO$ starting from $S_i$ reaches $L_j$. 
Validity

- All $S_i$ are sequentially reachable, so all $UO_i$ are valid.

- Step $j$ in RO is valid if there is $S_i$ such that a sequential execution of RO starting from $S_i$ reaches $L_j$.

- Step $j$ in RO is valid if there is a “base point” where the “base condition” of step $j$ holds.
Validity

• How to prove validity for any data structure.
  – Identify the base conditions for each step in each operation (it is sufficient to do so only for steps that access the shared memory).

  – Prove that in any concurrent execution, every step has a base point that satisfies its base condition.
Validity

- How to prove validity for any data structure.
  - Identify the base conditions for each step in each operation (it is sufficient to do so only for steps that access the shared memory).
  - Prove that in any concurrent execution, every step has a base point that satisfies its base condition.
Regularity

UO₁  UO₂  UO₃  …………  UOₙ

RO₁

RO₂
Regularity

\[ \text{UO}_1 \quad \text{UO}_2 \quad \text{UO}_3 \quad \cdots \quad \text{UO}_n \]

\[ \text{RO}_1 \]

\[ \text{RO}_2 \]

\[ \text{UO}_1 \quad \text{UO}_2 \quad \text{UO}_3 \quad \cdots \quad \text{UO}_n \]

\[ \text{RO}_1 \]

\[ \text{RO}_2 \]
Regularity

\[ \text{UO}_1 \quad \text{UO}_2 \quad \text{UO}_3 \quad \ldots \quad \text{UO}_n \]

\[ \text{RO}_1 \quad \text{RO}_2 \]

Linearizable
Regularity
Regularity

- Acceptable base points for RO's return step are only $S_1$, $S_2$, $S_3$.
  - Observes either the last update or a concurrent update.
Example

**Function** remove\( (n) \)

\[
\begin{align*}
p & \leftarrow \bot \\
\text{next} & \leftarrow \text{read}(\text{head}.\text{next}) \\
\textbf{while} \ \text{next} \neq n \\
\quad & p \leftarrow \text{next} \\
\quad & \text{next} \leftarrow \text{read}(p.\text{next}) \\
\textbf{write}(p.\text{next}, \ n.\text{next})
\end{align*}
\]

**Function** insertLast\( (n) \)

\[
\begin{align*}
\text{last} & \leftarrow \text{readLast}() \\
\textbf{write}(\text{last}.\text{next}, \ n)
\end{align*}
\]

Base conditions:

**Function** readLast\( () \)

\[
\begin{align*}
n & \leftarrow \bot \\
\Phi_1 & : \text{true} \\
\text{next} & \leftarrow \text{read}(\text{head}.\text{next}) \\
\textbf{while} \ \text{next} \neq \bot \\
\quad & n \leftarrow \text{next} \\
\Phi_2 & : \text{head} \Rightarrow^* n \\
\Phi_3 & : \text{head} \Rightarrow^* n \\
\textbf{return}(n)
\end{align*}
\]
Example

**Function** `remove(n)`

\begin{align*}
p & \leftarrow \bot \\
next & \leftarrow \text{read}(\text{head}.next) \\
\textbf{while} & \ nnext \neq n \\
p & \leftarrow \text{next} \\
next & \leftarrow \text{read}(p.next) \\
\text{write}(p.next, \ nnext) \\
\end{align*}

**Function** `insertLast(n)`

\begin{align*}
\text{last} & \leftarrow \text{readLast()} \\
\text{write}(\text{last}.next, \ n) \\
\end{align*}

**Base conditions:**

\begin{align*}
\Phi_1 : \text{true} \\
\Phi_2 : \text{head} & \Rightarrow n \\
\Phi_3 : \text{head} & \Rightarrow n \\
\end{align*}

**Function** `readLast()`

\begin{align*}
n & \leftarrow \bot \\
nnext & \leftarrow \text{read}(\text{head}.next) \\
\textbf{while} & \ nnext \neq \bot \\
n & \leftarrow \text{next} \\
nnext & \leftarrow \text{read}(n.next) \\
\text{return}(n) \\
\end{align*}
Example

**Function** remove(n)
\[
p \leftarrow \perp \\
next \leftarrow \text{read}(\text{head}.\text{next}) \\
\textbf{while} \; \text{next} \neq n \\
p \leftarrow \text{next} \\
next \leftarrow \text{read}(p.\text{next}) \\
\text{write}(p.\text{next}, \; n \; \text{next})
\]

**Function** insertLast(n)
\[
\text{last} \leftarrow \text{readLast}() \\
\text{write}((\text{last}.\text{next}, \; n)
\]

Base conditions:

**Function** readLast()
\[
n \leftarrow \perp \\
\text{next} \leftarrow \text{read}(\text{head}.\text{next}) \\
\textbf{while} \; \text{next} \neq \perp \\
n \leftarrow \text{next} \\
\text{next} \leftarrow \text{read}(\text{n}.\text{next})
\]

\[
\begin{align*}
\Phi_1 & : \text{true} \\
\Phi_2 & : \text{head} \Rightarrow n \\
\Phi_3 & : \text{head} \Rightarrow n
\end{align*}
\]

\[
\text{return}(n)
\]
Where is the Problem?

It covers only single-writer designs

It does not cover composable designs

Can we cover a wider set?

Optimistic Composable Data Structures
Optimistic Data Structures
Optimistic Data Structures

Concurrent Operation (add, remove, contains, ...)

Optimistic Data Structures

Concurrent Operation (add, remove, contains, ...)

Traversal (long - unmonitored)

Commit (short - monitored)
Composable Data Structures
Composable Data Structures

Atomic Block (Tx)

Traversal(Op1) Commit(op1)

Traversal(Op2) Commit(op2)
Composable Data Structures

Atomic Block (Tx)

Traversals:
- Traversal(Op1)
- Traversal(Op2)

Commitments:
- Commit(op1)
- Commit(op2)

Traversal(Tx)

Commit(Tx)
Our Models

Single Writer Commit (SWC)

Composable SWC (C-SWC)
SWC Model

\[ \text{UO}_1, \text{UO}_2, \text{UO}_3, \text{UO}_4, \text{UO}_5 \]

\[ \text{RO} \]

\[ \text{Step}_1, \text{Step}_2, \ldots, \text{Step}_n \]
SWC Model

Step 1
Step 2
Step n

RO

T_1  C_1
T_2  C_2
T_3  C_3
T_4  C_4
T_5  C_5
SWC Model
Even More…

- Do we really need single commit at a time:
  - NO!!!

- It is enough to execute commit phases atomically with single lock atomicity (SLA) guarantees.

- More practical alternatives:
  - HTM (e.g. Intel TSX).
  - STM (e.g. NOrec “the SLA version”).
Validity

- Guarantee that all $S_i$'s are sequentially reachable.
  - Comes for free in the SWMR!
  - In SWC: Prove that $S_{i-1}$ is the base point of the first step of $S_i$'s commit phase.

- Watch your step
  - In RO AND the traversal phase of all UO's
Acceptable base points for RO's return step are only $S_2$, $S_3$, $S_4$.

- Observes either the last commit or a concurrent commit.
Example

1:   procedure READLAST
2:       last ← ⊥
3:       next ← read(head.next) \( \triangleright \phi_1 : \text{true} \)
4:       while next ≠ ⊥ do
5:           last ← next
6:           next ← read(last.next) \( \triangleright \phi_2 : \text{head} \Rightarrow last \)
7:       return(last) \( \triangleright \phi_3 : \text{head} \Rightarrow last \)
8:   end procedure

9:   procedure INSERTLAST(n)
10:      last ← ⊥
11:      next ← read(head.next) \( \triangleright \phi_4 : \text{true} \)
12:      while next ≠ ⊥ do
13:          last ← next
14:          next ← read(last.next) \( \triangleright \phi_5 : \text{head} \Rightarrow last \)
15:      lockAcquire(gl) \( \triangleright \phi_6 : \text{head} \Rightarrow last \)
16:      if read(last.next) ≠ ⊥ then
17:          lockRelease(gl)
18:          go to 10
19:      write(last.next, n)
20:      lockRelease(gl)
21:   end procedure
Example

1: procedure READLAST
2:     last ← ⊥
3:     next ← read(head.next)         ▷ \phi_1 : true
4:     while next ≠ ⊥ do
5:     last ← next
6:     next ← read(last.next)        ▷ \phi_2 : head \Rightarrow^* last  
7:     return(last)                   ▷ \phi_3 : head \Rightarrow^* last
8: end procedure

9: procedure INSERTLAST(n)
10:  last ← ⊥
11:  next ← read(head.next)          ▷ \phi_4 : true
12:  while next ≠ ⊥ do
13:  last ← next
14:  next ← read(last.next)          ▷ \phi_5 : head \Rightarrow^* last
15:  lockAcquire(gl)
16:  if read(last.next) ≠ ⊥ then
17:     lockRelease(gl)
18:     go to \textcolor{red}{10}
19:  write(last.next, n)
20:  lockRelease(gl)
21: end procedure
Example

1: procedure READLAST
2:    last ← \perp
3:    next ← \textbf{read}(head.next)
4:    \textbf{while} next ≠ \perp \textbf{do}
5:    last ← next
6:    next ← \textbf{read}(last.next)
7:    return(last)
8: end procedure

9: procedure INSERTLAST(n)
10:  last ← \perp
11:  next ← \textbf{read}(head.next)
12:  \textbf{while} next ≠ \perp \textbf{do}
13:  last ← next
14:  next ← \textbf{read}(last.next)
15:  lockAcquire(gl)
16:  if \textbf{read}(last.next) ≠ \perp then
17:     lockRelease(gl)
18:     go to 10
19:  write(last.next, n)
20:  lockRelease(gl)
21: end procedure

▷ φ₁ : true
▷ φ₂ : head ⇒ last
▷ φ₃ : head ⇒* last
▷ φ₄ : true
▷ φ₅ : head ⇒ last
▷ φ₆ : head ⇒* last
Composable SWC Model (C-SWC)

1: procedure ATOMIC: T₁
2: \[ x = 5 \]
3: if readLast() ≠ x then
4: \[ insertLast(x) \]
5: if readLast() ≠ x then
6: \[ ... // illegal execution \]
7: end procedure
Composable SWC Model (C-SWC)

Atomic Block (Tx)

- Traversal(Op1)
- Commit(op1)
- Traversal(Op2)
- Commit(op2)

Traversal(Tx)

- Traversal(Op1)
- Traversal(Op2)
- Commit(op1)
- Commit(op2)

Commit(Tx)
What is remaining?

- Internal Consistency.
  - The commit phase of each operation reflects what the operation observed in its traversal.
  - The shared state of an operation is visible to subsequent operations in the same transaction.
How to prove internal consistency?

Traversal(Op1)  Traversal(Op2)  Commit(op1)  Commit(op2)
How to prove internal consistency?

Traversal(Op1) → Traversal(Op2) → Commit(op1)

Have the same base point

L L L
Related Work

- **SWMR Model (Lev-Ari et. al, DISC'14)**
  - SWC is a superset.

- **MWMR Model (Shao et. Al, SIAM'11)**
  - Lattice of consistency levels.
  - SWC corresponds to MWReg.

- **LS-Linearizability (Gramoli et. Al, PODC'12)**
  - Local serializability instead of validity.
  - Linearizability instead of regularity.
  - SWC is less conservative.
Conclusion

- SWMR model is the first step towards a general modeling of concurrent data structures, but it only covers:
  - Single writer designs.
  - Concurrent (non-composable) designs.

- SWC Model: allows multiple writers with SLA-based commit phases.

- C-SWC: extends SWC to allow operations composition.
Thanks!

Questions?