Locks Do Not Compose!

Example Code

class Account {
    float balance;
    void deposit(float amt) {
        this.lock();
        balance += amt;
        this.unlock();
    }
    void withdraw(float amt) {
        this.lock();
        balance -= amt;
        this.unlock();
    }
}

void transfer(Account from, Account to, float amt) {
    from.lock();
    from.withdraw(amt);
    to.deposit(amt);
    to.unlock(); from.unlock();
}

Thread 1

transfer(A, B, 10);
  -> call A.lock() ①
  -> call B.lock() ③

Thread 2

transfer(B, A, 10);
  -> call B.lock() ②
  -> call A.lock() ④

• Deadlock!!!
• Changing order of from.lock() and to.lock() inside transfer(...) will not solve problem
• Need elaborate scheme to ensure uniform lock ordering among Account objects for each new type of transaction
⇒ Not Composable!
Converting to a TM Paradigm

Example Code

```java
class Account {
    float balance;
    void deposit(float amt) {
        this.lock();
        balance += amt;
        this.unlock();
    }
    void withdraw(float amt) {
        this.lock();
        balance -= amt;
        this.unlock();
    }
    ...
    void transfer(Account from, Account to, float amt) {
        from.lock(); to.lock();
        from.withdraw(amt);
        to.deposit(amt);
        to.unlock(); from.unlock();
    }
}
```

TM Code

```java
class Account {
    float balance;
    void deposit(float amt) {
        begin_atomic;
        balance += amt;
        end_atomic;
    }
    void withdraw(float amt) {
        begin_atomic;
        balance -= amt;
        end_atomic;
    }
    ...
    void transfer(Account from, Account to, float amt) {
        begin_atomic;
        from.withdraw(amt);
        to.deposit(amt);
        end_atomic;
    }
}
```

⇒ Now no Deadlocks! (Roll back on atomicity violation)
Open vs. Closed Nesting

Example Code

class List {
    void insert(Object o) {
        begin_atomic;
        ...
        end_atomic;
    }
    ...
}    
void foo(List list, Object o) {
    ...
    list.insert(o);
    ...
}
void bar(List list, Object x, Object y) {
    /* nested transaction */
    begin_atomic;
    ...
    list.insert(x);
    list.insert(y);
    ...
    end_atomic;
}

Thread 1

foo(list, x, y);
- call list.insert(x);  
- call list.insert(y);  

Thread 2

bar(list, z);
- call list.insert(z)  

- Can list ordering x, z, y happen?
  - No: Closed Nesting
  - Yes: Open Nesting

- Closed Nesting
  - Inner transaction state merged to outer transaction on commit
  - Preserves atomicity of outer transaction
  - Most intuitive to programmers

- Open Nesting
  - Inner transaction state merged to main memory on commit
  - Breaks atomicity of outer transaction
  - Needs compensating code
  - Allows interleaving w/o squashing (e.g. malloc)