Transactions with Isolation and Cooperation (Smaragdakis et al., 2007)

Presented by Scott Fleming

A Paper Presentation for
CSE 891 - Formal Methods in Software Development: Reliable Computing With Threads
Fall Semester, 2008

Motivation for Transactional Programming

Problem: Complexity and error proneness in conventional concurrent programming models
- Non-local reasoning about lock acquisition order and condition signaling

Solution: Transactional programming
- Single, local design decision about which actions execute atomically

Problems with Prior TM Solutions

Problem: Thread cooperation incompatible with nesting
- Threads may need to expose results mid-transaction

Problem: Rollback incompatible with irreversible operations

Solution: Transactions with Isolation and Cooperation (TIC)
- Programming model
- Extends to atomic blocks (closed-nesting semantics)
- Implementation independent (e.g., pessimistic/optimistic concurrency control)

Example of Thread Cooperation

Barrier synchronization: Group of threads wait at barrier until all have arrived, then proceed
- Threads cooperate by exposing when they are at the barrier

```c
void barrier()
{
    atomic { ++count; }
    atomic(count == NUMTHREADS) {
        /* barrier reached */
    }
}
```
### Example of Cooperation Problem

How should outermost transaction behave?
- Evaluate nested guards at outermost level?
- Restart outermost transaction when unsatisfied inner guard reached?

Essential problem: Barrier needs to break isolation and atomicity

### Addressing Thread Cooperation: **Wait**

**Syntax:**

```c
atomic {
    statementBlockA
    wait(condition);
    statementBlockB
}
```

**Semantics:**
- If `condition` is satisfied, nothing happens
- Else, `suspend` until `condition` is satisfied, then `resume`:
  1. Current transaction commits before `wait` (top transaction)
  2. Thread blocks until `condition` is satisfied
  3. Everything after `wait` executes as new transaction (bottom transaction)

### Example: Barrier with Wait

```c
void barrier() {
    atomic {
        ++count;
        atomic(count == NUMTHREADS) {
            /* barrier reached */
        }
    }
}
```
Addressing Breach of Isolation: \texttt{expose}

Idea: Make consequences of waiting evident to programmer

Syntax:

\begin{verbatim}
expose \{expression\} \{establish \ statement\}
\end{verbatim}

- Call to \texttt{waiting method} is only legal in \texttt{expose} expression

Semantics:
- If waiting method in \texttt{condition} suspends, \texttt{statement} is executed when waiting method returns (having already resumed)
- \texttt{Statement} executes in bottom transaction abort/retries

Two Examples: Expose/Establish

\begin{verbatim}
atomic {
  beforeBarrier
  expose(barrier());
  afterBarrier
}

void foo()
{
  atomic {
    Wait(x > 0);
    --x;
  }
}

void bar()
{
  atomic {
    y = 0;
    expose \{foo()\}
    establish \{ y = 1; \};
  afterFoo
  }
}
\end{verbatim}
Programming Obligations: **expose**

Expression

Prior to waiting-method call: Establish global invariants
- Purpose: Relax atomicity
- Rationale: Other threads may observe intermediate results
**establish** clause (and code following it): Re-establish local invariants
- Purpose: Relax isolation
- Rationale: Other threads may have observable effects

```java
Establish global invariants
expose (waiting-method call)
establish {
  Re-establish local invariants
};
Re-establish local invariants
```

Two Types of Suspending Operations

Suspending operation: Any non-waiting method that violates atomicity or isolation
- Typically **external operations**

**Irreversible** suspending operation: Cannot be undone
- E.g.: Output to console

**Reversible** suspending operation: Can be undone
- E.g.: Memory allocation (undo by deallocating)

---

Addressing Irreversible Operations: **suspending**

Syntax: Annotation to root **suspending-method** declaration; e.g.:
```
public suspending native void write(int b) throws IOException;
```

- Suspending methods must be called in **expose** expression

Semantics: Same as waiting method where root suspending operation is like **Wait**, except
- On retry, root suspending method is not repeated
- Instead, execution resumes from return point of root suspending method call
- Hence, corresponding **establish** clause is executed first

---

Example: Irreversible Operation

```java
double bal;
double bet;
atomic
while (true) {
  bal = balance();
  expose(print("Your balance is "+ bal));
  bet = expose(input("How much will you wager?"))
  establish {
    if (bal == balance()) break;
  };
  if (bet <= balance) register(bet);
}
```
Addressing Reversible Operations: undo and oncommit

Idea: Treat as in open-nested transactions
- Reversible operation gets “undo” and “on-commit” operations

Syntax: Annotation to suspending-method declaration; e.g.:

```
undo(release)
Entity allocate(int length);
void release(Entity e, int length);
```

- May be called anywhere (i.e., outside of expose expression)

Semantics: Every method with undo annotation causes outermost transaction it contains to have open-nesting semantics, relative to method’s enclosing transactional context

Example: Reversible Operation

```
undo(releaseIntArray)
int[] allocateIntArray(int length);
void releaseIntArray(int[] a, int l);
```

```
atomic {
    int[] acopy;
    int size = sharedArray.length;
    acopy = allocateIntArray(size);
    for (int i = 0; i < size; ++i) {
        acopy[i] = sharedArray[i];
    }
}
Problem in the Programming Model

Consider this example where \( n \) is shared:

```java
n = 0;
atomic {
    n = 1;
    openatomic {
        ++n;
    } undo { --n; }
}...
```

Problem: Undo action is incorrect
- Increment inadvertently exposes effects of parent transaction

Solution to Problem: Weird Semantics

Semantics: Uncommitted shared memory transactions can not observe uncommitted parent effects to shared memory

Comment: This is nuts

To avoid weird semantics: Open-nested transaction should never perform a write that (control- or data-) depends on shared data written by its parent

Question: How easy is this in practice?

Evaluation

Criteria: Applicability of TIC
Evaluation: Reimplemented several programs/libraries in TIC

Criteria: Performance
Evaluation: Limited testing

Criteria: Usability of TIC
No evaluation

Criteria: Maintainability of TIC-based software
No evaluation