Language Extensions for Open Nested Transactions

Tony Hosking, Purdue University
Eliot Moss, University of Massachusetts

PURDUE
UNIVERSITY

Transactions are nice

- atomicity: undo-retry avoids lock anomalies

  (deadlock, wrong lock, priority inversion)

- consistency: preserves invariants

  (pre/post-conditions)

- isolation: handles conflicts with undo-retry

  (exceptions, errors)

Serializability in face of concurrent execution and txn failures

PURDUE UNIVERSITY

Limitations

- Isolation ⇒ no communication
- Long/large txns either reduce concurrency or are unlikely to commit
- Data structures often have false conflicts
  - e.g., reorganizing B-tree nodes
- Can't do conditional critical regions:
  - e.g., insert in buffer if/when room, etc.
- Do not themselves provide concurrency

PURDUE UNIVERSITY

Closed nesting helps

- CCRs: `atomic (P) {S}`
- alternatives: `atomic {S1} else {S2}`
- concurrency: concurrent sub-txns can fail and retry independently

PURDUE UNIVERSITY

Limitations of closed nesting

Derive from original non-nested semantics:

- Aggregates larger and larger conflict sets
  - Still hard to complete long/large txns
- Synchronizes at physical level
  - Gives false conflicts
- Isolation still strict
  - No communication, so fails to address a whole class of concurrent systems

PURDUE UNIVERSITY

Open nesting to the rescue

- Closed nesting has just one level of abstraction: memory contents
  - Basis for concurrency control
  - Basis for rollback
- Open nesting has more levels of abstraction
- Each level may have a distinct:
  - concurrency control model (style of locks)
  - recovery model (operations for undoing)

PURDUE UNIVERSITY

Open nested actions

- While running, an open nested action
  - operates at the memory level, and
  - may accumulate locks and undos from committed children
- When it commits:
  - Its memory changes are permanent
  - Concurrency control and recovery switch levels
    - Give up memory level "locks" and child locks
      - acquire abstract locks for new level
    - Give up memory level undo and child undos
      - undo with inverse operation for new level

PURDUE UNIVERSITY

Abstract serializability

- Lock parts of abstract state
- Undo in the abstract

  Result is abstract serializability
- Undo restores changed part of abstract state
- Lock must prevent conflicting forward operations
- Lock must ensure undo remains applicable

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
    s.add(x);
    s.add(y);
}
```

T_2:

```
atomic {
    s.add(z);
}
```

```
s: (      )
```

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
      s.add(x);
      s.add(y);
}
```

T_2:

```
atomic {
      s.add(z);
}
```

```
s: (x,y  )
```

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
      s.add(x);
      s.add(y);
}
```

T_2:

```
atomic {
      s.add(z);
}
```

```
s: (x,y,z)
```

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
    s.add(x);
    s.add(y);
}
```

T_2:

```
atomic {
    s.add(z);
}
```

```
s: (      )
```

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
      s.add(x);
      s.add(y);
}
```

T_2:

```
atomic {
      s.add(z);
}
```

```
s: (z     )
```

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
      s.add(x);
      s.add(y);
}
```

T_2:

```
atomic {
      s.add(z);
}
```

```
s: (z,x,y)
```

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
    s.add(x);
    s.add(y);
}
```

T_2:

```
atomic {
    s.add(z);
}
```

```
s: (      )
```

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
    s.add(x);
    s.add(y);
}
```

T_2:

```
atomic {
    s.add(z);
}
```

```
s: (x    )
```

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
    s.add(x);
    s.add(y);
}
```

T_2:

```
atomic {
    s.add(z);
}
```

```
s: (x,z  )
```

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
    s.add(x);
    s.add(y);
}
```

T_2:

```
atomic {
    s.add(z);
}
```

s: (x, ~~y~~ z)

PURDUE UNIVERSITY

Physical serializability

```
Set s = new LinkedHashSet();
```

T_1:

```
atomic {
    s.add(x);
    s.add(y);
}
```

T_2:

```
atomic {
    s.add(z);
}
```

s: (x, ~~z~~ y)

Abstract serializability

- Physical conflicts may be an artefact of the representation
- `(x,y,z)`, `(z,x,y)`, `(x,z,y)` are all valid representations of the unordered set `{x,y,z}`
- In terms of `s.contains`, they are the same

Abstract serializability

```
                    Set s;

T1:                          T2:
  atomic {                     atomic {
        s.add(x);                    s.add(z);
        s.add(y);                  }
  }
```

s: { }

PURDUE UNIVERSITY

Abstract serializability

```
                    Set s;

T1:                          T2:
  atomic {                     atomic {
        s.add(x);                    s.add(z);
        s.add(y);                  }
  }
```

s: {x }

PURDUE UNIVERSITY

Abstract serializability

```
                    Set s;

T1:                          T2:
  atomic {                     atomic {
        s.add(x);                    s.add(z);
        s.add(y);                  }
  }
```

s: {x, z}

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
      s.add(x);
      s.add(y);
}
```

T_2:

```
atomic {
      s.add(z);
}
```

```
s: {x,y,z}
```

PURDUE UNIVERSITY

Abstract serializability

- Still need conflict detection and resolution
- Must ensure that individual set insertions still happen atomically
- Result `(x,z)` is neither physically nor abstractly serializable
- Must ensure abstract conflicts do not occur

PURDUE UNIVERSITY

Abstract serializability

```
                                Set s;
```

T_1:

```
   atomic {
      s.add(x);
      if (!s.contains(z))
         s.add(y);
   }
```

T_2:

```
   atomic {
      s.add(w);
      if (!s.contains(y))
         s.add(z);
   }
```

```
                          s: {        }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {  x    }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {  x    }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {  x,y  }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x,y  }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x,y  }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x,y  }
```

PURDUE UNIVERSITY

Abstract serializability

```
                    Set s;

T1:                          T2:
   atomic {                     atomic {
      s.add(x);                    s.add(w);
      if (!s.contains(z))          if (!s.contains(y))
         s.add(y);                    s.add(z);
   }                            }


                    s: {         }
```

Abstract serializability

```
Set s;

T1:                              T2:
   atomic {                         atomic {
      s.add(x);                        s.add(w);
      if (!s.contains(z))              if (!s.contains(y))
         s.add(y);                        s.add(z);
   }                                }
```

s: { x }

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x    }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x    }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x,  z}
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x,  z}
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x,  z}
```

PURDUE UNIVERSITY

Abstract serializability

```
                    Set s;

T1:                          T2:
   atomic {                     atomic {
      s.add(x);                    s.add(w);
      if (!s.contains(z))          if (!s.contains(y))
         s.add(y);                    s.add(z);
   }                            }


                    s: {         }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {  x    }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x    }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x    }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

```
s: {w,x    }
```

PURDUE UNIVERSITY

Abstract serializability

```
                                Set s;
```

T_1:

```
   atomic {
      s.add(x);
      if (!s.contains(z))
         s.add(y);
   }
```

T_2:

```
   atomic {
      s.add(w);
      if (!s.contains(y))
         s.add(z);
   }
```

```
                          s: {w,x,y  }
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

PURDUE UNIVERSITY

Abstract serializability

```
Set s;
```

T_1:

```
atomic {
   s.add(x);
   if (!s.contains(z))
      s.add(y);
}
```

T_2:

```
atomic {
   s.add(w);
   if (!s.contains(y))
      s.add(z);
}
```

T_1 must lock abstract absence of z so T_2 waits

or

T_2 must lock abstract absence of y so T_1 waits

Open nesting and data abstraction

Open nesting best fits ADTs, not code chunks

- For safety, memory state accessed by an open nested action generally must not be accessed by closed actions
- Abstract data types neatly encapsulate state
- ADTs also tend to provide inverses
- Abstract locks match abstract state/operations

PURDUE UNIVERSITY

How ADTs fit open nesting

Data type works correctly as a whole:

- Protected concrete state
- Clearly understood abstract state
- Abstract locks, in terms of abstract state
- Abstract undos, in terms of abstract operations

Example: OpenMap wrapper

```
import java.util.*;
import java.openatomic.SXModes.*;
public openatomic class OpenMap implements Map {
  private final Map map;
  private final Object lock = new Object();
  public OpenMap(Map map) { this.map = map; }
  public Object get(Object key)
    locks(key:S) { return map.get(key); }
  public Object put(Object key, Object value)
    locks(key:X, lock:IX) { return map.put(key, value); }
    onabort {
      if (@result == null) map.remove(key);
      else map.put(key, @result);
    }
  public Object remove (Object key)
    locks(key:X, lock:IX) { return map.remove(key); }
    onabort { if (@result != null) map.put(key, @result); }
  public int size() locks(lock:S) { return map.size(); }
}
```

Example: OpenMap wrapper

```
import java.util.*;
import java.openatomic.SXModes.*;
public openatomic class OpenMap implements Map {
  private final Map map;
  private final Object lock = new Object();
  public OpenMap(Map map) { this
  public Object get(Object key)
    locks(key:S) { return map.get
  public Object put(Object key,
    locks(key:X, lock:IX) { retu
    onabort {
      if (@result == null) map.remove(key);
      else map.put(key, @result);
    }
  public Object remove (Object key)
    locks(key:X, lock:IX) { return map.remove(key); }
    onabort { if (@result != null) map.put(key, @result); }
  public int size() locks(lock:S) { return map.size(); }
}
```

3 lock modes:

- Shared
- eXclusive
- Intension eXclusive

PURDUE UNIVERSITY

Example: OpenMap wrapper

```
import java.util.*;
import java.openatomic.SXModes.*;
public openatomic class OpenMap implements Map {
  private final Map map;
  private final Object lock = new
  public OpenMap(Map map) { this.
  public Object get(Object key)
    locks(key:S) { return map.get(key); }
  public Object put(Object key, Object value)
    locks(key:X, lock:IX) { return map.put(key, value); }
    onabort {
      if (@result == null) map.remove(key);
      else map.put(key, @result);
    }
  public Object remove (Object key)
    locks(key:X, lock:IX) { return map.remove(key); }
    onabort { if (@result != null) map.put(key, @result); }
  public int size() locks(lock:S) { return map.size(); }
}
```

Abort handlers restore abstract state

Example: OpenMap wrapper

```
import java.util.*;
import java.openatomic.SXModes.*;
public openatomic class OpenMap implements Map {
  private final Map map;
  private final Object lock = new Object();
  public OpenMap(Map map) { this.map = map; }
  public Object get(Object key)
    locks(key:S) { return map.get(key); }
  public Object put(Object key, Object value)
    locks(key:X, lock:IX) { return map.put(key, value); }
    onabort {
      if (@result == null) map.remove(key);
      else map.put(key, @result);
    }
  public Object remove (Object key)
    locks(key:X, lock:IX) { return map.remove(key); }
    onabort { if (@result != null) map.put(key, @result); }
  public int size() locks(lock:S) { return map.size(); }
}
```

Log semantics:closed nesting

- Each new transaction starts with a clean log
  - Log reads/writes + old values for writes
- Closed nested commit appends log to parent
- Top-level commit discards log
- Abort traverses log in reverse, restores writes

Log semantics: conflicts

- Neither action an ancestor of the other and both access a location in conflicting modes (i.e., at least one writes)
- Conflicting actions must not both commit and must never write same locations

Log semantics: open nesting

- Add handlers and abstract locks to logs
- Nested commit appends own handlers/locks to parent's log
- Nested abort processes log in reverse:
  - invoke `onabort` handlers (from children)
  - undo writes
    - each location may be undone more than once to preserve view for interleaved handlers

Log semantics: abstract locks

- Lock consists of:
  - context instance (or class) whose method was run as openatomic
  - locked object mentioned in lock expression
  - lock mode
- Conflict: holder not an ancestor of acquirer
  - context ==, locked object `.equals`, modes conflict

PURDUE UNIVERSITY

Bending the rules

- "Improper" abstract locking offers controlled communication
  - e.g., can probably simulate wait/notify
- "Improper" undos permit truly permanent effects
  - record attempt to use a stolen credit card
  - roll back rest of transaction
- A general loophole: handy, but admittedly a dangerous "power tool"

PURDUE UNIVERSITY

Caveat Emptor

- Programmer errors bite!
  - programmer must specify conflicts and compensations
  - open nested aborts can deadlock
- Most problems arise because of mistakes in abstraction layering: accessing same state at different abstraction levels
- Avoiding mistakes can be achieved by observing guidelines that we hope to formalize (see PPoPP paper)

PURDUE UNIVERSITY

Conclusions

- Nesting is desirable, open nesting needed
- Open nesting probably for the experts
- Need to integrate:
  - Desired semantics
  - Language design (with exceptions, etc.)
  - Run-time support
  - Memory-level semantics
  - Hardware implementation

PURDUE UNIVERSITY

Who should use this?

- Single-level and closed nesting usually enough
  - Most programmers don't need open nesting
- Open nesting good for library classes
  - High concurrency, or special semantics
- Undos are usually trivial to provide
  - assuming lock release is implied
- Other clauses not often necessary
  - assuming lock release is implied
- Abstract locking takes getting used to
  - Good for libraries

PURDUE UNIVERSITY