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 */
    }
}
```
Scenario: Barrier for Three Threads
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: \texttt{Wait}

Syntax:

\begin{verbatim}
atomic {
  statementBlockA
  \texttt{Wait(condition)};
  statementBlockB
}
\end{verbatim}

Semantics:

- If \textit{condition} is satisfied, nothing happens
- Else, \textit{suspend} until \textit{condition} is satisfied, then \textit{resume}:
  \begin{enumerate}
  \item Current transaction commits before \texttt{Wait(}\textit{top transaction}\texttt{)}
  \item Thread blocks until \textit{condition} is satisfied
  \item Everything after \texttt{Wait} executes as new transaction (\textit{bottom transaction})
  \end{enumerate}
Example: Barrier with Wait

```c
void barrier() {
    atomic {
        ++count
        Wait(count == NUMTHREADS);
    }
}
```

```c
atomic {
    beforeBarrier
    barrier();
    afterBarrier
}
```
Scenario: Barrier w/ Wait

T1
- beforeBarrier ++count;
- beforeBarrier ++count;
- afterBarrier

T2
- beforeBarrier ++count;
- afterBarrier

T3
- beforeBarrier ++count;
- afterBarrier

T1
- afterBarrier

T2
- afterBarrier

T3
- afterBarrier

Scenario: Barrier w/ Wait
Addressing Breach of Isolation: \texttt{expose}

Idea: Make consequences of waiting evident to programmer

Syntax:

\begin{itemize}
  \item \texttt{expose \{expression\} \{establish \ statement\}}
\end{itemize}

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

Semantics:

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

```
atomic {
    beforeBarrier
    expose(barrier());
    afterBarrier
}

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

void bar()
{
    atomic {
        y = 0;
        expose (foo())
        establish { y = 1; }
    }
    afterFoo
}
```
Scenario: Foo/Bar

T1
y = 0;
--x;
*afterFoo*

T2
x = 0;
y = 42;

T1
y = 0;

T2
x = 70;

T1
x = 0;
y = 0;

T2
x = 0;
y = 42;

T1
--x;
y = 1;
*afterFoo*

T2
x = 70;

Scenario: Foo/Bar
Programming Obligations: \texttt{expose} Expression

Prior to waiting-method call: Establish global invariants
  \begin{itemize}
    \item Purpose: Relax atomicity
    \item Rationale: Other threads may observe intermediate results
  \end{itemize}

\texttt{establish} clause (and code following it): Re-establish local invariants
  \begin{itemize}
    \item Purpose: Relax isolation
    \item Rationale: Other threads may have observable effects
  \end{itemize}

\begin{lstlisting}
Establish global invariants
expose (waiting-method call)
establish {
  Re-establish local invariants
};
Re-establish local invariants
\end{lstlisting}
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.:

```java
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

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);
balance
10.00

Scenario: No Contention

T1
bal = 0.0
bet = 0.0

bal = 10.00
bal = balance();

print(...);

bet = 5.00
bet = input(...);

break;
register(bet);

T2

T1
Scenario: Contention

T1

bal = 0.0
bet = 0.0

bal = 10.00
bal = balance();
print(... + bal);

T2

bet = input(...);
withdraw(7.00);

T1

bet = 5.00
bet = input(...);

bal = 3.00
bal = balance();
print(... + bal);

bal = 3.00
bal = balance();
print(... + bal);
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.:

```java
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

```java
atomic {
    int[] acopy;
    int size = sharedArray.length;

    acopy = allocateIntArray(size);

    for (int i = 0; i < size; ++i) {
        acopy[i] = sharedArray[i];
    }
}
```

```java
undo(releaseIntArray)
int[] allocateIntArray(int length);
void releaseIntArray(int[] a, int l);
```
Scenario: Undoing

T1

size = 3

sharedArray

(8, 9, 1)

T2

size = 3

sharedArray.length;

... new int[...];

releaseIntArray(...);

size = 3

sharedArray.length;

... new int[...];

Scenario: Undoing
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