Composition vs. Concurrency

Vincent Gramoli   Rachid Guerraoui   Mihai Leția
EPFL

January 17, 2011

Sequential composition

- Code needs to ”play well“ with other code (composition)

Sequential composition

- Code needs to ”play well“ with other code (composition)
- Composition
  - we have operations O_1 and O_2
  - We create $O_3 = \{O_1(); O_2();\}$

Sequential composition

- Code needs to "play well" with other code (composition)

- Composition
  - we have operations O_1 and O_2
  - We create $O_3 = \{O_1(); O_2();\}$

  - $O_1 = \text{remove(x)}$, $O_2 = \text{insertIfNotPresent}(y, x)$

In a concurrent system

- Can multiple instances of O_3 run concurrently?
- Correctness?

In a concurrent system

- Can multiple instances of O_3 run concurrently?
- Correctness?

- O_1 and O_2 ensure atomicity and deadlock-freedom
- We want O_3 to ensure the same properties
- No modification to O_1 or O_2

Locks...

Locks...

- ...don't compose

Locks...

- ...don't compose
- take $O_3 = \{O_1(); O_2();\}$ and $O_4 = \{O_2(); O_1();\}$

Along came transactions

- very simple programming model:

```
begin-tx();
    ...
    ...
end-tx();
```

Along came transactions

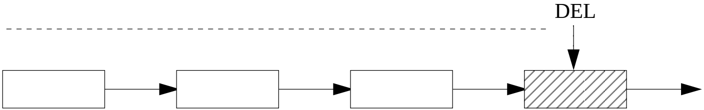
- very simple programming model:

```
begin-tx();
    ...
    ...
end-tx();
```

- guarantees atomicity and deadlock freedom

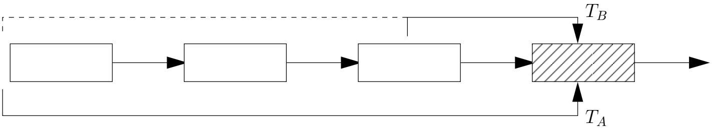
And more efficient transactions

That ignore some potential conflicts



And more efficient transactions

That ignore some potential conflicts



And more efficient transactions

- Reason at a more abstract layer

- Use an underlying thread-safe library
- Use abstract locks

Composition hierarchy

- Composition is more complex in the concurrent world
- Atomicity, deadlock freedom
- Three different levels
- Increasing strictness

Simple composition

- We have a data type with some thread-safe operation A
- We define operation B
- Both A and B are thread-safe

A || B

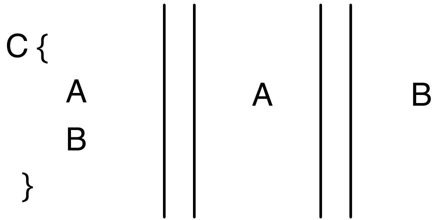
Outer composition

- Starting from the previous data type with operations A and B
- We define operation C
- C calls both A and B
- C is thread-safe w.r.t. itself

```
C {              ||        C {
      A          ||              A
      B          ||              B
}                ||        }
```

Nested composition

- Starting from the previous data type with operations A and B
- We define operation C
- C calls both A and B
- C is thread-safe w.r.t. itself and to A and B



Escaped transactions

- Opt to ignore read/write conflicts on some memory locations (possibly after a certain point in time)
- Use $O_1 = \text{contains(y)}$ and $O_2 = \text{insert(x)}$ to obtain $O_3 = \text{insertIfNotPresent}(x, y)$
- y is not in the set
- Location L_y becomes unprotected after the execution of O_1
- A different thread can insert y, breaking atomicity

Conclusions

- Composition should not be taken for granted
- Compromise between concurrency and reusability?
- In which direction do we need to go?
- Composition is important for software reuse

- Future work
  - guidelines for developing composable concurrency control