Composition vs. Concurrency

Vincent Gramoli  Rachid Guerraoui  Mihai Leția

September 16, 2010
Sequential composition

- Code reuse
- Code needs to "play well" with other code (composition)
- Composition is well understood (sequential)
- Composition
  - we have operations $\pi_1$ and $\pi_2$
  - We create $\pi_3 = \pi_1 \circ \pi_2$
  - $\pi_1 =$ remove$(x)$, $\pi_2 =$ insertIfNotPresent$(y, x)$
Can multiple instances of $\pi_3$ run concurrently?
Correctness?
$\pi_1$ and $\pi_2$ ensure atomicity and deadlock-freedom
We want $\pi_3$ to ensure the same properties
No modification to $\pi_1$ or $\pi_2$
Transactions make the developer’s life easy

- Regular transactions are easily composable
- Simply encapsulate operations inside a transaction

But

- Transactions abort over-conservatively
- Alternative transactional models go around this problem
- At what cost?
Open-nesting, transactional boosting

Transactions operate at a higher level than memory reads and writes

Conflicts are detected between higher level operations, but they need to be reverted on rollback

Some operations can easily be reverted (insert, remove)

Java: insertAll, removeAll return a boolean value
Escaped transactions

- Early release, view transactions
- Opt to ignore read/write conflicts on some memory locations (possibly after a certain point in time)
- Use $\pi_1 = \text{contains}(y)$ and $\pi_2 = \text{insert}(x)$ to obtain $\pi_3 = \text{insertIfNotPresent}(x, y)$
- $y$ is not in the set
- Location $L_y$ becomes unprotected after the execution of $\pi_1$
- A different thread can insert $y$, breaking atomicity
Conclusions

- Compromise between concurrency and reusability?
- In which direction do we need to go?