Vertical Composition of Reversible Atomic Objects

Timos Antonopoulos       Paul Gazzillo       Eric Koskinen       Zhong Shao

Yale University

NJ Programming Languages and Systems Seminar
Rutgers University
30 September 2016
java.util.concurrent.ConcurrentHashMap
Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and documents a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.3.2 (Programming Techniques): Concurrent Programming
D.2.2 [Software Engineering]: Requirements/Specifications; D.3.3 [Programming Languages]: Language Constructs abstract data types, concurrent programming structures, data types and structures; F.1.3 (Computation by Abstract Devices): Models of Computation—parallelism; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—pre and post-condition, verification techniques
java.util.concurrent.ConcurrentHashMap

cht.get(foo)
cht.put(bar,22)
Atomically move the value of `foo` to instead be associate with `bar`.

```java
cht.move(foo, bar)
```

```
cht.get(foo)
cht.put(bar, 22)
```
Atomically move the value of `foo` to instead be associate with `bar`.

```
cht.move(foo, bar)
```

**MovableHashtable**

```java
atomic {
    k = cht.get(foo)
    cht.put(bar, k)
}
```

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>42</td>
</tr>
<tr>
<td>bar</td>
<td>18</td>
</tr>
<tr>
<td>zip</td>
<td>99</td>
</tr>
<tr>
<td>jaz</td>
<td>71</td>
</tr>
</tbody>
</table>
**FileSystem**

```java
atomic {
    ...
    movHT.move("/tmp","/dead")
}
```

**MovableHashtable**

```java
atomic {
    k = cht.get(foo)
    cht.put(bar,k)
}
```
**WebService**

```
atomic {
    ...
    FS.move("/tmp","/dead")
    ...
}
```

**FileSystem**

```
atomic {
    ...
    movHTT.move("/tmp","/dead")
}
```

**MovableHashtable**

```
atomic {
    k = cht.get(foo)
    cht.put(bar, k)
}
```
Is there a theory or methodology that permits vertical composition of transactions?

Not really.
Wait, what about nested transactions?
Wait, what about nested transactions?

Nested transactions are a **different** kind of vertical composition.
Wait, what about nested transactions?

Nested transactions are a different kind of vertical composition.

```cpp
{x=3; something();
 { x=a; foo(); bar(); ...
   {x=2;} {y=9;}
}
x y z ...
```

Each has (sequential) thread-local state

Isolated concurrent (base) objects

**WebService**

**FileSystem**

**MovableHashtable**

conc HT  conc Q
How do we make this work?

What is the programming model?
Reversible Atomic Objects: A New Order.

1. Align transactions with method boundary.
3. Invocations of constituent operations.
4. Assembling the inverse.
5. The commit-and-return statement.
Reversible Atomic Objects: A New Order.

1. Align transactions with method boundary.
3. Invocations of constituent operations.
4. Assembling the inverse.
5. The commit-and-return statement.

**Consequences:**

Efficient Implementation (comparable with Boosting)
Contextual Refinement Theorem (serializability)
Vertical Composition Theorem
Transactional version of Universal Construction
Reversible Atomic Objects: A New Order.

1. Align transactions with method boundary.
3. Invocations of constituent operations.
4. Assembling the inverse.
5. The commit-and-return statement.

**Consequences**

Efficiency:

\[ [P \oplus CO]_{interleaved} \sqsubseteq [P \oplus SO]_{interleaved} \]

Contextual Refinement Theorem (serializability)

Vertical Composition Theorem

Transactional version:

\[ [CO \oplus CQ]_{interleaved} \sqsubseteq [SO \oplus SQ]_{interleaved} \]
Reversible Atomic Objects: A New Order.

```java
1 class MoveableHashtable[K, V] : RAO {
2
3
4     get(k) =
5
6     }
7
8
9
10    move(k₁, k₂) =
11
12
13
14
15    }
16 }
```
Reversible Atomic Objects: A New Order.

1. Align transactions with method boundary.

```java
class MoveableHashtable[K, V] : RAO {
    get(k) = atomic{
        cmt_return(v)
    }
    move(k1, k2) = atomic{
        cmt_return
    }
}
```
Reversible Atomic Objects: A New Order.


```java
1 class MoveableHashtable[K, V] : RAO {

2

3 get(k) = atomic{
4     /* conflict: k */
5     cmt_return (v)
6 }
7
8 /* conflict: k_1, k_2, size */
9 move(k_1, k_2) = atomic{
10        cmt_return
11 }
12 }
13
14
15 }}
16 }
```

Spec indicates circumstances under which the method `commutes` with all other operations. (See H&K PPoPP’08, DRVK PLDI’14, etc.)
Reversible Atomic Objects: A New Order.

3. Invocations of constituent operations.

```java
class MoveableHashtable[K, V] : RAO {
    Hastable[K, V] ht

    get(k) = atomic{
        /* conflict: k */
        v := ht.get(k)
        cmt_return ( v )
    }

    /* conflict: k1, k2, size */
    move(k1, k2) = atomic{
        v := ht.get(k1)
        v_old := ht.put(k2, v)
        ht.remove(k1)
        cmt_return
    }
}
```
4. Assembling the inverse.

Every object method must assemble its own inverse.
Reversible Atomic Objects: A New Order.

4. Assembling the inverse.

```java
class MoveableHashtable<K, V> : RAO {
    Hastable[K, V] ht

    ... get(k) = atomic{
        /* conflict: k */
        v := ht.get(k)
        cmt_return (skip, v)
    }

    /* conflict: k1, k2, size */
    move(k1, k2) = atomic{
        v := ht.get(k1)
        vold := ht.put(k2, v)
        ht.remove(k1)
        cmt_return (atomic{}, ⊥)
    }
}
```
Reversible Atomic Objects: A New Order.

Assembling the inverse.

This is another reversible atomic object

Already provides its own inverse

Assemble them together

```
mov\(k_1, k_2\) =
  v := ht.get(k_1) \rightarrow inv_1
  v_{old} := ht.put(k_2, v) \rightarrow inv_2
  ht.remove(k_1) \rightarrow inv_3
  cmt\_return (atomic\{\{inv_3; inv_2; inv_1\}\}, \bot)
```
Reversible Atomic Objects: A New Order.

5. The commit-and-return statement.

```java
1 class MoveableHashtable[K, V] : RAO {
2     HasTable[K, V] ht
3     ...
4     get(k) = atomic{{
5         // conflict: k */
6         l0: v := ht.get(k) \(\rightarrow\) skip
7         cmt_return (skip, v)
8     }
9     /* conflict: k₁, k₂, size */
10    move(k₁, k₂) = atomic{
11        l0: v := ht.get(k₁) \(\rightarrow\) inv₁
12        l₁: v_old := ht.put(k₂, v) \(\rightarrow\) inv₂
13        l₂: ht.remove(k₁) \(\rightarrow\) inv₃
14        cmt_return (atomic{{{inv₃; inv₂; inv₁}}, \bot})
15     }
16 }
```

Commit the txn, return inverse and value
class FileSystem[P, V] : RAO {
  MoveableHashTable[P, V] mht
  DirectoryTree[P] tree
...
  moveFile(p₁, p₂) = opt_atomic{ /* conflict: p₁, p₂,
    lca(p₁, p₂) */
    l₀: v = mht.get(p₁) ⇔ inv₀
    if (v is empty) {
      cmt_return (inv₀, false)
    } else {
      l₁: mht.move(p₁, p₂) ⇔ inv₁
      l₂: tree.moveNode(p₁, p₂) ⇔ inv₂
      cmt_return (opt.atomic{inv₂; inv₁; inv₀}, true)
    }
  }
}

class HashTable[K, V] : RAO {
  ConcurrentHashTable[K, V] cht
  get(k) { /* conflict: k */
    x = cht.get(k)
    cmt_return (skip, x)
  }
  put(k, v) { /* conflict: k, sz */
    v_old = cht.get(k)
    cht.put(k, v)
    cmt_return (cht.put(k, v_old), v_old)
  }
  remove(k) { /* conflict: k, sz */
    v_old := cht.get(k)
    cht.remove(k)
    cmt_return (cht.put(k, v_old), v_old)
  }
}
The Base Case Building Blocks: Atomic Objects

- Lift it to a *reversible* atomic object
- Simply make a wrapper
- Create inverses
- These inverses used above
- No transactions executed

```java
1 class HasTable[K, V] : RAO {
2     ConcurrentHasTable[K, V] cht
3     get(k) { /* conflict: k */
4         x = cht.get(k)
5         cmt_return (skip, x)
6     }
7     put(k, v) { /* conflict: k, sz */
8         v_old = cht.get(k)
9         cht.put(k, v)
10        cmt_return (cht.put(k, v_old), v_old)
11     }
12     remove(k) { /* conflict: k, sz */
13         v_old := cht.get(k)
14         cht.remove(k)
15        cmt_return (cht.put(k, v_old), v_old)
16     }
17 }
```

Save the old value
Per-layer transactional machinery and Conflict Detection

- **Synchronization** - Flexibility in each layer
  
  \[
  \text{opt}\text{\_atomic} \{\ldots\} \quad \text{pess}\text{\_atomic} \{\ldots\}
  \]

- **Conflict Mitigation** - Formalized contention manager as an environment that:

  - Schedules the transactions
  - Detects deadlocks
  - Can partially abort operations by invoking constituent inverses on behalf of the transaction
Reversible Atomic Objects: A New Order.

1. Align transactions with method boundary.
3. Invocations of constituent operations.
4. Assembling the inverse.
5. The commit-and-return statement.

**Consequences:**
- Efficient Implementation (comparable with Boosting)
- Contextual Refinement Theorem (serializability)
- Vertical Composition Theorem
- Transactional version of Universal Construction
Formal Treatment
Formal Treatment

• At a given layer
• Constituent base objects
• Upper layer only has **observations** of lower layer
• Inverses: \( \forall \sigma, \ O.f^{-1} (\bar{y}) (O.f (\bar{x}) (\sigma)) = \sigma \)

Now let’s add concurrency…
Formal Treatment

Compositional Semantics

Log is a communication medium.

[Push/Pull Model, KP PLDI’2015]

Threads / Environment take turns appending to the log.

Log is a communication medium.

[Push/Pull Model, KP PLDI’2015]

\[ Ev ::= \begin{align*}
(\tau, \text{lvk } O.f(\bar{x})) \\
(\tau, a) \\
(\tau, a^{-1}) \\
(\tau, \text{CmtRet } O.f(\bar{y})) \\
(\tau, \text{Term}) \\
(\tau, \triangledown)
\end{align*} \]

Invoke an abstract method
Implementation base operation
Cancel a base operation
Commit and establish inverse
Thread termination
Yield to another thread

Threads / Environment

Environment can invert a thread's operation

Conflict • Commutativity • Log-based Obs. Equivalence

\[
\begin{align*}
\text{Semantics} & \quad \frac{\ell, \tau, (T, tm) \xrightarrow{e} \ell \cdot \tau, (T, tm)}{\text{ENV}} \\
\tau \in T & \quad tm \tau = (st, c, r) \quad st, c \xrightarrow{\ell} st', c', \ell' \cdot e \quad e \in \{\triangledown, \text{Term}\} \quad \text{THR}
\end{align*}
\]
Formal Treatment

Contextual Refinement

For every (multi-threaded) program $P$, execution of $P$ composed with the objects’ implementations is a contextual refinement of $P$ composed with the objects’ corresponding atomic specification.

\[
[P \oplus CO]_{\text{interleaved}} \subseteq [P \oplus SO]_{\text{interleaved}}
\]
Formal Treatment

Vertical Composition

Let $O$ and $Q$ be two objects, then

$$\left[C_O \oplus C_Q\right]_{\text{interleaved}} \subseteq \left[S_O \oplus S_Q\right]_{\text{interleaved}}$$
<table>
<thead>
<tr>
<th>Implementation Style</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimistic STM</td>
<td>TL2 [12], TinySTM [14], McRT [47]</td>
</tr>
<tr>
<td>Checkpoints</td>
<td>Herlihy &amp; Koskinen [26]</td>
</tr>
<tr>
<td>Closed nested</td>
<td>LogTM [39]</td>
</tr>
<tr>
<td>Pessimistic STM</td>
<td>Matveev and Shavit [37]</td>
</tr>
<tr>
<td>Pessimistic objects</td>
<td>Boosting [19]</td>
</tr>
<tr>
<td>Irrevocable transactions</td>
<td>[58]</td>
</tr>
<tr>
<td>Non-opaque</td>
<td>Early release [21], dependent [45]</td>
</tr>
<tr>
<td>Nondetermin. choice</td>
<td>HaskellSTM retry/orElse [17]</td>
</tr>
<tr>
<td>Hardware TM</td>
<td>Intel [24]</td>
</tr>
</tbody>
</table>
match try_append(tid, op) with
  case Success => ...
  case Conflict => ...
  case Aborted => ...

- Begin events
- Base op events
- Commit events
match try_append(tid, op) with
    case Success => ...
    case Conflict => ...
    case Aborted => ...

- Begin events
- Base op events
- Commit events
Universal Construction

Reversible Atomic Objects Implementation
[Paul Gazzillo (Yale)]
Reversible Atomic Objects: A New Order.

1. Align transactions with method boundary.
3. Invocations of constituent operations.
4. Assembling the inverse.
5. The commit-and-return statement.

**Consequences:**
Efficient Implementation (comparable with Boosting)
Contextual Refinement Theorem (serializability)
Vertical Composition Theorem
Transactional version of Universal Construction
Thank you!

Vertical Composition of Reversible Atomic Objects

Ioannis Louridas, Paul Gazzillo, Eric Koskinen, Zhong Shao
Yale University

Abstract

The classic Herlihy-Wang notion of concurrent objects has had great success in theories and implementations (e.g., Java, EJ, CosenNat), providing programmers with the simple abstraction of an atomic object. Since then, software transactions have appeared, also touting the goal of providing an abstraction above transactions. However, despite some vertical composition strategies within particular STM implementations, a fundamental notion of vertical composition has remained elusive.

In this paper, we show the existence of vertical composition with the notion of reversible atomic objects. By modeling occurrences of transactions as the method boundary and requiring that every object method construct its own inverse, we obtain a clearer semantics and support vertical composition. In fact, we do not even require that one layer use the same implementation (e.g., pessimistic versus optimistic) as another, nor that the object be transactional at all. Precisely, we begin with semantics in which abstract-level operations are composed from concrete base operations, accounting for method and invariants. These transactional implementations are part of the context of an environment that includes a need a deadlock-avoiding container manager that enforces progress. The container manager may, at any point, apply inversions on behalf of a currently executing transaction. Our work here has two key points: first, that programs that use transactions in this framework are a conceptual refinement of the same program instead composed with atomic specifications, and that layers can be composed vertically.

Our compositional treatment in terms of a single shared layer gives rise to novel frictionless, scalable, efficient systems. We show that a library of reusable atomic objects is transactional, and in fact, we prove that it is easy to derive in the framework. This is a consequence of the fact that the abstract-level composition is not unique.

In the literature, this idea of vertical composition has been the focus of several recent works in both software transactions [45, 46, 47], whose success has been modest owing to performance and semantic difficulties. These systems permit what we believe to be a limited form of vertical composition: all transaction layers are handled by the same monolithic transactional implementation [3, 23]. But is it reasonable to require this? In this paper we argue for the fundamental idea that transactions should be distilled to a single concept that is independent of implementation. Consider a transactional system, for example, where POSIX-level commands (e.g., tenancy) are implemented transactionally over GFS internal data structures, and user-level tools provide a transactional interface. One would not expect that application-level transaction support should be required to make the same implementation decisions as the low-level block device (that interacts with hardware) just because they both use transactions.

Scope. In this paper, we focus on the Herlihy-Wang idea of vertically compositional concurrent objects in the new context where objects may be implemented with transactions. Specifically, we (1) build a framework for transactional implementation of objects, (2) study the form of creating our transactions are aligned with object method boundaries, and (3) require that every operation commit in its own layer. This choice leads to a cleaner semantics (thanks to #2 and #3) and, at the same time, more freedom in the post-transactional optimization (thanks to #1).

In this new scope, a reusable atomic object implements a higher-level abstraction composed with a transaction that applies various operations on a collection of base reversible atomic objects. These base objects can themselves be implemented with transactions or data, as we show, be a simple wrapper around an existing concurrent atomic object. The programmer must also specify each abstract-level composition and, in the body of the method, construct the method’s entries before committing. Inverse composition can be easily implemented using the base objects’ inversions provided that, at least, the lowest level atomic objects have explicit inversions. Nonetheless, this approach provides a library of reusable atomic objects that provides minimal semantics including containment.

1. Introduction

The famous Liveness paper of Herlihy and Wing [42] established the idea of concurrent objects that can be expressed as atomic from the perspective of threads accessing them. This has been enormously successful, leading to libraries (e.g., Java, EJ, CosenNat, and the C++ Boost libraries) that exploit this object abstraction and allow easy construction of concurrent systems. In recent years, there has been a push toward supporting transactions on a programming-language level of abstraction, where programmers can build atomically consistent executions in a user-level setting.

The primary goal of this paper is to show that this abstract-level composition is expressible, and we prove that it is easy to derive in the framework. This is a consequence of the fact that the abstract-level composition is not unique.
More Goodies in the Paper

- Full Formalization
- Novel treatment of contention manager as an environment
- Proofs
- Semantic model leads to …
- Transactional Version of the universal construction