Vertical Composition of Reversible Atomic Objects

Timos Antonopoulos   Paul Gazzillo   Eric Koskinen   Zhong Shao
Yale University

Abstract

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

In this paper, we distill the essence of vertical composition, with the notion of *reversible atomic objects*. By restricting occurrences of transactions to the method boundary and requiring that every object method construct its own inverse, we obtain a cleaner semantics that supports vertical composition. In fact, we do not even require that one layer use the same implementation (e.g. pessimism versus optimism) as another, nor that the object be transactional at all. Formally, we begin with a semantics in which abstract-level operations are composed from constituent base operations, accounting for conflict and inverses. These transactional implementations are put in the context of an environment that includes a novel a deadlock-mitigating contention manager that ensures progress. The contention manager may, at any point, apply inverses on behalf of a currently executing transaction. Our work has the first proof that programs composed with implementations in this framework are a contextual 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 log gives rise to a novel transactional variant of the universal construction. We have implemented a library of reversible atomic objects. We demonstrate that it is easy and intuitive to build complex concurrent implementations (e.g. a multi-threaded layered filesystem) by vertical composition of atomic components.

1. Introduction

The landmark Linearizability paper of Herlihy and Wing [22] established the idea of concurrent objects that can be viewed as atomic from the perspective of threads accessing them. This has been enormously successful, leading to theories [4, 15, 43, 54, 55] and implementations (e.g. `java.util.Concurrent` and the C++ Boost libraries) that exploit this atomicity abstraction and allow one to build complex systems from sensible building blocks.

In recent years, there has been a push toward supporting transactions: a programming language abstraction intended, at first, to allow programmers to build atomic sections of ad-hoc memory operations, with conflict management handled by an underlying runtime system. Later it was realized [19, 29] that these transactions need not consist of memory operations but can, instead, consist of base ADT operations. These works showed that the ADT operations can themselves be concurrent (linearizable/atomic) objects and, if they are, there are performance gains to be had [19].

In the literature thus far, vertical composition has come in the form of so-called *nested transactions* [39, 40, 42], 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 not: the fundamental idea of transactions should be distilled to a single concept that is independent of implementation. Consider a transactional filesystem, for example, where POSIX-level commands (e.g. `rename`) are implemented transactionally over OS internal data-structures, and low-level disk drivers 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 re-focus on the Herlihy/Wing idea of vertically composable concurrent objects in the new context where objects may be implemented with transactions. Specifically we (1) break free of a monolithic implementation of nesting, (2) restrict the form of nesting so that transactions are aligned with object method boundaries, and (3) require that every operation construct its own inverse. This choice leads to a cleaner semantics (thanks to #2 and #3) and, at the same time, more freedom in the pessimistic-vs-optimistic transactional implementation (thanks to #1).

In this new scope, a *reversible atomic object* implements a high-level abstract operation assembled 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 else, as we show, be a simple wrapper around an existing concurrent atomic object. The programmer must also specify each method's commutativity and, in the body of the method, construct the method's inverse before it commits/returns. Inverse construction can be easily automated using the base objects' inverses provided that, at least, the lowest level reversible atomic objects have explicit inverses. Nonetheless, a programmer may choose to provide one manually, especially in cases where the inverse can be achieved with fewer base operations.

That reversible atomic objects are based on commutativity and inverses is, on the one hand, unsurprising since so many recent works on transactions involved commutativity and inverses [19, 20, 28–30, 33, 39, 40, 42]. What is perhaps surprising is that these can be used as the fundamental base ingredients, bridging together the new transactional world with the existing legacy of concurrent atomic objects. The main benefit of working with reversible atomic objects is that they provide formal guarantees including contextual refinement and vertical composition, as described below.

This paper. We develop a methodology, theory, and implementation of vertically composable reversible atomic objects. We find that, with the above approach, we can achieve a concept of implementation-independent and vertically composable atomic transactional objects, thus lifting the Herlihy/Wing notion of atomic objects [22]. Specifically, we make the following steps forward:

- A specification of vertically composable reversible atomic objects and well-formedness criteria thereof. (Section 4)
- A vertically composable semantics of concurrent threads in which abstract-level operations are composed from constituent base operations, accounting for conflict and inverses. (Section 5)
- A proof that programs composed with implementations in this framework are a termination-sensitive contextual refinement of the same program instead composed with atomic specifications, as well as a proof of vertical composition. (Section 6)
- A demonstration of how progress is achieved through a novel treatment of contention management as an environment that may, at any point, invert an object's uncommitted operations. (Section 7)
- A novel transactional variant of the universal construction, arising from our compositional treatment in terms of a single shared log. (Section 8)
- An implementation of a collection of publicly available[1] reversible atomic objects. (Section 8)

The model we devise is expressive enough to cover a wide variety of known implementations, including all of those that are described by the recent Push/Pull model [28]. Our formal framework abstracts over thread implementation, leaving the question of opacity [16] up to the threads: they may choose to view or ignore log entries that contain effects of uncommitted transactions. Our implementation ensures that they don't.

Our implementation demonstrates that it is easy and intuitive to build complex concurrent implementations (e.g. a multi-threaded filesystem) by vertical composition of atomic components. Our filesystem is constructed from several reversible atomic objects (RAOs): the filesystem RAO itself with standard file operations, a moveable hashtable RAO built with transactions, a directory tree RAO for the directory hierarchy, and a hashtable RAO that wraps a base atomic hashtable object.

Limitations. We believe that reversible atomic objects provides an avenue toward composing transactions in a way that is both semantically clean and efficient. Our work is currently limited a couple ways. We assume threads cannot access objects across layers without using a wrapper method. In practice, wrapper methods are straightforward to create or even automate. They do not permit arbitrary atomic sections, i.e., not aligned with methods; we argue this permits a cleaner semantics. We assume recursive calls are not allowed.

2. Programming with Reversible Atomic Objects

We now give an overview of reversible atomic objects by way of an example. Let us say that we want to implement a fast, concurrent file system for user applications. File system operations touch multiple data structures that must be kept in sync with each for integrity, a challenge for concurrent programs. At the top level, we will implement a FileSystem reversible atomic object that will provide standard POSIX-like file operations such as moveFile (a.k.a. rename), mkdir, rm, etc. The FileSystem object can be seen on the top of Figure 1. This FileSystem object needs to be implemented using fast base objects so we will exploit our vertical composition, and use a reversible atomic tree (for directory lookups) and reversible atomic hashtable (to store the payload file data for each

[1] Our implementation is available as part of the anonymous supplemental material.

FileSystem, **MoveableHashtable**, and **Hashtable**

```
class FileSystem[P, V] : RAO {
  MoveableHashTable[P, V] mht
  DirectoryTree[P] tree
  ...
  moveFile($p_1$, $p_2$) = opt_atomic{{ /* conflict: $p_1$,$p_2$,
        $lca(p_1,p_2)$ */
    $l_0$: $v$ = mht.get($p_1$) ↦ $inv_0$
    if ($v$ is empty) {
      cmt_return ($inv_0$, false)
    else {
      $l_1$: mht.move($p_1$, $p_2$) ↦ $inv_1$
      $l_2$: tree.moveNode($p_1$, $p_2$) ↦ $inv_2$
      cmt_return (opt_atomic{{$inv_2$; $inv_1$; $inv_0$}}, true)

    }
  }}
}
```

```
class MoveableHashtable[K, V] : RAO {
  Hastable[K, V] ht
  ...
  get($k$) = pess_atomic{{ /* conflict: $k$ */
    $l_0$: $v$ := ht.get($k$) ↦ $skip$
    cmt_return (skip, $v$)
  }}

  /* conflict: $k_1$, $k_2$, $size$ */
  move($k_1$, $k_2$) = pess_atomic{{
    $l_0$: $v$ := ht.get($k_1$) ↦ $inv_1$
    $l_1$: $v_{old}$ := ht.put($k_2$, $v$) ↦ $inv_2$
    $l_2$: ht.remove($k_1$) ↦ $inv_3$
    cmt_return (pess_atomic{{$inv_3$; $inv_2$; $inv_1$}}, ⊥)
  }}
}
```

```
class Hashtable[$\mathbf{K}$, $\mathbf{V}$] : RAO {
  ConcurrentHashtable[$\mathbf{K}$, $\mathbf{V}$] cht

  get($\mathbf{k}$) { /* conflict: $\mathbf{k}$ */
    x = cht.get($\mathbf{k}$)
    cmt_return (skip, x)
  }
  put($\mathbf{k}$, $\mathbf{v}$) { /* conflict: $\mathbf{k}$, $sz$ */
    $v_{old}$ = cht.get($\mathbf{k}$)
    cht.put($\mathbf{k}$, $\mathbf{v}$)
    cmt_return (cht.put($\mathbf{k}$, $v_{old}$), $v_{old}$)
  }
  remove($\mathbf{k}$) { /* conflict: $\mathbf{k}$, $sz$ */
    $v_{old}$ := cht.get($\mathbf{k}$)
    cht.remove($\mathbf{k}$)
    cmt_return (cht.put($\mathbf{k}$, $v_{old}$), $v_{old}$)
  }
}
```

Figure 1. Implementation of a filesystem, using a Tree and a MoveableHashTable.

full path). These are the internal data-structures mht and tree. Notice that each object may have its own transactional implementation style (i.e. pessimistic-vs-optimistic), denoted as **pess_atomic{{ }}**, **opt_atomic{{ }}**, etc. We will return to this later in this section.

A reversible atomic object O implements an abstract atomic operation f, built from base operations, that it assumes are already atomic. Such an object constructs an abstract state transformer $O.f(\vec{x}) : \Sigma \to \Sigma$ out of one or more *constituent* (base) state transformers $O_1.a, O_2.b, O_3.c : \Sigma \to \Sigma$, without knowledge of the detailed implementation of O_1, O_2, O_3. One can view this as:

$$\begin{tikzcd}
\sigma \arrow[r, "O_1.a"'] \arrow[rrr, bend left=40, "O.f(\vec{x})"] & \sigma_1 \arrow[l, dashed, bend right, "O_1.a^{-1}"'] \arrow[r, "O_2.b"'] & \sigma_2 \arrow[l, dashed, bend right, "O_2.b^{-1}"'] \arrow[r, "O_3.c"'] & \sigma' \arrow[l, dashed, bend right, "O_3.c^{-1}"'] \arrow[lll, dashed, bend left=40, "O.f^{-1}(\vec{x})"]
\end{tikzcd}$$

Let us take, for example, the moveFile method in Figure 1. The code in light gray could be inserted by a compiler. While the remaining user code of this simple example looks straight-forward (we hope!) and the features of this method look familiar, there are many behind-the-scenes details to be appreciated. We will discuss each of these in turn:

1. Alignment of transaction with method boundary.
2. Specification of conflict.
3. Invocations of constituent operations.
4. Assembling the inverse.
5. The commit-and-return statement.

Aligning transactions with the method boundary. The first thing to note is that reversible atomic objects restrict the way in which transactions can be used: they must be correlated with object methods. The body of moveFile *is* an atomic section, denoted **opt_atomic{{**...**}}**. Moreover, this atomic block ends with a **cmt_return** statement that both commits the transaction and returns the abstract operation's response (in this case it is void) back to the caller. We will discuss the **cmt_return** statement more later.

Conflict specification. All reversible atomic object methods must provide a *conflict specification*. Considering the current state of O and the current active operations on the object, O must provide a conflict (commutativity) specification $\bowtie_O$ that specifies when an invocation of $O.f(\vec{x})$ commutes with all other active operations.

In the implementation of moveFile, we use the notion of commutativity conflict points (see [13]) that provide a concise specification that can be checked efficiently at runtime. This conflict specification is beyond the scope of this paper but, intuitively, the moveFile conflict specification (Line 5) means that an invocation of moveFile does not commute with a concurrent invocation that accesses the contents of file p_1, accesses the contents of file p_2, or, indeed, accesses the directory subtree rooted at the least common ancestor of the two paths. The commutativity specification here is independent of how the transactional runtime conflict management system operates (pessimistically, optimistically, etc.) at any given level.

One possible transactional implementation at this layer would be *optimistic*. As we do in our implementation (Section 8), the implementation of this layer could furnish each thread/transaction with its own local copy of the object that they can immediately mutate. Later, at commit time, the transactional system must somehow communicate these operations with other threads and, in the event of conflict on p_1, p_2, or $lca(p_1, p_2)$, invert some or all of these local operations. A more conservative approach is *pessimism*, whereby the transactional system pauses the transaction until there are no conflicting concurrent operations that access $p_1, p_2, lca(p_1, p_2)$.

It is important to remember: the above conflict is at the FileSystem layer and *has nothing to do* (yet) with conflict at the level of the constituent objects: MoveableHashtable and DirectoryTree.

Invoking constituent operations. The implementation of a reversible atomic object method is free to perform methods on constituent reversible atomic objects: $O_1.a, O_2.b$, etc. In the running example, the body of moveFile manipulates methods of the constituent objects mht and tree. moveFile moves the file payload to the new path and then performs a tree manipulation to restructure the directory hierarchy. These constituent operations may involve return values as in the call to mht.get on Line 6 and moveFile may take different actions depending on these return values. Looping is also permitted, provided that the loop eventually terminates.

With each operation, there are two things to note. First, the location of each operation is marked (perhaps by a compiler) with a label: l_0, l_1, l_2, etc. At each such location immediately preceding the operation, the compiler also captures the continuation [26]. Later, this will be used in case the operation is inverted. Second, each constituent operation, since it is also a reversible object, will return an inverse operation. This inverse inv_0, inv_1, etc. is saved for two reasons. The inverse operation may be invoked by the environment contention manager (discussed below). Furthermore, it may be used in the construction of the overall inverse operation for moveFile, as seen on Line 12 and discussed next.

Assembling the inverse. Before completing the operation $O.f(\vec{x})$ (via a commit-and-return), a reversible atomic object must prepare the inverse operation $O.f^{-1}(\vec{y})$. Inverses can be generated automatically (perhaps by a compiler) by assembling the inverses of the constituents, as in this case for the inverse of moveFile on Line 12. In other cases the programmer might provide a smarter inverse. Note that the inverses themselves, while atomic, do not themselves have inverses.

This inverse operation may be used at the next level up. In a transaction at that higher level, $O.f(\vec{x})$ is viewed as an atomic (and reversible) constituent object and, therefore, may potentially be inverted as discussed in Section 2.1. There is a subtlety here about how inverses interact with commutativity: how do we know whether $O.f^{-1}(\vec{y})$ is still a valid inverse when there may be other concurrent operations? Crucially, this inverse $O.f^{-1}(\vec{y})$ is a short-lived inverse and can only be used during the lifespan of the above transaction. Consequently, since the above transaction ensures that $O.f(\vec{x})$ is free of conflict with any other concurrent operation, it is easy to show that one can commute $O.f(\vec{x})$ forward in time, until it is adjacent to $O.f^{-1}(\vec{y})$ and that these two then annihilate each other.

***The commit-and-return statement* cmt_return.** The method (and transaction) completes with a single statement **cmt_return**. At this point, the transaction is attempting to commit. The way in which the commit happens is up to the transactional system at this level. A pessimistic implementation **pess_atomic{{ }}** will already have ensured that the current transaction has the right-of-way (i.e. there are no concurrent operations that pertain to p_1, p_2,, or $lca(p_1, p_2)$) so this commit event can happen immediately. An optimistic implementation, on the other hand, would need to perform some conflict detection pertaining to p_1, p_2, and $lca(p_1, p_2)$. The first argument to **cmt_return** is the inverse for moveFile. Once **cmt_return** completes, the operation is considered complete, and control is returned to the calling object in the next level up.

Client threads. Threads, denoted $P = (C_1 \parallel \cdots \parallel C_n)$ are the clients of reversible atomic objects and are the top-most layer. As with the lower layers, the clients call reversible atomic object meth-

```
class Creator : Thread {
  run(FileSystem[K,V] fs) {
    for i := 1..100
      fs.addFile("file" + i, i * i);
  }
}
class Mover : Thread {
  run(FileSystem[K,V] fs) {
    while (num_moved < 50) {
      for i := 1..100 step 2 {
        moved = fs.moveFile("file" + i, "moved_file" + i);
        if (moved)
          num_moved++;
      }
    }
  }
}
class Printer : Thread {
  run(FileSystem[K,V] fs) {
    num_files = 0;
    while (num_files < 100)
      print fs.numFiles();
  }
}
```

Figure 2. Example clients using a reversible atomic FileSystem object. Creator makes files named `file1` to `file100`, Mover moves with even-numbered names to `moved_file#`, and Printer prints the number of files forever.

ods, treating them as atomic. The client cannot use transactions. It need not be reversible nor atomic, nor does it have to collect inverses of constituent objects.

Figure 2 is an example of threads all operating on the same FileSystem object. The Creator thread generates files named `file1` to `file100`. The Mover thread moves files with an even number in their name to `moved_file##` until its has moved 50 files. Lastly, Printer reports the number of files until there are 100 files.

For this example, we assume a nondeterministic scheduler that can schedule the threads in any order. For expressivity, no particular ordering for these threads (i.e. waiting) is required. Instead, the Mover takes advantage of the `moveFile`'s interface that tells the thread whether the move happened or not. The file operations need to be atomic, otherwise inconsistencies between the underlying hashtable and tree objects may arise. For instance, the number of files seen by Printer may decrease, even though files are never removed in this example, because the move method creates new file before removing the old one. With atomicity, the number of files printed is monotonically increasing.

2.1 Across this layer: Conflict, Progress, Refinement.

Before we look at the implementation of vertically composed MoveableHashtable and DirectoryTree, let us discuss contention that may arise from other threads invoking operations on the FileSystem. Transactional memory systems address contention with a so-called contention manager that implements some policy, deciding whom should be aborted [48, 49, 51]. If the contention manager is able to also know when deadlocks occur (e.g. [27]), then it can implement a policy that ensures overall progress.

Reversible atomic objects take a novel formal view of the contention manager as an environment that (a) controls the scheduling of transactions [36] (b) can detect deadlocked transactions, and (c) can partially abort operations by invoking the operations' inverses on behalf of the transaction. Combining these elements, we show a simple such environment that is able to ensure that every transaction eventually completes (Section 7).

For example, let us say that the following has occurred:

1. Transaction τ begins
2. Transaction τ' begins
3. Transaction τ' completed mht.move(7,8)
4. Transaction τ completed mht.move(5,6)
5. Now, τ wants to invoke mht.get(6)
   and τ' wants to invoke mht.get(8).

There is a deadlock here because each transaction would like to execute an operation that conflicts with one already completed by the other transaction. To resolve this deadlock, the environment can clear a path for the oldest transaction τ to complete by executing the *inverse* of τ' operation mht.move(7,8) and then preventing τ' from being scheduled until τ commits. When a later conflict occurs with some other transaction τ'', the environment may have to abort and unschedule τ (if τ'' is older than τ) or else abort and unschedule τ''.

Because the environment is able to invert operations on behalf of transactions, those transactions must be aware that this may happen. Later (and in Section 8) we will discuss how this can be done via a novel transactional version of the universal construction. When a thread finds that the environment has inverted one or more of its operations (always in reverse order) back to some location l_i, the thread must resume execution at l_i via a previously captured continuation.

Contextual Refinement. The main formal result of this paper (Section 6) is that reversible atomic objects provide a formal contextual refinement guarantee. If objects in the system follow the above criteria, abiding conflict specifications and establishing inverses then, 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. Formally,

$$[\![C_O]\!]_{\text{interleaved}} \sqsubseteq [\![S_O]\!]_{\text{interleaved}}$$

Our formalization is a compositional semantics in which abstract-level operations are composed from constituent base operations. Threads are executed in the context of an environment scheduler. By quantifying over all possible schedulers, each individual trace of the system is deterministic.

Threads and environment communicate by appending events to a single *shared event log*. This is the mechanism by which the environment can perform contention management. Transactions publish their operations in the log as well as indicate when they cannot make progress. Meanwhile, the environment can be proactive by appending inverses to the log and/or scheduling threads as it deems appropriate. The shared log also gives rise to a novel transactional version of the universal construction, which we discuss later.

2.2 Below this layer: Vertical Composition.

***The reversible atomic* MoveableHashtable.** FileSystem is built from two constituent reversible atomic objects, one of which is the MoveableHashtable, defined in the middle of Figure 1. This is a new type of hashtable that provides an operation to move data between keys, using a single Hashtable RAO as a constituent object. The move operation, defined on line 10 has three constituent operations: ht.get the value from k_1, ht.put it in k_2, and ht.remove k_1. The MoveableHashtable's move operation must, as always, construct its own inverse.

At this layer, notice that we have decided to execute transactions pessimistically with **pess_atomic{{ }}**. This demonstrates the expressivity of reversible atomic objects, that one can use different transactional implementations at different vertical layers.

Basic building blocks. Part of the power of reversible atomic objects can be seen in the base case, where one can place existing implementations of concurrent objects, say a ConcurrentHashtable (cht), in a reversible wrapper. This wrapper simply lifts the cht operations, specifies conflict and constructs inverses. These inverses "get the ball rolling," by allowing higher level operations to be able to automatically construct (at least default) inverses.

An example reversible atomic Hashtable is given on the bottom of Figure 1. It is important to realize that, although this particular implementation of MoveableHashtable uses the same key space as Hashtable, they are conceptually different. Therefore, we denote Hashtable keys/values with different fonts: $\mathbf{k} : \mathbf{K}$, $\mathbf{v} : \mathbf{V}$.

Take the put method, for example. The conflict specification pertains to key $\mathbf{k}$ and the fact that the overall size of the cht sz may change. The Hashtable put method first calls the concurrent hashtable cht.get method, saving the return value v_{old}. This is needed in order to be able to construct an inverse. Next, cht.put is called, atomically updating the ConcurrentHashtable. Finally, this wrapper returns the newly constructed inverse cht.put($\mathbf{k}$,v_{old}) and returning the old value to the caller.

The most important aspect to note is that this Hashtable reversible object wrapper *does not* execute transactions. These constituent cht operations will never be inverted or be the reason for conflict. Both inversion and conflict is covered by the wrapper Hashtable operation. For example, the reversible atomic Hashtable.put operation has conflict specification $\mathbf{k}$,sz. This is a sound specification because it covers both cht.get($\mathbf{k}$) and cht.put($\mathbf{k}$,$\mathbf{v}$). In summary, even though a concurrent atomic object building block is not transactional, the conflict specification in the wrapper reversible atomic Hashtable allows it to be used by a parent object such as MoveableHashtable.

In Section 6.1 we show formally that objects can be vertically composed. Theorem 6.2 says that, for any two objects O and Q with implementations/specifications C_O/S_O and C_Q/S_Q, that

$$[\![C_O \oplus C_Q]\!] \sqsubseteq [\![S_O \oplus S_Q]\!]$$

The FileSystem here is a limited example, but it demonstrates the key elements of reversible atomic objects. Below and in Section 8 we will discuss our implementation that includes the complete FileSystem example.

In comparison to nested transactions, reversible atomic objects are, on the one hand more restrictive—because atomic sections are aligned with object methods—yet, from this restriction we can obtain a cleaner compositional semantics. Moreover, this clear semantics allows one to plug in different kinds of transactional implementations. That is, if one decides to adopt the methodology of reversible atomic objects, then it comes with all of the formal guarantees discussed in the following sections of the paper.

2.3 Universal Construction and a Library of RAOs

The log-based semantics of the formalization of RAOs lead us to a novel universal construction for transactional concurrent objects. In 1991, Herlihy described a technique for a linearizable concurrent object out of any given sequential implementation and a single shared linearizable queue (i.e. log) [18]. As a reminder to the reader, in the original universal construction, threads communicate via a shared log and replicate objects locally. Each thread competes to append the next method invocation to the log. The thread-local copies are updated by applying the invocations in the shared log. This construction does not directly apply to a transactional setting.

As noted above, the compositional semantics that we present involves threads communicating their constituent operations through a single shared log. This gives rise to a *transactional universal construction* in which we can build a concurrent object that supports *transactions* given only a sequential implementation of the object. The construction works as follows. First, to support transactions, the log contains more information. Log entries have a transaction identifier, and threads may also append begin and commit messages to the log. Most importantly, inverses may be appended by a contention manager, signaling an abort. While appending a new operation to the log, a thread that receives inverses aborts without appending the new operation, retrying the inverted portion of its transaction. The `try_op` abstraction captures this behavior and is the core of the universal construction. Using `try_op`, reversible atomic object implementations can express a range of transactional policies from pessimistic to optimistic. The more eagerly a method appends its operations to the log, the more pessimistic it is. In the other extreme, an RAO may optimistically perform all operations on a thread-local copy, attempting to append the all operations at once to the log.

The transactional universal construction directly translates to an implementation of a runtime system for reversible atomic objects. We have implemented such a system as well as library of reversible atomic objects, including a complete version of FileSystem object. This library, along a discussion of implementation strategies, is discussed in Section 8. The transactional universal construction demonstrates that reversible atomic objects are not an arbitrary formalism, but reveal that reversibility is fundamental to transactional objects and vertical composition.

3. Preliminaries

In this section we establish some formal preliminaries. We will described the shared log, how threads/objects/environments work with it, as well as inverses and conflict.

3.1 States, Operations, Event Logs

We will work with a *state space* Σ. An operation is given by a, b, etc. and are of type $\Sigma \to \Sigma$. We let $\mathbf{Ops}$ be a set of base operations.

Global Log and Threads. We will work with a globally shared system log ℓ : list Ev, which records threads' events from a domain of events Ev. The domain of logs is $\mathcal{L}$ and we let $\mathcal{T}$ be a domain of unique thread identifiers, with τ to denote a single thread ID. We will define events later but, for now, one possible event is (τ, a) where a is a base operation. We use the notation $\ell[i]$ to mean the ith element of the log ℓ. We will use $\cdot$ to denote the append operation on lists/sequences such as event logs. We write $\ell \cdot (\tau, a)$ to mean $\ell \cdot \{(\tau, a)\}$.

We abstract away thread-local internal details, treating a *thread configuration* as (st, c, r) which is a thread-local state $st \in St$, a continuation code $c \in Cd$, and a function $r \in R : \mathcal{L} \to (St \times Cd) \to (St \times Cd \times \mathcal{L})$. We denote a such transition as $(st, c) \xrightarrow{r\ \ell} (st', c', \ell')$ which, from a start configuration (st, c) and current system log ℓ (described next), generates a sequence of events ℓ and a next configuration (st', c').

Observations. An *observation* $obs(\ell \cdot \{(\tau, a)\})$ is the return value of the last operation a in log $\ell \cdot \{(\tau, a)\}$ and we will assume that it is uniquely determined. For example, a reasonable semantics for a hashtable would have behavior such that $\forall \ell.\ obs(\ell \cdot (\tau, ht.put(3, 42)) \cdot (\tau, ht.get(3))) = 42$. We use $obs_i(\ell)$ as shorthand for the observation of $\ell[i]$.

3.2 Objects

We have a collection of *objects* $O_1, ..., O_n$, and each object has access to an isolated region of that space. An object *method* is given by $O.f(\vec{x})$ where O is the name of the object, f is the name of the method, and $\vec{x}$ are the arguments, which is a sequence over some domain $\mathbf{D}$.

Given an object O and one of its methods $O.f(\vec{x})$, we define $\mathsf{spec}_{O.f(\vec{x})} : \mathbf{D}^* \to \mathbf{Ops}$. Such a specification function returns the exact sequence of operations to be performed for the given arguments to the method. As a simple example, if the method $O.f(x)$ performs the base operation sequence a, b if $x > 0$ and c otherwise, then the function $\mathsf{spec}_{O.f(\vec{x})}(x)$ would simply return the corresponding sequence according to the value of x. Furthermore, let $\mathsf{specLog}_{O.f(x)} : \mathcal{L} \times \mathbf{D}^* \to \mathcal{L}$, which is a mapping that given a log ℓ and a sequence of arguments $\vec{x}$ for the method $O.f(\vec{x})$, traverses the log ℓ and consults the function $\mathsf{spec}_{O.f(\vec{x})}$ and produces the correct sequence of events in the log, that corresponds with the actions of invoking the method, executing the correct sequence of base operations and then appending the event of committing and returning.

Given an object O, we define its specification $S_O : \mathcal{L} \to \mathcal{L}$ to be a mapping that given a log ℓ returns an extension of it $\ell' = \ell \cdot \ell''$, where ℓ'' comprises the necessary events to be completed until the CmtRet for the object method. Formally, suppose ℓ_1, ℓ_2 are logs such that $\ell_1 = \ell \cdot (\tau, \mathsf{Ivk}\ O.f(\vec{x}))$ and $\ell_2 = \mathsf{specLog}_{O.f(\vec{x})}(\ell_1)$. Let $\ell_{2,p}$ be any prefix of ℓ_2 and $\ell_{s,2}$ the remaining suffix (such that $\ell_2 = \ell_{2,p} \cdot \ell_{2,s}$). Then $S_O(\ell_1 \cdot \ell_{2,p})$ is equal to $\ell_{2,s} \cdot (\tau, \triangledown)$.

The implementation of an object O, denoted by $C_O : \mathcal{L} \to \mathcal{L}$, also returns an extension on the given log, but in contrast to S_O, this extension does not contain all necessary events, but contains only the next event, together with a yield event. Formally, for any log ℓ, if $S_O(\ell) = \ell' \cdot (\tau, \triangledown)$, then $C_O(\ell) = e \cdot (\tau, \triangledown)$, where e is the first event in the log ℓ', in the case where ℓ' is not the empty sequence, and $C_O(\ell) = (\tau, \triangledown)$ otherwise.

3.3 Parameterized base operations

We require a prefix-closed predicate on logs $\mathsf{allowed}(\ell)$ that indicates whether $\forall i \in [0, len(\ell) - 1]$. $obs_i(\ell)$ is valid according to the sequential specifications of the objects. For convenience we will also write ℓ allows n which simply means $\mathsf{allowed}(\ell \cdot \{n\})$. Taking a stack S, for example, and $\ell = \{S.push(5) \cdot S.pop()\}$ we would say that $\mathsf{allowed}(\ell)$ provided that $obs(\ell) = 5$.

We define a precongruence over operation sequences $\ell_1 \preccurlyeq_{obs} \ell_2$ by requiring that all $\mathsf{allowed}$ extensions of the log ℓ_1, are also $\mathsf{allowed}$ extensions to the log ℓ_2. We use a coinductive definition so that the precongruence can be defined up to all infinite suffixes. Formally, for all ℓ_1, ℓ_2,

$$\frac{\mathsf{allowed}(\ell_1) \Rightarrow \mathsf{allowed}(\ell_2) \quad \forall a.\ (\ell_1 \cdot a) \preccurlyeq_{obs} (\ell_2 \cdot a)}{\ell_1 \preccurlyeq_{obs} \ell_2}\ gfp$$

Informally, the above greatest fixpoint says that there is no sequence of observations we can make of ℓ_2, that we can't also make of ℓ_1. This is more general than simply requiring that the set of states reached from the first sequence be included in the second. Unobservable state differences are also permitted.

We also require an abstract version of the $\mathsf{allowed}$ predicate, denoted by $\widehat{\mathsf{allowed}}$, that indicates whether the observation of each CmtRet event is valid. More generally, $\widehat{\mathsf{allowed}}$ can be parameterized by a set of objects $O_1, \ldots, O_n$ in which case the predicate indicates whether the observation of each CmtRet event, restricted to the methods in the set of objects, is valid.

Using the predicate $\widehat{\mathsf{allowed}}$, we define the notion of abstract observational precongruence.

$$\frac{\widehat{\mathsf{allowed}}(\ell_1) \Rightarrow \widehat{\mathsf{allowed}}(\ell_2) \quad \forall \ell \exists \ell'. \ell_1 \cdot \ell \preccurlyeq_{\widehat{obs}} \ell_2 \cdot \ell'}{\ell_1 \preccurlyeq_{\widehat{obs}} \ell_2}\ gfp$$

$Ev ::=$	$(\tau, \mathsf{Ivk}\ O.f(\vec{x}))$	Invoke an abstract method
	(τ, a)	Implementation base operation
	(τ, a^{-1})	Cancel a base operation
	$(\tau, \mathsf{CmtRet}\ O.f(\vec{y}))$	Commit and establish inverse
	(τ, Term)	Thread termination
	$(\tau, \triangledown)$	Yield to another thread

Figure 3. Events of the system.

3.4 Inverses and conflict

We assume that for every operation a, there is an inverse operation a^{-1}, which is to be exactly such that $\forall \sigma,\ a^{-1}(a(\sigma)) = \sigma$. Unfolding the structure of a, we say that $O.f^{-1}(\vec{y})$ is the function such that $\forall \sigma,\ O.f^{-1}(\vec{y})(O.f(\vec{x})(\sigma)) = \sigma$. Notice that constructing an inverse operation f^{-1} may require arguments other than those passed to f. Many existing implementations already have a requirement of inverses [19, 40, 42].

We define an operation *conflict* relation with respect to an operation sequence and observations thereof as follows:

$$\ell_a \overset{\ell}{\lhd} \ell_b \quad \equiv \quad \ell \cdot \ell_a \cdot \ell_b \preccurlyeq_{obs} \ell \cdot \ell_b \cdot \ell_a$$

Unfolding the definition of $\preccurlyeq_{obs}$, one can see that conflict (commutativity) means that ℓ_a and ℓ_b make the same observations in either order and the sequences $\ell \cdot \ell_a \cdot \ell_b$ and $\ell \cdot \ell_b \cdot \ell_a$ are observationally equivalent prefixes.

4. Vertical Composition through Abstraction

In this section, we describe how an object implementation C_O (or specification S_O) constructs an overall operation $O.f(\vec{x})$ out of a series of base operations. We then give well-formedness criteria for these objects.

4.1 Events

In addition to the base event (τ, a), there are other events that can be emitted by a thread transition $\xrightarrow{\tau\ \ell}$. The events are given in Figure 3. As mentioned above, event (τ, a) is an instance of thread τ performing operation a. The first event in Figure 3 is $(\tau, \mathsf{Ivk}\ O.f_i(\vec{x}))$ which models thread τ invoking an operation $O.f_i(\vec{x})$. If $O.f_i(\vec{x})$ is already an atomic event, then the next method generated by τ is a response event $(\tau, \mathsf{CmtRet}\ O.f_i(\vec{x}))$ whose observations give the operation's return value. Otherwise, $O.f_i(\vec{x})$ may be implemented with a transaction. We will describe this in the next section.

The event Term signals thread termination and event $\triangledown$ signals that the thread is yielding to the environment (described later) Note that there is no explicit abort event. We model abort by a series of cancellation steps.

The events described above permit us to model vertical composition, where an abstract operation $O.f(\vec{x})$, with atomic semantics $S.f(\vec{x})$ is implemented via a series of transaction events involving base operations $a, b, \ldots$. In this section we formalize this composition, by exploring how an object implementation may construct abstract operations (mutations) and observations (return values) from the mutations and observations of the base observations.

4.2 Abstract operations

The events in Figure 3 allow us to model vertical composition. Specifically, an object operation $O.f_i(\vec{x})$ may be implemented with transactions, but only in a particular way (unlike nested transactions). That is, the implementation of $O.f_i(\vec{x})$ consists of a transaction immediately within the body of the method:

$$f_i(\vec{x}) = \textbf{pess\_atomic}\{\!\{ \quad \dots \quad \textbf{cmt\_return}\ k; \}\!\}$$

Here, k is a depiction of the observation of the overall abstract operation $O.f_i(\vec{x})$.

A thread calling this operation is modeled as the following following event sequence:

$$(\tau, \mathsf{Ivk}\ O.f_i(\vec{x})), (\tau, a), (\tau, b), (\tau, c), (\tau, \mathsf{CmtRet}\ O.f_i(\vec{x}))$$

The invocation of $O.f_i(\vec{x})$ also signals the beginning of a transaction (unlike nested transactions, there is no separate "begin" event).

We call such an event sequence (or log segment), an *abstract operation sequence*. In other words, an abstract operation sequence ℓ is inductively defined as a sequence $(\tau, \mathsf{Ivk}\ O.f(\vec{x})) \cdot \ell' \cdot (\tau, \mathsf{CmtRet}\ O.f(\vec{x}))$, where ℓ' comprises a sequence of base operations and abstract operation sequences. Furthermore, in such a case we call this abstract operation sequence, an $O.f(\vec{x})$ abstract operation sequence. We define the predicate $\mathsf{aos}_\tau(\ell, O.f)$ that holds for a segment ℓ when it is an $O.f(\vec{x})$ abstract operation sequence over the thread τ.

4.3 Well-formedness

We now give some well-formedness constraints on objects.

We first have a basic well-formedness constraint, requiring the object to yield sensible event histories. We formalize this with an inductive predicate over logs $\mathit{wfir}_{\tau,O.f}$. This predicate, intuitively, means that inverses are used only to cancel previously issued operations from the same invocation.

A second condition, as discussed in Section 2, is that an object method $O.f(\vec{x})$ must construct a corresponding abstract inverse $O.f^{-1}(\vec{y})$, that is returned to the caller in the CmtRet event. The inverse may be used at that higher level by the parent or, more likely, the contention management scheme. As we discuss in Section 8, this can be done incrementally during the transaction or else immediately before the transaction commits.

We will further require that threads only generate (τ, a) events provided that a commutes with every operation b from another uncommitted transaction. To this end, we have a few definitions:

- $\mathsf{committedOps}(\ell) \subseteq \mathit{Ops}$: the set of base operations (τ, a) for which there is a subsequent $(\tau, \mathsf{CmtRet}\ \_)$ event in ℓ.
- $\mathsf{activeThreads}(\ell) \subseteq \mathcal{T}$: the set of every thread identifier τ such that there is a (τ, Ivk) event in ℓ, but no correlated $(\tau, \mathsf{CmtRet}\ \_)$ event.
- $\mathsf{activeOps}_\tau(\ell) \subseteq \mathit{Ops}$: the *sequence* of operations corresponding to τ (such that there is a (τ, Ivk) event in ℓ, but no correlated $(\tau, \mathsf{CmtRet}\ \_)$ event.) in the order they were generated
- $\mathsf{activeOps}_{\neg\tau}(\ell) \subseteq \mathit{Ops}$: the *sequence* of operations corresponding to all $\tau' \in \mathcal{T} \setminus \{\tau\}$ (such that there is a (τ', Ivk) event in ℓ, but no correlated $(\tau', \mathsf{CmtRet}\ \_)$ event.) in the order they were generated

We can now give the commutativity well-formedness condition:

$$\frac{\mathit{wfc}_\tau(\ell) \quad \mathsf{aos}_\tau(\ell_{O.f}, O.f) \quad \ell_{O.f} \overset{\ell}{\lhd} \mathsf{activeOps}_{\neg\tau}(\ell)}{\mathit{wfc}_\tau(\ell \cdot \ell_{O.f})}$$

$$\frac{\mathit{wfc}_\tau(\ell) \quad (\tau, a) \overset{\ell}{\lhd} \mathsf{activeOps}_{\neg\tau}(\ell)}{\mathit{wfc}_\tau(\ell \cdot (\tau, a))}$$

$$\frac{\mathit{wfc}_\tau(\ell) \quad e \in \{\mathsf{CmtRet}, \mathsf{Ivk}, \mathsf{Term}, a^{-1}\}}{\mathit{wfc}_\tau(\ell \cdot \{(\tau, e)\})}$$

Semantics

$$\frac{\mathcal{E}\ \ell\ T = (\ell', \tau)}{\ell, \bot, (T, \mathit{tm}) \xrightarrow{\mathcal{E}} \ell \cdot \ell', \tau, (T, \mathit{tm})}\ \textsc{Env}$$

$$\frac{\tau \in T \quad \mathit{tm}\ \tau = (\mathit{st}, c, r) \quad \mathit{st}, c \xrightarrow{r\ \ell} \mathit{st}', c', \ell' \cdot e \quad e \in \{\triangledown, \mathsf{Term}\}}{\ell, \tau, (T, \mathit{tm}) \xrightarrow{\tau} \ell \cdot \ell' \cdot (\tau, e), \bot, (T, \mathit{tm}[\tau \mapsto (\mathit{st}', c', r)])}\ \textsc{Thr}$$

Figure 4. The rules for Reversible Atomic Objects.

Intuitively, the first rule above means that every time thread τ generates a (τ, a) event, it commutes with all uncommitted operations of other transactions. All other events are well-formed.

Overall, we say that an object is well-formed if it satisfies both wfir_τ and wfc_τ for all τ.

5. Shared Log Semantics

We now describe a compositional game semantics that combines threads (given as a composition of C_O/S_O agents) with environments. We define a machine that is a game between a group of threads and an environment $\mathcal{E}$ from domain $\mathfrak{E}$, communicating via shared log ℓ. Threads invoke object operations $(\tau, \mathsf{Ivk}\ O.f(\vec{x})))$. The implementation of these operations, provided by C_O or S_O generates events for base operations $a, b, \dots$ and then a response is generated. Thread execution may yield and relies on environment $\mathcal{E}$ for scheduling. A similar use of a shared log for communication appears elsewhere [10, 28].

Definition 5.1 (RAO Game). *An RAO Game $\mathfrak{G} = (V, \rightsquigarrow)$ is a game between a set of threads/transactions and an environment. Game vertices $V : \mathcal{L} \times \mathcal{T} \times (\mathcal{P}(\mathcal{T}) \times \mathcal{TM})$ include the shared log ℓ, the current transaction's identifier τ, the set of threads in hand $T \subseteq \mathcal{T}$, a mapping $\mathit{tm} : \mathcal{T} \to (\mathit{St} \times C \times R)$ and an environment oracle $\mathcal{E}$.*

A partitioning on the vertices is induced, separating the vertices $V_\mathcal{E} = \{(\ell, \tau, \_) \mid \tau \notin T\}$ where it is the environment's turn and the vertices $V_T = \{(\ell, \tau, \_) \mid \tau \in T\}$ where it is the turn of one of the threads in hand. V_T can be further partitioned.

Edges. The edges $\rightsquigarrow$ have two different types $\xrightarrow{\tau}$ and $\xrightarrow{\mathcal{E}}$, given in Figure 4. The Env rule occurs when the current player τ is not in hand T. We denote such a thread with the symbol $\bot$. The environment takes a step, leaving the current thread (T, tm) untouched and yielding some new log events ℓ' and schedules the next thread $\tau' \in T$.

The environment $\mathcal{E} : \mathcal{L} \to \mathcal{P}(A) \to (\mathcal{L} \times \mathcal{T} \times \mathit{Ev})$ is taken from some domain $\mathfrak{E}$. In the simplest form, the environment can be thought of as a scheduler. We assume that the environment is *deterministic*, shifting the nondeterminism into the choice of $\mathcal{E}$ from domain $\mathfrak{E}$.

The Thr rule occurs when the current player τ is in T. Here, the thread's configuration (st, c, r) is loaded and a transition is taken under the current log ℓ, emitting new events $\vec{ev} \cdot \triangledown$. These events are used to construct log ℓ', the current thread is set to $\bot$ and the environment is consulted. Finally, all the accumulated events are enqueued and the tm is updated.

Merging Thread Components. The compositionality comes from the fact that it is easy to merge two thread groups T_1 and T_2. The merge $\oplus$ is defined as:

$$(T_1, \mathit{tm}_1) \oplus (T_2, \mathit{tm}_2) \equiv \left(T_1 \cup T_2, \lambda\tau. \begin{cases} \mathit{tm}_1\ \tau & \text{if } \tau \in T_1 \\ \mathit{tm}_2\ \tau & \text{otherwise} \end{cases}\right)$$

Object Components. Objects contain the implementation of operations and the implementation is executed on behalf of the calling thread. We define composition between a thread component and an object component's implementation C_O as follows:

$$(T, tm) \oplus C_O \equiv (T, \lambda\tau.\text{let } tm\ \tau = (st, c, r) \text{ in } (st, c, r \cup r_O))$$

where r_O contains the implementation of C_O which may consult the log ℓ in generating events. While $C_O : \mathcal{L} \to \mathcal{L}$, r_O has the same type as r which carries $(St, Cd) \to (St, Cd)$. However, r_O is the identity over this transformation.

We also have the specification component S_O of an object O, and define composition between a thread component and the specification component S_O as follows:

$$(T, tm) \oplus S_O \equiv (T, \lambda\tau.\text{let } tm\ \tau = (st, c, r) \text{ in } (st, c, r \cup r_S))$$

Here, r_S consults ℓ and, when τ has generated a $(\tau, \mathsf{Ivk}\ O.f(\vec{x}))$ event, r_S generates a single (atomic) event $(\tau, S.f(\vec{x}))$. From the above composition rules, one can construct more elaborate compositions between groups of threads and objects. Note that for objects and specifications, the operator $\oplus$ is not commutative.

6. Contextual Refinement & Vertical Composition

In this section we give our main theoretical results. We show that programs composed with implementations in this framework are a contextual refinement of the same program instead composed with atomic specifications and that layers can be composed vertically. We begin by defining traces and the whole program machine.

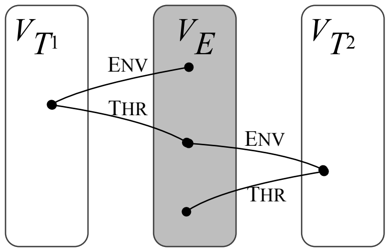
A trace of a game is an infinite alternation between a group of threads and the environment, taking turns moving a token through the game graph.

Definition 6.1 (Trace)**.** *For threads T, initial values tm_0, and environment $\mathcal{E}$ a trace Π of the game is an sequence*

$$\ell_0, \tau_0, (T, tm_0) \xrightarrow{\tau_0} \ell_1, \bot, (T, tm_1) \xrightarrow{\mathcal{E}}$$
$$\ell_2, \tau_2, (T, tm_1) \xrightarrow{\tau_2} \ell_3, \bot, (T, tm_3) \xrightarrow{\mathcal{E}} \ldots$$

Definition 6.2 (Whole Program)**.** *For thread components $P = (T_1, tm_1), \ldots, (T_n, tm_n)$ and environment $\mathcal{E}$, the* whole program trace *denoted $\Pi(P, \mathcal{E})$ is the trace between $\ell, \tau, (T_1, tm_1) \oplus \cdots \oplus (T_n, tm_n)$ and $\mathcal{E}$.*

For two components, a trace $\Pi((T_1, tm_1) \oplus (T_2, tm_2), \mathcal{E})$ can be visualized as follows:



We lift observations to traces, say that an observation $obs_i(\Pi)$ of a trace is simply the observation $obs_i(\ell_i)$, which is the same for all steps of the trace after which the ℓ has size at least i.

Whole-program semantics. For a program $P = (T_1, tm_1) \oplus \cdots \oplus (T_n, tm_n)$ we can now define the whole-program semantics:

$$[\![P]\!]_{\mathfrak{E}} \equiv \{\Pi(P, \mathcal{E}) \mid \mathcal{E} \in \mathfrak{E}\}$$

Definition 6.3. *We say that a system $\mathfrak{E}_A$ with object implementation C_O* contextually refines *a system $\mathfrak{E}_B$ with object specification S_O written $[\![C_O]\!]_{\mathfrak{E}_A} \sqsubseteq [\![S_O]\!]_{\mathfrak{E}_B}$, if for every $\mathcal{E}_A \in \mathfrak{E}_A$ and every P, there exists $\mathcal{E}_B \in \mathfrak{E}_B$ such that $\Pi(P \oplus C_O, \mathcal{E}_A) \preccurlyeq_{\widehat{obs}} \Pi(P \oplus S_O, \mathcal{E}_B)$.*

In our setting agents C_O and S_O are defined to invoke the same base operations given the same arguments, with the difference that the threads performing these operations are allowed to yield at different locations. Therefore, the extra condition on *going wrong* in other definitions of contextual refinement, is not fruitful in our case.

We study two particular classes of environments, the *interleaved* ones (denoted $\mathfrak{E}_{\text{interleaved}}$), and the *atomic* ones (denoted $\mathfrak{E}_{\text{atomic}}$). An environment $\mathcal{E}_I$ in the former class, can schedule any thread irrespectively of which thread's action was last performed, whereas an atomic environment $\mathcal{E}_A$, will only switch threads if the last event in the log is of the form $(\tau, \triangledown)$, for some τ, and schedules the thread τ otherwise.

Theorem 6.1. *For any object O we have*

$$[\![C_O]\!]_{\textit{interleaved}} \sqsubseteq [\![S_O]\!]_{\textit{interleaved}}$$

Proof. The proof can be found in the Appendix. □

6.1 Vertical composition of contextual refinement between implementations and specifications

Theorem 6.2. *Let O and Q be two objects. Then*

$$[\![C_O \oplus C_Q]\!]_{\textit{interleaved}} \sqsubseteq [\![S_O \oplus S_Q]\!]_{\textit{interleaved}}.$$

Proof. The proof can be found in the Appendix. □

Applications. Reversible atomic objects abstract away thread implementation details, many of which are described by the Push/Pull model [28]. Specifically, in the formalism described in Section 5, there is a single global append-only shared log that is visible by all threads. Therefore, there is no need for specific *push* or *pull* rule to ferry events between logs. The essence of transactional objects is what must happen in order for a thread to commit and what happens during/after a commit, both of which are captured by reversible atomic objects. Therefore, reversible atomic objects covers all of the following implementations:

Implementation Style	**Instances**
Optimistic STM	TL2 [12], TinySTM [14], McRT [47]
Checkpoints	Herlihy & Koskinen [26]
Closed nested	LogTM [39]
Pessimistic STM	Matveev and Shavit [37]
Pessimistic objects	Boosting [19]
Irrevocable transactions	[58]
Non-opaque	Early release [21], dependent [45]
Nondetermin. choice	HaskellSTM retry/orElse [17]
Hardware TM	Intel [24]

7. Progress

In this section we describe a novel treatment of contention management as an environment that breaks deadlocks and ensures progress. The key is to use the fact that inverses are always available, that there is a common shared log, and that a priority scheme can be used that ensures the oldest transaction will commit.

Most of the information the environment needs in order to do contention is already provided by the log. However, the environment also needs to know what operations deadlocked threads would like to do. We thus augment the $(\tau, \triangledown)$ event to instead be $(\tau, \triangledown_a)$ where a is the operation that transaction τ would like to perform but currently is unable to. The environment can then cross-reference this with the uncommitted operations of other transactions, consulting the commutativity specifications for conflict.

Our use of shared logs and consistent availability of inverses means that the environment can serve as a contention manager, logically, by appending an operation inverses (τ, a^{-1}) on behalf of thread τ that generated event (τ, a). The thread τ becomes aware of this inverse by observing the log (we discuss this in more detail in Section 8). We further require that a well-formed thread will take note of these inverse operations and act appropriately.

As an example, consider the following log:

$$\begin{aligned}\ell = \; & (\tau_1, \mathsf{Ivk}\ \_), (\tau_1, a), (\tau_1, \triangledown), (\tau_2, \mathsf{Ivk}\ \_), (\tau_2, b), (\tau_2, \triangledown),\\ & (\tau_3, \mathsf{Ivk}\ \_), (\tau_3, c), (\tau_3, \triangledown), (\tau_1, \triangledown_e), (\tau_2, \triangledown_f), (\tau_3, \triangledown_g)\end{aligned}$$

the last three $\triangledown$ events indicate that threads τ_1, τ_2, τ_3 (resp.) are stuck trying to perform operations e, f, g (resp.). Let us say that a conflicts with f, b conflicts with g, and c conflicts with e. Then there is a deadlock cycle and none of $\{\tau_1, \tau_2, \tau_3\}$ are able to make progress.

We will now describe a simple *contention management* [48, 49, 51] policy that ensures that all transactions eventually terminate. We can instantiate a base environment that has a simple scheduler protocol that is able to resolve deadlocks. First, let us say that deadlocked(ℓ) is the set of deadlocked trheads and oldest(T, ℓ) indicates that thread whose Ivk event is earliest in the log. Now, we can define the environment as:

```
$\mathcal{E}_{cm}(\ell, T)$:
    let pause $\ell$ $\tau$ = rev mkSeq $\{(\tau, a^{-1}) \mid \forall a \in \mathsf{activeOps}_\tau(\ell)\}$ in

    $\lambda$ $\ell$ $T$.
      let $T'$ = deadlocked($\ell$) in
      if $T' = \varnothing$ then
        choose($\ell, T$)
      else
        let $\tau$ = oldest($T'$,$\ell$) in
          (concat [] (map (pause $\ell$) $T' \setminus \tau$),
           $\tau$)
```

This environment determines which transactions are deadlocked. If there are none, then it defaults to making a nondeterministic decision. Otherwise, it determines the oldest transaction, and inverts the operations of all other deadlocked threads by generating a sequence of events on behalf of each such thread (in the reverse order they they were generated). Finally, it marks τ as the next thread to execute.

This is overly conservative: the above contention manager may abort more transactions than necessary. Also, it may not need to abort all active operations. It could do better by considering which particular operations cause conflict for the oldest transaction. However, it is sufficient to yield provable progress guarantees and comparative analysis of conflict management strategies is beyond the scope of this paper.

Returning to the above example, $\mathcal{E}_{cm}$ may return the sequence of events $(\tau_2, b^{-1}), (\tau_3, c^{-1})$ and schedule τ_1 to execute next.

8. Implementation

The semantics given in the previous sections give rise to a novel transactional version of the universal construction, whereby sequential implementations of ADTs/data-structures can be used as concurrent, transactional objects. This transactional universal construction in turn, informs a real-world implementation of a runtime for reversible atomic objects. In addition to this runtime, we have implemented a library of reversible atomic objects. The construction and its implementation demonstrate that reversibility in our formalism is a fundamental idea.

8.1 Transactional Universal Construction

The Herlihy universal construction uses the shared log to hold all method invocations on the shared object [18]. All operations from the log, applied in sequence, is the state of the shared object. To mutate the object, threads compete to append the next operation onto the log. The winner's method invocation becomes the next log entry, and the rest keep trying.

Transactions complicate this construction in several ways. First, the log needs to communicate more information than method events. The log includes begin and commit events, each being associated with a transaction identifier. To support reversibility, the log also permits inverse operations. Second, the procedure for appending operations to the log accounts for transaction aborts and conflict. A thread's operations may be inverted underneath it, in which case the thread can no longer append the method event without first accounting for the inverse. The append interface needs to reflect this communication.

The append procedure, try_op, is the core of the transactional universal construction. It takes a transaction identifier and a method invocation from the thread (τ, a) and reports either a Success, a Conflict, or an Abort response. Success means the invocation was appended to the log. A Conflict means the invocation conflicts with an entry already on the log. Abort means some operations of the transaction have been inverted. Along with procedures to append begin and commit events, try_op provides the interface necessary for threads to support transactions on reversible atomic objects.

Figure 5 defines the UniversalTransactionalObject class. It assumes the Thread object contains a local copy of a sequential implementation of the shared object in thread.local and a reference to the last log entry seen by the thread in thread.lastSeen. Line 3 defines try_op, which takes a transaction identifier and a method invocation. After constructing a new entry (Line 5), it starts iterating over the log from thread.lastSeen until it reaches the end (Lines 10–17). This loop checks whether the transaction's new operation is permitted on the log. First, It checks whether the thread's new entry conflicts with one already on the log, according to RAO specification tx.isConflict (Lines 12–14). Second, it checks for any inversions of the current transaction by the contention manager (Lines 15–16). Because inverses are applied in reverse order, the loop continues looking for inverses until the end of the log, saving the last one found (Lines 18). An Abort or Conflict causes the loop to break early (Line 19).

When there is no conflict or abort, the thread is finally free to compete for the end of the log on Line 20. If the thread loses, the entire process begins again at Line 8. If the thread wins, however, the next node returned by decideNext is the thread's own new entry, and the loop terminates. Lastly, the thread-local copy of the sequential version of the object created by applying the operations from the log, as long as they are from committed operations and the current transaction (Lines 26–31).

With try_op defined, it is straightforward to append begin and commit events. The begin function (Line 34) creates a new transaction identifier for the thread (Line 35) and calls try_op to append the begin message (Line 36) [2]. The call cannot result in a Conflict or Abort, because there are no operations yet from the transaction. Similarly, try_commit (Line 39) uses try_op to append a commit event. In this case, however, a threads operation may have been inverted before the thread appends the commit, so Line 40 saves the response from try_op. If response is Success, the commit is recorded in a global summary for use in updating of the thread-local copy of the object (Lines 27–28). The summary is provided for convenience, and can always be recreated by traversing the log.

[2] For simplicity, begin and commit events are also represented by the Invoc type.

```
public class UniversalTransactionalObject {
  private Transaction[] committed;
  private Node logHead;
  public Result try_op(Transaction tx, Invoc invoc) {
    Thread thread = tx.thread;
    Node entry = new Node(th, tx, invoc);
    Node current = thread.lastSeen;
    do {
      Result result = null;
      while (current.next != null) {
        current = current.next;
        if (tx.isConflict(entry, current)) {
          result = new Conflict();
          break;
        } else if (tx.isInverse(current))
          abort = current;
      }
      if (abort != null) result = new Abort(abort)
      if (result != null) break;
      Node next = current.decideNext(entry);
      if (next == entry) result = new Success();
    } while (result == null);
    thread.local = new SeqObject();
    current = logHead;
    while (current != null) {
      if (committed[current.tx] or current.tx == tx)
        thread.local.apply(current.invoc);
      thread.lastSeen = current;
      current = current.next;
    }
    return result;
  }
  public Transaction begin(Thread thread, ObjectID obj) {
    Transaction tx = new Transaction(thread, obj);
    try_op(tx, begin);
    return tx;
  }
  public Result try_commit(Transaction tx) {
    Result result = try_op(tx, commit);
    if (result is Success) {
      committed[tx] = true;
    }
    return result;
  }
}
```

Figure 5. The universal construction for transactional objects.

Remark: Related work. The original universal construction is due to Herlihy [18]. Crain et al. [11] describe a universal construction for atomic read/write objects based on a specific STM setup of m processors, n processes, and some assurance of progress. However, they don't seem to unearth a general methodology for universal construction of transactional objects. There is also some similarity between multi-core universal construction and replication in distributed systems (e.g. CORFU [6], Tango [7], and state machine replication [44]).

8.2 Implementing Reversible Atomic Objects

The transactional universal construction is a convenient way to make a prototype implementation of reversible atomic objects. Intuitively, the begin and commit events appear at the beginning and end of the method, reflecting the alignment of transactions with method boundaries. Then method calls to child RAOs can be instrumented with try_op to handle conflicts, aborts, and shared communication with the log.

Specification of MoveableHashtable.put

```
1 /* conflict: $k$, $size$ */
2 put($k$, $v$) = atomic{{
3   $l_0$: $v_{old}$ := ht.get($k$) ↦ $inv_0$
4   $l_1$: ht.put($k$, $v$) ↦ $inv_1$
5   cmt_return (pess_atomic{{$inv_1$; $inv_0$}}, ⊥)
6 }}
```

Pessimistic Implementation of MoveableHashtable.put

```
1  put($k$, $v$) {
2    tx = begin(thread, mht)
3    $l_0$: $v_{old}$ := ht.get($k$)
4    Invoc invoc = ⟨ht.put($k$, $v$)⟩;
5    inverses[invoc] = ⟨ht.put$^{-1}$($k$, $v_{old}$)⟩;
6    $l_1$: $r_1$ = try_op(tx, invoc)
7    if ($r_1$ conflict or abort)
8      goto $r_1$.location
9    ht.put($k$, $v$)
10
11   $l_2$: $r_{cmt}$ = try_commit()
12   if ($r_{cmt}$ failed)
13     goto $r_{cmt}$.location
14   return
15 }
```

Optimistic Implementation MoveableHashtable.put

```
1  put($k$, $v$) {
2    tx = begin(thread, mht)
3    do {
4      local_ht = ht.copy()
5      Invoc invoc = ⟨ht.put($k$, $v$)⟩;
6      events.add(invoc)
7      inverses[tx,invoc] = ⟨ht.put$^{-1}$($k$, $v_{old}$)⟩;
8      events.addAll(local_ht.put($k$, $v$))
9      events.add(commit)
10     if (try_op_all(events))
11       return
12   } while (true)
13 }
```

Figure 6. The specification of MoveableHashtable.put with pessimistic and optimistic implementations using the transactional universal construction.

Figure 6 demonstrates how our implementation works. We first show the specification of MoveableHashtable.put. It follows the same convention as the rest of the FileSystem example from Figure 1. In an atomic section, the method first calls get and then put, ending the transaction and returning the inverse and return value.

The second program listing in Figure 6 shows the pessimistic implementation of MoveableHashtable.put. Like the specification, each child object invocation has a label l_i, used to retry the transaction. After beginning the transaction (Line 2) and getting the old value of k (Line 3), Line 4 creates an Invoc object for the child object method call. The brackets around the method ⟨ht.put(k, v)⟩

RAO Feature	Strategy
Aborts and conflicts	gotos, continuations
Method-local state	snapshots, continuations
Representing inverses	function pointers, closures
Conflict detection	try_op, spinlocks

Table 1. Features of reversible atomic objects and their implementation strategies.

distinguish the representation of the invocation from the method call itself. In practice, an invocation can be represented with an opcode and argument list. Similarly, Line 5 creates the inverse operation from a manually implemented inverse method, ht.put^{-1}. This is stored in a global table, inverses, for use when aborting or retrying the transaction.

With the invocation and its inverse, try_op attempts to append the operation to the log (Line 6). Lines 8–9 handle aborts and conflicts by retrying part of the transaction from a given location l_i. For a more complex example, the jump would also need to restore any thread-local state, e.g., local variables.

The last program listing in Figure 6 is the optimistic implementation. The difference from pessimistic comes down to how eagerly the method calls try_op. The pessimistic implementation calls try_op as early as possible. The optimistic one, however, performs all operations locally on a deep copy of the child object (Line 4), collects operations in a local list (Lines 6–9), and attempts to append them all at once at the end of the method (Lines 10–13).

In this example, the log is the sole tool for detecting conflicts, but in general, the thread is not bound by any specific conflict handling technique. One example conflict detection technique is explained by Dimitrov et al. [13]. Threads may use other conflict detection mechanisms, such as locks, in addition to the log, as long as the reversible atomic object specification is met. Threads may also wait, watching the log for the conflicting operation to be committed or inverted, or invert and retry part of its own transaction [17].

The semantics of reversible atomic objects and the transactional universal construction permit a wide array of implementation design choices. Furthermore, our experience implementing them by hand reveals the potential for automation, where a compiler or runtime system takes the serial implementation of an RAO and generates a concurrent version using the transactional universal construction.

Table 1 sketches the key features of reversible atomic objects and potential implementation strategies. To handle retrying a transaction after an abort or a conflict, our implementation example above used gotos. Continuations may be used for retry and to handle the method-local state saving needed when retrying. Inverses can be represented with closures, as seen in ScalaSTM [1]. As discussed above, conflict detection need not rely on the log and can use other techniques, e.g., spinlocks.

8.3 Library of Reversible Atomic Objects

We have implemented a Java framework based on the transactional universal construction and a library of reversible atomic objects built on it. The framework provides a shared log implementation and the ObjectDefinition superclass. ObjectDefinition provides the try_op methods to interact with the log along with abstract methods for specifying RAOs as subclasses. The library includes the complete RAOs from Figure 1 along with a multi-threaded implementation of the example from Figure 2.

The framework provides everything needed to implement reversible atomic objects. The Log is a linked-list that uses compare-and-swap to append a new entry. The ThreadID provides access to local object copies, while ObjectID identifies shared objects in the log, independent of any thread. The ObjectDefinition superclass is the heart of the framework. All RAOs subclass it, implementing the apply method for updating thread-local objects and the isConflict method. RAOs method are implemented following the pattern in Figure 6, albeit with a few simplifications. First, the RAO methods restart transactions from the beginning when there is a conflict, although ObjectDefinition provides support for saving and performing inverses. Second, there is no contention manager implementation. Lastly, inverses are manually specified rather than generated from return values.

The RAO library contains the Hashtable base object, MoveableHashtable, DirectoryTree, and FileSystem. The DirectoryTree stores directory contents with the Hashtable, mapping directory paths to a list of contents. The example program has three threads, one to generate files `/file1` to `/file100`, another thread to move even-numbered files to the `/evens` directory, and third to repeatedly print the number of existing files. The program finishes by printing the directories and contents of the FileSystem object.

Discussion. Building the reversible atomic library prototype lead to a few interesting observations. After implementing the framework, the library objects were straightforward to implement, becoming rote. The base objects are simple wrapper that provide a conflict specification. By following the pattern of Figure 6, the upper layer objects became systematic to implement, constructing operations and calling try_op. This leads us to believe it is possible create reversible atomic objects automatically by, for instance, compiling a serial specification into an RAO.

Another observation is that only the base objects, such Hashtable, require an apply implementation. Only these methods mutate shared state. The upper layers have no state themselves; rather they specify how the base object operations are organized and scheduled. This is why only base objects need an apply method.

9. Related Work

Nested transactions. Nested transactions originated in the database community [57] and, since 2006 [41], they have also appeared in software. Because reversible atomic object operations may be implemented as transactions, there is an inherent nesting in our model. In this sense, reversible atomic objects are an instance of nested transactions.

However, not all nested transactions are reversible atomic objects. Unlike nested transactions, reversible atomic objects require that transactions be aligned with method boundaries. Also, reversible atomic objects construct their own inverses (and we show that they are typically easy to construct). Finally, reversible atomic objects may, at each level, use different implementation styles (pessimistic-vs-optimistic). To the best of our knowledge, this is not possible with nested transactions. An example of a nested transaction that is not a reversible atomic object is given in Figure 7. There are two class definitions and a vertical OO hierarchy. Transactions are not directly aligned with object methods. Conflict/commutativity/inverses are used in an ad hoc manor (not shown here).

There are several key consequences of reversible atomic objects, in relation to nested transactions:

- *Clearer semantics.* The semantics of nested transactions is quite complex [40, 57] and, perhaps consequently, to date there has not been much formal backbone developed in this area. By contrast, reversible atomic objects have a semantics that is simple and clear, while also permitting great complexity, variety of implementation choices, and vertical composition (see Sections 3-7).
- *Progress.* Since we require inverses to consistently be available, a contention managing environment can be used to abort transactions. Using a priority scheme, progress can be ensured.

```
FileSystem.moveFile(p1, p2) {
  atomic {
    v = ht.get(p1)
    ht.put(p2, v)
    ht.remove(p1)
  }
  directory.move(p1, p2);
}

Directory.move(p1, p2) {
  atomic {
    all_paths.add(p2)
    all_paths.remove(p1)
    atomic {
      Node node = tree.find(p2.directory)
      node.addChild(p2.filename)
    }
    atomic {
      Node node = tree.find(p1.directory)
      node.removeChild(p1.filename)
    }
  }
}
```

Figure 7. An example of nested transactions that *aren't* reversible atomic objects.

- *Ease of implementation.* Reversible atomic objects make it easier for a layperson to build complex concurrent transactional systems. Many details are abstracted away from the programmer and can be accomplished by a compiler or non-monolithic runtime systems.
- *Automation.* The transactional universal construction makes automatic generation of reversible atomic objects possible from a serial specification.

STM data-structure implementations. Two recent works have aimed at developing real-world efficient implementations of transactional data-structures. Herman et al. [23] recently described a way of implementing transactional data-structures. They build on top of a core infrastructure that provides operations on version numbers and abstract tracking sets that can be used to make object-specific decisions at commit time. Similar work by Spiegelman et al. [3] describes how to build data-structure libraries using traditional STM read/write tracking primitives. In this way, the implementation can exploit these STM internals. These data-structures can combine pessimistic and optimistic implementations. This strategy is appropriate for STM experts, but does not seem to provide a general theory and/or framework for vertical composition.

Other works. The recent Push/Pull model provided a formal semantics for describing a range of transactional implementations [28]. At a technical level, the Push/Pull model uses thread-local logs for describing detailed thread-local behavior. We abstract away these details. The Push/Pull model modifies the global log by removing entries, where as our semantics is append-only.

The key distinction is that the Push/Pull model does not investigate how transactional objects can be composed, nor does it provide contextual refinement results or liveness guarantees. However, as discussed at the end of Section 6, reversible atomic objects was designed so that it still captures the range of implementations covered by Push/Pull.

Many have formalized correctness criteria of various STM implementations. Recently, it was shown that TMS is equivalent to contextual refinement [5] for the case where shared and local variables are rolled back when a transaction aborts.

Others [32] describe a method of specifying and verifying TM algorithms. They specify some transactional algorithms in terms of I/O automata [35] and this choice of language enables them to fully verify those specifications in PVS.

Ziv et al. [59] describe how to compose transactions with other kinds of concurrency control such as two-phase locking and two-phase commit.

More distant are works that address the privatization problem [2, 38, 52], dynamic/static/hybrid atomicity [56], message passing within transactions [33], proof engineering [34], and commutativity [8, 9, 25, 31, 46, 50, 53, 56].

10. Conclusions and Future Work

We have described a model for vertical composition of transactional objects that is semantically simple and yet expressive and amenable to implementation flexibility. In our model, abstract-level operations are composed from constituent base operations, accounting for conflict and ensuring availability of inverses. These transactional implementations are put in the context of an environment that includes a novel deadlock-mitigating contention manager that ensures progress. Our model is the first proof of contextual refinement and vertical composition for transactional objects. We believe that reversible atomic objects provide a feasible avenue toward a broader availability of composable transactional objects.

There are several areas for future work. We intend to investigate recursion in the context of reversible atomic objects. We will also develop more mature versions of our implementations that can be used in realistic settings, where a performance evaluation can be made. Finally, we will automate the translation from user-level syntax to pessimistic and optimistic implementations.

References

[1] ScalaSTM. `https://nbronson.github.io/scala-stm/`.

[2] Abadi, M., Birrell, A., Harris, T., and Isard, M. Semantics of transactional memory and automatic mutual exclusion. In *The 35th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL'08)* (2008), pp. 63–74.

[3] Alexander Spiegelman, Guy Golan-Gueta, I. K. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)* (2016).

[4] Amit, D., Rinetzky, N., Reps, T. W., Sagiv, M., and Yahav, E. Comparison under abstraction for verifying linearizability. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings* (2007), pp. 477–490.

[5] Attiya, H., Gotsman, A., Hans, S., and Rinetzky, N. Safety of live transactions in transactional memory: TMS is necessary and sufficient. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings* (2014), F. Kuhn, Ed., vol. 8784 of *Lecture Notes in Computer Science*, Springer, pp. 376–390.

[6] Balakrishnan, M., Malkhi, D., Davis, J. D., Prabhakaran, V., Wei, M., and Wobber, T. CORFU: A distributed shared log. *ACM Trans. Comput. Syst. 31*, 4 (2013), 10.

[7] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. Tango: distributed data structures over a shared log. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 325–340.

[8] Beeri, C., Bernstein, P., Goodman, N., Lai, M.-Y., and Shasha, D. A concurrency control theory for nested transactions (preliminary report). In *Proceedings of the 2nd annual ACM symposium on Principles of distributed computing (PODC'83)* (New York, NY, USA, 1983), ACM Press, pp. 45–62.

[9] Bernstein, A. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers 15*, 5 (1966), 757–763.

[10] Chen, H., Wu, X. N., Shao, Z., Lockerman, J., and Gu, R. Toward compositional verification of interruptible os kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016), ACM, pp. 431–447.

[11] Crain, T., Imbs, D., and Raynal, M. Towards a universal construction for transaction-based multiprocess programs. In *International Conference on Distributed Computing and Networking* (2012), Springer, pp. 61–75.

[12] Dice, D., Shalev, O., and Shavit, N. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)* (September 2006).

[13] Dimitrov, D., Raychev, V., Vechev, M., and Koskinen, E. Commutativity race detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14), Edinburgh, UK* (2014).

[14] Felber, P., Fetzer, C., and Riegel, T. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)* (2008), pp. 237–246.

[15] Filipović, I., OHearn, P., Rinetzky, N., and Yang, H. Abstraction for concurrent objects. *Theoretical Computer Science 411*, 51 (2010), 4379–4398.

[16] Guerraoui, R., and Kapalka, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)* (2008), ACM, pp. 175–184.

[17] Harris, T., Marlow, S., Jones, S. L. P., and Herlihy, M. Composable memory transactions. *Commun. ACM 51*, 8 (2008), 91–100.

[18] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13*, 1 (1991), 124–149.

[19] Herlihy, M., and Koskinen, E. Transactional boosting: A methodology for highly concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)* (2008).

[20] Herlihy, M., and Koskinen, E. Composable transactional objects: A position paper. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings* (2014), pp. 1–7.

[21] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of distributed computing (PODC'03)* (2003), pp. 92–101.

[22] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[23] Herman, N., Inala, J. P., Huang, Y., Tsai, L., Kohler, E., Liskov, B., and Shrira, L. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016* (2016), C. Cadar, P. Pietzuch, K. Keeton, and R. Rodrigues, Eds., ACM, pp. 31:1–31:16.

[24] Intel. Transactional synchronization in haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-syn`

[25] Korth, H. F. Locking primitives in a database system. *J. ACM 30*, 1 (1983), 55–79.

[26] Koskinen, E., and Herlihy, M. Checkpoints and continuations instead of nested transactions. In *Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08)* (2008), pp. 160–168.

[27] Koskinen, E., and Herlihy, M. Dreadlocks: efficient deadlock detection. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures (SPAA'08)* (New York, NY, USA, 2008), ACM, pp. 297–303.

[28] Koskinen, E., and Parkinson, M. J. The push/pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (2015), D. Grove and S. Blackburn, Eds., ACM, pp. 186–195.

[29] Koskinen, E., Parkinson, M. J., and Herlihy, M. Coarse-grained transactions. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)* (2010), ACM, pp. 19–30.

[30] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)* (2007), pp. 211–222.

[31] Lam, M., and Rinard, M. Coarse-grain parallel programming in Jade. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'91)* (1991), ACM New York, NY, USA, pp. 94–105.

[32] Lesani, M., Luchangco, V., and Moir, M. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR'12)* (2012), vol. 7454, pp. 516–530.

[33] Lesani, M., and Palsberg, J. Communicating memory transactions. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'11)* (2011), pp. 157–168.

[34] Lesani, M., and Palsberg, J. Decomposing opacity. In *Proceedings of the 28th International Symposium on Distributed Computing (DISC'14)* (2014), pp. 391–405.

[35] Lynch, N. A., and Tuttle, M. R. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)* (1987), pp. 137–151.

[36] Maldonado, W., Marlier, P., Felber, P., Suissa, A., Hendler, D., Fedorova, A., Lawall, J. L., and Muller, G. Scheduling support for transactional memory contention management. In *ACM Sigplan Notices*, vol. 45, pp. 79–90.

[37] Matveev, A., and Shavit, N. Towards a fully pessimistic STM model. In *Proceedings of the 2012 Workshop on Transactional Memory (TRANSACT12)* (2012).

[38] Moore, K. F., and Grossman, D. High-level small-step operational semantics for transactions. In *Proceedings of the 35th annual*

ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'08) (2008), pp. 51–62.

[39] MORAVAN, M. J., BOBBA, J., MOORE, K. E., YEN, L., HILL, M. D., LIBLIT, B., SWIFT, M. M., AND WOOD, D. A. Supporting nested transactional memory in logTM. *SIGOPS Operating Systems Review 40*, 5 (2006), 359–370.

[40] MOSS, J. E. B. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues* (2006), vol. 28.

[41] MOSS, J. E. B., AND HOSKING, A. L. Nested transactional memory: model and architecture sketches. *Science of Computer Programming 63*, 2 (2006), 186–201.

[42] NI, Y., MENON, V. S., ADL-TABATABAI, A.-R., HOSKING, A. L., HUDSON, R. L., MOSS, J. E. B., SAHA, B., AND SHPEISMAN, T. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'07)* (2007), pp. 68–78.

[43] O'HEARN, P. W., RINETZKY, N., VECHEV, M. T., YAHAV, E., AND YORSH, G. Verifying linearizability with hindsight. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010* (2010), pp. 85–94.

[44] PEDONE, F., GUERRAOUI, R., AND SCHIPER, A. The database state machine approach. *Distributed and Parallel Databases 14*, 1 (2003), 71–98.

[45] RAMADAN, H. E., ROY, I., HERLIHY, M., AND WITCHEL, E. Committing conflicting transactions in an stm. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)* (2009), pp. 163–172.

[46] RINARD, M., AND LAM, M. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems (TOPLAS) 20*, 3 (1998), 483–545.

[47] SAHA, B., ADL-TABATABAI, A.-R., HUDSON, R. L., MINH, C. C., AND HERTZBERG, B. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)* (2006), pp. 187–197.

[48] SCHERER III, W. N., AND SCOTT, M. L. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java programs* (2004), pp. 70–79.

[49] SCHERER III, W. N., AND SCOTT, M. L. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing* (2005), ACM, pp. 240–248.

[50] SCHWARZ, P. M., AND SPECTOR, A. Z. Synchronizing shared abstract types. *ACM Transactions on Computer Systems 2*, 3 (1984), 223–250.

[51] SPEAR, M. F., DALESSANDRO, L., MARATHE, V. J., AND SCOTT, M. L. A comprehensive strategy for contention management in software transactional memory. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 141–150.

[52] SPEAR, M. F., MARATHE, V. J., DALESSANDRO, L., AND SCOTT, M. L. Privatization techniques for software transactional memory. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)* (2007), pp. 338–339.

[53] STEELE, JR, G. L. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'90)* (New York, NY, USA, 1990), ACM Press, pp. 218–231.

[54] VAFEIADIS, V. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[55] VAFEIADIS, V., HERLIHY, M., HOARE, T., AND SHAPIRO, M. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006* (2006), pp. 129–136.

[56] WEIHL, W. E. Data-dependent concurrency control and recovery (extended abstract). In *Proceedings of the 2nd annual ACM symposium on Principles of Distributed Computing (PODC'83)* (1983), ACM Press, pp. 63–75.

[57] WEIKUM, G., AND SCHEK, H.-J. Concepts and applications of multilevel transactions and open nested transactions, 1992.

[58] WELC, A., SAHA, B., AND ADL-TABATABAI, A.-R. Irrevocable transactions and their applications. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'08)* (2008), ACM, pp. 285–296.

[59] ZIV, O., AIKEN, A., GOLAN-GUETA, G., RAMALINGAM, G., AND SAGIV, M. Composing concurrency control. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (2015), pp. 240–249.

A. Appendix

Proofs for Section 6

A log is inverse-free, if it contains no event that is an inverse of another event in the log. A log ℓ is (τ, a)-free, if it contians events neither of the form (τ, a) nor of the form (τ, a^{-1}). Given a log $\ell \in \mathcal{L}$, the function $\mathsf{ki}(\ell)$ is defined to be such that $\mathsf{ki}(\ell) = \ell$ when ℓ is inverse-free and $\mathsf{ki}(\ell_1 \cdot (\tau, a) \cdot \ell_2 \cdot (\tau, a^{-1}) \cdot \ell_3) = \mathsf{ki}(\ell_1 \cdot \ell_2 \cdot \ell_3)$ when ℓ_2 is (τ, a)-free.

A thread τ is *committed* in a log ℓ if $\ell = \ell_1 \cdot (\tau, \mathsf{CmtRet}\_) \cdot \ell_2$ for some logs ℓ_1, ℓ_2 when ℓ_2 contains no $(\tau, \mathsf{Ivk}\ \_)$ event and it is *uncommitted* if not. A log ℓ is *uncommitted-ordered* if there are no thread identifiers τ, τ' such that τ is committed in ℓ, τ' is uncommitted in ℓ and $\ell = \ell_1 \cdot (\tau', e') \cdot \ell_2 \cdot (\tau, e) \cdot \ell_3$ for logs ℓ_1, ℓ_2, ℓ_3. Finally, a log ℓ is *thread-method-ordered*, if it is uncommitted-ordered and it is a sequence of abstract operation sequences and base operations. In other words, ℓ is equal to $\ell_1 \cdots \ell_n$, where for each $i \leq n$, there exists τ_i such that $\ell_1 = (\tau, a)$ or $\mathsf{aos}_{\tau_i}(\ell_i, O.f)$, for some $O.f$.

Given a log ℓ, we say that $\ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\vec{x}))$ is invoked before $\ell_2 \cdot (\tau, \mathsf{Ivk} Q.g(\vec{x}))$ in ℓ, if $\ell = \ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\vec{x})) \cdot \ell_1' \cdot (\tau, \mathsf{Ivk}\ Q.g(\vec{x})) \cdot \ell_3$ and $\ell_2 = \ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\vec{x})) \cdot \ell_1'$. Furthermore, given a log ℓ, we say that an event (τ, e) *belongs* to $\ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\vec{x}))$ if $\ell = \ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\vec{x})) \cdot \ell_2 \cdot (\tau, e) \cdot \ell_3$ and ℓ_2 does not contain an event $(\tau, \mathsf{CmtRet}\ O.f(\vec{x}))$. Finally, a log segment ℓ_1 in a log ℓ *should be before* another log segment ℓ_2 in that log, if the constituent events of ℓ_1 belong to $\ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\vec{x}))$, the constituent events of ℓ_2 belong to $\ell_2 \cdot (\tau', \mathsf{Ivk}\ Q.g(\vec{x}))$ and $\ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\vec{x}))$ is invoked before $\ell_2 \cdot (\tau, \mathsf{Ivk}\ Q.g(\vec{x}))$.

The function $\mathsf{ro}(\ell)$ is defined to be such that $\mathsf{ro}(\ell) = \ell$ when ℓ is thread-method-ordered and $\mathsf{ro}(\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4) = \mathsf{ro}(\ell_1 \cdot \ell_3 \cdot \ell_2 \cdot \ell_4)$ when ℓ_3 should be before ℓ_2 in ℓ.

Given a committed-ordered log $\ell = \ell_1 \cdot \ell_2$, where ℓ_1 comprises all the committed operations and ℓ_2 comprises all the uncommitted ones, we define $\mathsf{tr}(\ell)$ to be equal to ℓ_1.

Lemma A.1. *For all* ℓ, ℓ', $\ell' \preccurlyeq_{obs} \ell \ \Rightarrow\ \ell' \preccurlyeq_{\widehat{obs}} \ell$.

Proof. Suppose that ℓ and ℓ' are two logs such that $\ell \preccurlyeq_{obs} \ell'$. Therefore, $\mathsf{allowed}(\ell) \Rightarrow \mathsf{allowed}(\ell')$ and by definition of $\widehat{\mathsf{allowed}}$, it holds that $\widehat{\mathsf{allowed}}(\ell) \Rightarrow \widehat{\mathsf{allowed}}(\ell')$. Furthermore, we know that for any event (τ, e), $\ell \cdot (\tau, e) \preccurlyeq_{obs} \ell' \cdot (\tau, e)$. Consequently, for any log ℓ_1 we have that $\ell \cdot \ell_1 \preccurlyeq_{obs} \ell' \cdot \ell_1$.

We want to show that $\widehat{\mathsf{allowed}}(\ell) \Rightarrow \widehat{\mathsf{allowed}}(\ell')$ and that for any log ℓ_1, there exists log ℓ_2, such that $\ell \cdot \ell_1 \preccurlyeq_{\widehat{obs}} \ell' \cdot \ell_2$. In particular, for a given log ℓ_1, we let $\ell_2 = \ell_1$ and we instead show that $\ell \cdot \ell_1 \preccurlyeq_{\widehat{obs}} \ell' \cdot \ell_1$. The latter follows from the fact that $\ell \cdot \ell_1 \preccurlyeq_{obs} \ell' \cdot \ell_1$ and therefore $\ell \preccurlyeq_{\widehat{obs}} \ell'$. □

Lemma A.2. *For all well-formed logs* ℓ, $\ell \preccurlyeq_{obs} ki(\ell)$.

Proof. Let ℓ be a well-formed log. We proceed by induction on the number n of inverses in ℓ. For the base case, suppose that there are no inverses. By definition, $\mathsf{ki}(\ell) = \ell$, and by reflexivity $\ell \preccurlyeq_{obs} \mathsf{ki}(\ell)$. Suppose then that the statement is correct for all logs with n inverses, where $n < K$ for some $K \in \mathbf{N}$, and consider a log ℓ with K inverses. Then there exist a basic operation a, a thread τ and logs ℓ_1, ℓ_2, ℓ_3, such that ℓ_2 is (τ, a)-free and $\ell = \ell_1 \cdot (\tau, a) \cdot \ell_2 \cdot (\tau, a^{-1}) \cdot \ell_3$. By definition, $\mathsf{ki}(\ell) = \mathsf{ki}(\ell_1 \cdot \ell_2 \cdot \ell_3)$, and by the inductive hypothesis, $\ell_1 \cdot \ell_2 \cdot \ell_3 \preccurlyeq_{obs} \mathsf{ki}(\ell_1 \cdot \ell_2 \cdot \ell_3)$. It remains to be shown that $\ell \preccurlyeq_{obs} \ell_1 \cdot \ell_2 \cdot \ell_3$, or in other words, $\ell_1 \cdot (\tau, a) \cdot \ell_2 \cdot (\tau, a^{-1}) \cdot \ell_3 \preccurlyeq_{obs} \ell_1 \cdot \ell_2 \cdot \ell_3$.

We show this by induction on the size of the length of ℓ_2. For the base case, suppose that $|\ell_2| = 0$. Then $\ell_1 \cdot \ell_2 \cdot \ell_3 = \ell_1 \cdot \ell_3$ and $\ell_1 \cdot (\tau, a) \cdot (\tau, a^{-1}) \cdot \ell_3 \preccurlyeq_{obs} \ell_1 \cdot \ell_3$. Suppose then that the statement holds for all $m < M$ for some $M \in \mathbf{N}$, and consider the case where $|\ell_2| = M$. Then $\ell_2 = \ell_2' \cdot (\tau', e)$. Since ℓ is well-formed, it follows that $(\tau', e) \overset{\ell''}{\lhd} (\tau, a^{-1})$, for $\ell'' = \ell_1 \cdot (\tau, a) \cdot \ell_2'$, and therefore, $\ell_1 \cdot (\tau, a) \cdot \ell_2' \cdot (\tau', e) \cdot (\tau, a^{-1}) \preccurlyeq_{obs} \ell_1 \cdot (\tau, a) \cdot \ell_2' \cdot (\tau, a^{-1}) \cdot (\tau', e)$. Hence,

$$\ell_1 \cdot (\tau, a) \cdot \ell_2' \cdot (\tau', e) \cdot (\tau, a^{-1}) \cdot \ell_3 \preccurlyeq_{obs} \ell_1 \cdot (\tau, a) \cdot \ell_2' \cdot (\tau, a^{-1}) \cdot (\tau', e) \cdot \ell_3.$$

Then $|\ell_2'| < M$ and by the inductive hypothesis,

$$\ell_1 \cdot (\tau, a) \cdot \ell_2' \cdot (\tau, a^{-1}) \cdot (\tau', e) \cdot \ell_3 \preccurlyeq_{obs} \ell_1 \cdot \ell_2 \cdot \ell_3,$$

and hence, by transitivity of $\preccurlyeq_{obs}$,

$$\ell_1 \cdot (\tau, a) \cdot \ell_2' \cdot (\tau', e) \cdot (\tau, a^{-1}) \cdot \ell_3 \preccurlyeq_{obs} \ell_1 \cdot \ell_2 \cdot \ell_3,$$

where the left hand side is equal to ℓ. □

Lemma A.3. *For all well-formed* ℓ, $\ell \preccurlyeq_{obs} ro(\ell)$.

Proof. Let ℓ be a well-formed log. We proceed by induction on the number n of pairs of events $((\tau, e), (\tau', e'))$ where (τ, e) should be before (τ', e') in ℓ and where (τ', e') appears before (τ, e) in ℓ. For the base case, suppose that $n = 0$. Then ℓ is thread-method-ordered and therefore, $\mathsf{ro}(\ell) = \ell$. By reflexivity, $\mathsf{ro}(\ell) \preccurlyeq_{obs} \ell$. Suppose then that the statement holds for all $n < N$, for some $N \in \mathbf{N}$, and consider the case where the number of events satisfying the above condition is N. Then $\ell = \ell_1 \cdot (\tau', e') \cdot \ell_2 \cdot (\tau, e) \cdot \ell_3$, where (τ, e) should be before (τ', e') in ℓ. It follows that $\mathsf{ro}(\ell) = \mathsf{ro}(\ell_1 \cdot (\tau', e') \cdot \ell_2 \cdot (\tau, e) \cdot \ell_3)$ and by definition, the latter is equal to $\mathsf{ro}(\ell_1 \cdot (\tau, e) \cdot (\tau', e') \cdot \ell_2 \cdot \ell_3)$. By the inductive hypothesis, $\ell_1 \cdot (\tau, e) \cdot (\tau', e') \cdot \ell_2 \cdot \ell_3 \preccurlyeq_{obs} \mathsf{ro}(\ell_1 \cdot (\tau, e) \cdot (\tau', e') \cdot \ell_2 \cdot \ell_3)$. It remains to be shown that

$$\ell \preccurlyeq_{obs} \ell_1 \cdot (\tau, e) \cdot (\tau', e') \cdot \ell_2 \cdot \ell_3.$$

Notice, that since ℓ is well-formed, it must be the case that $(\tau, e) \overset{\ell_1}{\lhd} (\tau'', a)$ for all (τ, a) in $(\tau', e) \cdot \ell_2$. By induction over the length of $(\tau', e) \cdot \ell_2$, similarly to the proof of Lemma A.2, it follows that $\ell_1 \cdot (\tau, e) \cdot (\tau', e') \cdot \ell_2 \preccurlyeq_{obs} \ell_1 \cdot (\tau', e') \cdot (\tau, e) \cdot \ell_2$, as required. □

Lemma A.4. *For any log* ℓ *that is inverse-free and committed-ordered,* $\ell \preccurlyeq_{\widehat{obs}} tr(\ell)$.

Proof. By reflexivity of $\preccurlyeq_{\widehat{obs}}$, we know that for all logs ℓ, $\ell \preccurlyeq_{\widehat{obs}} \ell$. Let $\ell = \mathsf{tr}(\ell) \cdot \ell'$ and let ℓ_1 be any log. By definition of the predicate $\widehat{\mathsf{allowed}}$, it holds that $\widehat{\mathsf{allowed}}(\ell) \Rightarrow \widehat{\mathsf{allowed}}(\mathsf{tr}(\ell))$. Let $\ell_2 = \ell' \cdot \ell_1$. Then $\ell \cdot \ell_1 = \mathsf{tr}(\ell) \cdot \ell' \cdot \ell_1 = \mathsf{tr}(\ell) \cdot \ell_2$ and therefore, $\ell \cdot \ell_1 \preccurlyeq_{\widehat{obs}} \mathsf{tr}(\ell) \cdot \ell_2$, as required. □

Lemma A.5. *For all* ℓ, ℓ', ℓ'',

$$\textit{if } \ell \preccurlyeq_{\widehat{obs}} \ell' \textit{ and } \ell' \preccurlyeq_{\widehat{obs}} \ell'' \textit{ then } \ell \preccurlyeq_{\widehat{obs}} \ell''.$$

Proof. Suppose that $\ell \preccurlyeq_{\widehat{obs}} \ell'$ and $\ell' \preccurlyeq_{\widehat{obs}} \ell''$. Then $\mathsf{allowed}(\ell)$ implies $\mathsf{allowed}(\ell')$ and $\mathsf{allowed}(\ell')$ implies $\mathsf{allowed}(\ell'')$. It follows that $\mathsf{allowed}(\ell) \Rightarrow \mathsf{allowed}(\ell'')$. We know that for all ℓ_1, there exists ℓ_2 such that $\ell \cdot \ell_1 \preccurlyeq_{\widehat{obs}} \ell' \cdot \ell_2$. Furthermore, we know that given ℓ_2, there exists ℓ_3 such that $\ell' \cdot \ell_2 \preccurlyeq_{\widehat{obs}} \ell'' \cdot \ell_3$. Therefore, for all ℓ_1, there exists ℓ_3 such that $\ell \cdot \ell_1 \preccurlyeq_{\widehat{obs}} \ell' \cdot \ell_3$, as required. □

Lemma A.6. *For all programs* P *and* $\mathcal{E}_I \in \mathfrak{E}_{interleaved}$ *there exists* $\mathcal{E}_A \in \mathfrak{E}_{atomic}$ *such that*

$$tr(ro(ki(log(\Pi(P, \mathcal{E}_I))))) \preccurlyeq_{obs} log(\Pi_A(P, \mathcal{E}_A)).$$

Proof. We want to construct a valid $\mathcal{E}_A$ that simply schedules the appropriate threads. Let P be any program, and let $\mathcal{E}_I$ be any interleaved environment. Let ℓ be equal to $\log(\Pi(P, \mathcal{E}_I))$ and ℓ_{norm} be $\mathsf{tr}(\mathsf{ro}(\mathsf{ki}(\ell)))$. By definition of the transformations $\mathsf{tr}(.)$, $\mathsf{ro}(.)$ and $\mathsf{ki}(.)$, it follows that $\mathsf{tr}(\mathsf{ro}(\mathsf{ki}(\ell)))$ is thread-method-ordered, without inverses and all operations that appear in it are committed. We then define $\mathcal{E}_A$ to be simply the environment that schedules the threads and methods according to the order they appear in this normalized version of the log ℓ. □

Lemma A.7. *For all objects O, and environments $\mathcal{E}_I$ in $\mathfrak{E}_{interleaved}$, there exists $\mathcal{E}_A$ in $\mathfrak{E}_{atomic}$ such that for all client programs P, $\Pi(P \oplus C_O, \mathcal{E}_A) = \Pi(P \oplus S_O, \mathcal{E}_I)$. Similarly, for all $\mathcal{E}_A$ in $\mathfrak{E}_{atomic}$, there exists $\mathcal{E}_I$ in $\mathfrak{E}_{interleaved}$, such that for all P, $\Pi(P \oplus C_O, \mathcal{E}_A) = \Pi(P \oplus S_O, \mathcal{E}_I)$.*

Proof. By definition of $\mathfrak{E}_{\text{interleaved}}$ and $\mathfrak{E}_{\text{atomic}}$. □

Theorem 6.1. *For any object O we have*

$$[\![C_O]\!]_{interleaved} \sqsubseteq [\![S_O]\!]_{interleaved}$$

Proof. We want to show that for every client program P and $\mathcal{E}_I \in \mathfrak{E}_{\text{interleaved}}$ there exists $\mathcal{E}''_I \in \mathfrak{E}_{\text{interleaved}}$ such that $\Pi(P \oplus C_O, \mathcal{E}_I) \preccurlyeq_{\widehat{obs}} \Pi(P \oplus S_O, \mathcal{E}''_I)$.

Notice that by Lemma A.7, it suffices to show that for all P and $\mathcal{E}_I \in \mathfrak{E}_{\text{interleaved}}$, there exists $\mathcal{E}_A \in \mathfrak{E}_{\text{atomic}}$, such that $\Pi(P \oplus C_O, \mathcal{E}_I) \preccurlyeq_{\widehat{obs}} \Pi(P \oplus C_O, \mathcal{E}_A)$. Fix a program P and an interleaved environment $\mathcal{E}_I$, and let $\mathcal{E}_A$ be the atomic environment obtained by Lemma A.6. For readability, let ℓ_I be $\log(\Pi(P \oplus C_O, \mathcal{E}_I))$ and let ℓ_A be $\Pi(P \oplus C_O, \mathcal{E}_A)$. We have that $\mathsf{tr}(\mathsf{ro}(\mathsf{ki}(\ell_I))) \preccurlyeq_{obs} \ell_A$, and by Lemma A.1, it holds that $\mathsf{tr}(\mathsf{ro}(\mathsf{ki}(\ell_I))) \preccurlyeq_{\widehat{obs}} \ell_A$.

Since ℓ_I is assumed to be a well-formed log, by Lemmas A.2 and A.3, $\ell_I \preccurlyeq_{obs} \mathsf{ro}(\mathsf{ki}(\ell_I))$ and by Lemma A.1, $\ell_I \preccurlyeq_{\widehat{obs}} \mathsf{ro}(\mathsf{ki}(\ell_I))$. Furthermore, by Lemma A.4, $\mathsf{ro}(\mathsf{ki}(\ell_I)) \preccurlyeq_{\widehat{obs}} \mathsf{tr}(\mathsf{ro}(\mathsf{ki}(\ell_I)))$, and thus, $\ell_I \preccurlyeq_{\widehat{obs}} \mathsf{tr}(\mathsf{ro}(\mathsf{ki}(\ell_I)))$, by transitivity of $\preccurlyeq_{\widehat{obs}}$ (Lemma A.5). Finally, since $\mathsf{tr}(\mathsf{ro}(\mathsf{ki}(\ell_I))) \preccurlyeq_{\widehat{obs}} \ell_A$ and $\ell_I \preccurlyeq_{\widehat{obs}} \mathsf{tr}(\mathsf{ro}(\mathsf{ki}(\ell_I)))$, again by Lemma A.5 we obtain $\ell_I \preccurlyeq_{\widehat{obs}} \ell_A$, as required. □

Lemma A.8. *For any object specification S_O, program P, and environment $\mathcal{E} \in \mathfrak{E}_{interleaved}$ it holds that the base observations of $\Pi(P, \mathcal{E})$ are equal to the abstract observations of $\Pi(P \oplus S_O, \mathcal{E})$.*

Theorem 6.2. *Let O and Q be two objects. Then*

$$[\![C_O \oplus C_Q]\!]_{interleaved} \sqsubseteq [\![S_O \oplus S_Q]\!]_{interleaved}.$$

Proof. From Theorem 6.1, we know that (a) $[\![C_O]\!]_{\text{interleaved}} \sqsubseteq [\![S_O]\!]_{\text{interleaved}}$ and (b) $[\![C_Q]\!]_{\text{interleaved}} \sqsubseteq [\![S_Q]\!]_{\text{interleaved}}$.

We have to show that for all P and all $\mathcal{E}$, there exists $\mathcal{E}'$ such that $\Pi(P \oplus C_O \oplus C_Q, \mathcal{E}) \preccurlyeq_{\widehat{obs}} \Pi(P \oplus S_O \oplus S_Q, \mathcal{E}')$. Fix environment $\mathcal{E}$ and client program P. Then by (b), there exists $\mathcal{E}_1$ such that, $\Pi(P \oplus C_O \oplus C_Q, \mathcal{E}) \preccurlyeq_{\widehat{obs}} \Pi(P \oplus C_O \oplus S_Q, \mathcal{E}_1)$. By Lemma A.8, there exists $\mathcal{E}_2$ such that $\Pi(P \oplus C_O \oplus S_Q, \mathcal{E}_1) \preccurlyeq_{\widehat{obs}} \Pi(P \oplus C_O, \mathcal{E}_2)$. By (a), there exists $\mathcal{E}_3$ such that $\Pi(P \oplus C_O, \mathcal{E}_2) \preccurlyeq_{\widehat{obs}} \Pi(P \oplus S_O, \mathcal{E}_3)$. Finally, by applying Lemma A.8 again, there exists $\mathcal{E}'$ such that $\Pi(P \oplus S_O, \mathcal{E}_3) \preccurlyeq_{\widehat{obs}} \Pi(P \oplus S_O \oplus S_Q, \mathcal{E}')$ as needed. □

B. Source Code

Log.java

```java
import java.util.Hashtable;

class Log {
  private LogNode logHead;
  public final static LogNode SENTINEL = new LogNode(null);
  public final Hashtable<Transaction, Boolean> committed;
  public final Hashtable<Transaction, Boolean> ignored;

  public Log() {
    this.logHead = SENTINEL;
    this.committed = new Hashtable<Transaction, Boolean>();
    this.ignored = new Hashtable<Transaction, Boolean>();
  }

  public LogNode getHead() {
    return SENTINEL;
  }
}
```

LogNode.java

```java
import java.util.concurrent.atomic.AtomicReference;

class LogNode {
  public LogNode prev;
  public AtomicReference<LogNode> next;
  public Entry data;

  public LogNode(Entry data) {
    this.prev = null;
    this.next = new AtomicReference<LogNode>(); // starts as null
    this.data = data;
  }

  public String toString() { return this.data.toString(); }
}
```

Entry.java

```java
/** Log operations. */
class Entry {
  /** Transaction id. */
  protected Transaction tx;

  /** Object id. */
  ObjectID objectID;

  /** Sentinel for the beginning of the log. */
  public static final Entry SENTINEL = new Entry(null, null) {
      public String toString() { return "SENTINEL"; }
    };

  protected Entry(Transaction tx, ObjectID objectID) {
    this.tx = tx;
    this.objectID = objectID;
  }

  public boolean isStart() { return false; }
  public boolean isOp() { return false; }
  public boolean isInverse() { return false; }
  public boolean isCommit() { return false; }

  /** Get transaction id. */
  public Transaction getTransaction() { return tx; }

  /** Get object id. */
  public ObjectID getObjectID() { return objectID; }

  public String toString() {
    return super.toString() + "\t" + this.getTransaction() + "\t" +
        this.getObjectID();
  }

  /** Start transaction entry. */
  static public class Start extends Entry {
    public Start(Transaction tx, ObjectID objectID) {
      super(tx, objectID);
    }

    public boolean isStart() { return true; }

    public String toString() { return super.toString() + "\tBEGIN"; }
  }

  /** Operation entry. */
  static public class Op extends Entry {
    /** The operation. */
    Operation op;

    /** The inverse operation. */
    Operation inv;

    public Op(Transaction tx, ObjectID objectID, Operation op,
          Operation inv) {
      super(tx, objectID);
      this.op = op;
      this.inv = inv;
    }

    public boolean isOp() { return true; }

    public String toString() {
      String opstring = tx.getThreadID().getObject(objectID).opName(op
            .opcode);
      String invstring = tx.getThreadID().getObject(objectID).opName(
            inv.opcode);

      return super.toString() + "\tOP(" + opstring + ",␣" + invstring
            + ")";
    }
  }

  /** Inverse operation entry. */
  static public class Inverse extends Entry {
    /** The operation. */
    Operation op;

    public Inverse(Transaction tx, ObjectID objectID, Operation op) {
      super(tx, objectID);
      this.op = op;
    }

    public boolean isInverse() { return true; }

    public String toString() {
      String opstring = tx.getThreadID().getObject(objectID).opName(op
            .opcode);

      return super.toString() + "\tINV(" + opstring + ")";
    }
  }

  /** Commit transaction entry. */
  static public class Commit extends Entry {
    public Commit(Transaction tx, ObjectID objectID) {
      super(tx, objectID);
    }

    public boolean isCommit() { return true; }

    public String toString() { return super.toString() + "\tCOMMIT"; }
  }
}
```

ObjectID.java

```java
class ObjectID {
  public final String typeName;

  public ObjectID(String typeName) {
    this.typeName = typeName;
  }

  public String toString() {
    return this.typeName + ":" + this.hashCode();
  }
}
```

ThreadID.java

```java
class ThreadID {
  protected final java.util.Hashtable<ObjectID, ObjectDefinition> local;

  public ThreadID() {
    this.local = new java.util.Hashtable<ObjectID, ObjectDefinition>();
  }

  public ObjectDefinition addObject(ObjectID obj, ObjectDefinition
        instance) {
    return local.put(obj, instance);
  }

  public ObjectDefinition getObject(ObjectID obj) {
    return local.get(obj);
  }

  public void clear() {
    for (ObjectID obj : local.keySet()) {
      local.get(obj).clear();
    }
  }
}
```

Operation.java

```java
class Operation {
  protected ObjectID objectID;
  protected int opcode;
  protected Object[] args;

  public Operation(ObjectID objectID, int opcode, Object[] args) {
    this.objectID = objectID;
    this.opcode = opcode;
    this.args = args;
  }

  public ObjectID getObjectID() { return this.objectID; }
  public int getOpcode() { return this.opcode; }
  public Object[] getArgs() { return this.args; }
}
```

Result.java

```java
class Result {
  public boolean isSuccess() {
    return false;
  }

  public boolean isAbort() {
    return false;
  }

  public boolean isConflict() {
    return false;
  }

  public static class Success extends Result {
    public boolean isSuccess() {
      return true;
    }
  }

  public static class Abort extends Result {
    protected Entry entry;

    public Abort(Entry entry) {
      this.entry = entry;
    }

    public Entry getEntry() {
      return this.entry;
    }

    public boolean isAbort() {
      return true;
    }
  }

  public static class Conflict extends Result {
    public boolean isConflict() {
      return true;
    }
  }
}
```

Transaction.java

```
class Transaction {
  protected ThreadID threadID;
  protected ObjectID objectID;

  public Transaction(ThreadID threadID, ObjectID objectID) {
    this.threadID = threadID;
    this.objectID = objectID;
  }

  public ThreadID getThreadID() { return threadID; }
  public ObjectID getObjectID() { return objectID; }

  public String toString() {
    return "Transaction(" + this.threadID + ")";
  }
}
```

ObjectDefinition.java

```
import java.util.Deque;
import java.util.ArrayDeque;

abstract class ObjectDefinition {
  protected final ObjectID objectID;
  protected final ThreadID threadID;
  protected Log log;
  private Deque<Operation> inverses;

  public ObjectDefinition(Log log, ObjectID objectID, ThreadID threadID) {
    this.log = log;
    this.objectID = objectID;
    this.threadID = threadID;

    this.threadID.addObject(this.objectID, this);
  }

  public final ObjectID getObjectID() { return this.objectID; }
  public final ThreadID getThreadID() { return this.threadID; }

  public boolean apply(Operation op) {
    return true;
  }

  public void clear() {
  }

  public abstract String opName(int opcode);
  public abstract boolean isConflict(Operation op1, Operation op2);

  public final Result _try_op(Entry entryin) {
    Transaction tx = entryin.getTransaction();
    ThreadID thread = tx.getThreadID();
    LogNode entry = new LogNode(entryin);
    LogNode lastSeen = log.getHead();
    do {
      LogNode abort = null;
      while (null != lastSeen.next.get()) {
        lastSeen = lastSeen.next.get();
        if (log.ignored.containsKey(lastSeen.data.getTransaction())) continue;
        if (log.committed.containsKey(lastSeen.data.getTransaction())) continue;
        if (lastSeen.data.getTransaction() == tx) continue;
        if (isConflict(entryin, lastSeen.data)) {
          return new Result.Conflict();
        } else if (lastSeen.data.isInverse()
                && lastSeen.data.getTransaction().equals(tx)) {
          abort = lastSeen;
        }
      }
      if (null != abort) {
        return new Result.Abort(abort.data);
      }
    } while (! lastSeen.next.compareAndSet(null, entry));
    return new Result.Success();
  }

  protected final boolean isConflict(Entry entryin, Entry other) {
    if (entryin.isOp() && other.isOp()) {
      Entry.Op a = ((Entry.Op) entryin);
      Entry.Op b = ((Entry.Op) other);
      return !(a.getTransaction().equals(b.getTransaction()))
        && a.getObjectID().equals(b.getObjectID())
        && this.getThreadID().getObject(a.getObjectID()).isConflict(a.op, b.op);
    }
    return false;
  }

  protected final boolean apply(Entry entry) {
    if (entry.isOp()) {
      Entry.Op op = ((Entry.Op) entry);
      ObjectID obj = op.op.getObjectID();
      return this.getThreadID().getObject(obj).apply(((Entry.Op) entry).op);
    } else if (entry.isInverse()) {
      Entry.Inverse inv = ((Entry.Inverse) entry);
      ObjectID obj = inv.getObjectID();
      return this.getThreadID().getObject(obj).apply(((Entry.Inverse) entry).op);
    }
    return false;
  }

  public final void sync() {
    sync(null);
  }

  public final void sync(Transaction tx) {
    getThreadID().clear();
    LogNode lastSeen = log.getHead().next.get();
    while (null != lastSeen) {
      if (! log.ignored.containsKey(lastSeen.data.getTransaction())) {
        if (log.committed.containsKey(lastSeen.data.getTransaction())
            || null == tx
            || lastSeen.data.getTransaction() == tx) {
          apply(lastSeen.data);
        }
      }
      lastSeen = lastSeen.next.get();
    }
  }

  public final Transaction begin(ThreadID thread, ObjectID objid) {
    inverses = new ArrayDeque<Operation>();
    Transaction tx = new Transaction(thread, objid);
    sync(tx);
    _try_op(new Entry.Start(tx, this.getObjectID()));
    return tx;
  }

  public final Result try_commit(Transaction tx, ObjectID obj) {
    Entry entry = new Entry.Commit(tx, obj);
    Result result = _try_op(entry);
    if (result.isSuccess()) {
      log.committed.put(tx, true);
    }
    return result;
  }

  public final void add_inverse(Operation inv) {
    inverses.push(inv);
  }

  public final void do_inverses(Transaction tx) {
    log.ignored.put(tx, true);
    while (! inverses.isEmpty()) {
      Operation inv = inverses.pop();
      do_inverse(tx, inv.getObjectID(), inv);
    }
  }

  public final Result do_inverse(Transaction tx, ObjectID obj, Operation
         inv) {
    return _try_op(new Entry.Inverse(tx, obj, inv));
  }

  public final Result try_op(Transaction tx, ObjectID obj, Operation op,
         Operation inv) {
    return _try_op(new Entry.Op(tx, obj, op, inv));
  }
}
```

Hashtable.java

```
import java.util.Set;

class Hashtable<K, V> extends ObjectDefinition {
  public static final String typeName = "HT";

  // opcodes
  public static final int PUT = 1;
  public static final int PUT_INV = 2;
  public static final int REMOVE = 3;
  public static final int REMOVE_INV = 4;

  public String opName(int opcode) {
    switch (opcode) {
    case PUT:
      return "PUT";
    case PUT_INV:
      return "PUT_INV";
    case REMOVE:
      return "REMOVE";
    case REMOVE_INV:
      return "REMOVE_INV";
    default:
      return null;
    }
  }

  @SuppressWarnings("unchecked")
  public boolean apply(Operation op) {
    switch (op.opcode) {
    case PUT:
    case PUT_INV:
    case REMOVE_INV:
      ht.put(((K) (op.args[0])), ((V) (op.args[1])));
      break;
    case REMOVE:
      ht.remove(((K) (op.args[0])));
      break;
    default:
      return false;
    }
    return true;
  }

  public void clear() {
    ht.clear();
  }

  /**
   * Assuming operations are on the same objectID and different,
   * uncommitted transactions
   */
  @SuppressWarnings("unchecked")
  public boolean isConflict(Operation op1, Operation op2) {
    return ((K) (op1.args[0])).equals(((K) (op2.args[0])));
  }

  private final java.util.Hashtable<K, V> ht;

  public Hashtable(Log log, ObjectID objectID, ThreadID thread) {
    super(log, objectID, thread);
    this.ht = new java.util.Hashtable<K, V>();
  }

  public V get(K k) {
    return ht.get(k);
  }

  public Set<K> keySet() {
    return ht.keySet();
  }

  public int size() {
    return ht.size();
  }

  public boolean containsKey(K k) {
    return ht.containsKey(k);
  }

  public V put(K k, V v) {
    return ht.put(k, v);
  }

  public V remove(K k) {
    V v = ht.remove(k);
    return v;
  }
}
```

DirectoryTree.java

```
import java.util.Set;
import java.util.HashSet;

class DirectoryTree extends ObjectDefinition {
  public static final String typeName = "MHT";

  // opcodes
  public static final int ADD = 1;
  public static final int ADD_INV = 2;
  public static final int REMOVE = 3;
  public static final int REMOVE_INV = 4;
  public static final int MOVE = 5;
  public static final int MOVE_INV = 6;

  protected final Hashtable<String, Set<String>> ht;

  public DirectoryTree(Log log, ObjectID objectID, ThreadID thread,
         Hashtable<String, Set<String>> ht) {
    super(log, objectID, thread);
    this.ht = ht;
  }

  public String opName(int opcode) {
    switch (opcode) {
    case ADD:
      return "ADD";
    case ADD_INV:
      return "ADD_INV";
    case REMOVE:
      return "REMOVE";
    case REMOVE_INV:
      return "REMOVE_INV";
    case MOVE:
      return "MOVE";
    case MOVE_INV:
      return "MOVE_INV";
    default:
      return null;
    }
  }

  protected Set<String> collectKeys(Operation op) {
    Set<String> dirs = new HashSet<String>();
    switch (op.opcode) {
    case MOVE:
      String arg1 = ((String) (op.args[1]));
      if (null != arg1) dirs.add(dirname(arg1));
    case ADD:
    case REMOVE:
      String arg0 = ((String) (op.args[0]));
      if (null != arg0) dirs.add(dirname(arg0));
      break;
    default:
      break;
    }
    return dirs;
  }

  @SuppressWarnings("unchecked")
  public boolean isConflict(Operation op1, Operation op2) {
    Set<String> dirs1 = collectKeys(op1);
    Set<String> dirs2 = collectKeys(op2);

    // there is a conflict if set of dirs is disjoint
    Set<String> all = new HashSet<String>();
    all.addAll(dirs1);
    all.addAll(dirs2);
    return all.size() < (dirs1.size() + dirs2.size());
  }

  /**
   * Assumes paths begin with '/'. Returns path ending in '/'.
   */
  public static String dirname(String path) {
    if (path.equals("/")) return "/";

    String[] s = path.split("/");
    s[s.length−1] = "";
    return String.join("/", s);
  }

  /**
   * Assumes paths begin with '/'.
   */
  public static String basename(String path) {
    String[] s = path.split("/");
    return s[s.length−1];
  }

  public Set<String> dirs() {
    sync();
    return ht.keySet();
  }

  public Set<String> list(String d) {
    sync();
    return ht.get(d);
  }

  public int size() {
    sync();
    return ht.size();
  }

  /**
   * Add filename f to directory d.
   */
  public void add(String d, String f) {
    while (true) {
      Transaction tx = begin(this.getThreadID(), this.getObjectID());
      Set<String> old_list = ht.get(d);
      Set<String> new_list;
      if (null == old_list) {
        new_list = new HashSet<String>();
      } else {
        new_list = new HashSet<String>(old_list);
      }
      new_list.add(f);

      Operation op = new Operation(ht.getObjectID(), ht.PUT, new
              Object[] { d, new_list });
      Operation inv = new Operation(ht.getObjectID(), ht.PUT_INV,
              new Object[] { d, old_list });
      add_inverse(inv);
      Result result = try_op(tx, op.getObjectID(), op, inv);
      if (! result.isSuccess()) {
        do_inverses(tx);
        continue;
      }
      ht.put(d, new_list);

      Result cresult = try_commit(tx, this.getObjectID());
      if (cresult.isSuccess()) {
        return;
      } else {
        do_inverses(tx);
        continue;
      }
    }
  }

  public boolean move(String p1, String p2) {
    while (true) {
      Transaction tx = begin(this.getThreadID(), this.getObjectID());
      boolean retval;

      String d1 = dirname(p1);
      String f1 = basename(p1);
      Set<String> old_list1 = ht.get(d1);
      Set<String> new_list1;
      if (null == old_list1) {
        new_list1 = new HashSet<String>();
      } else {
        new_list1 = new HashSet<String>(old_list1);
      }
      if (new_list1.remove(f1)) {
        {
          Operation op1 = new Operation(ht.getObjectID(), ht.PUT,
                  new Object[] { d1, new_list1 });
          Operation inv1 = new Operation(ht.getObjectID(), ht.
                  PUT_INV, new Object[] { d1, old_list1 });
          add_inverse(inv1);
          Result result1 = try_op(tx, op1.getObjectID(), op1, inv1);
          if (! result1.isSuccess()) {
            do_inverses(tx);
            continue;
          }
          ht.put(d1, new_list1);
        }

        {
          String d2 = dirname(p2);
          String f2 = basename(p2);
          Set<String> old_list2 = ht.get(d2);
          Set<String> new_list2;
          if (null == old_list2) {
            new_list2 = new HashSet<String>();
          } else {
            new_list2 = new HashSet<String>(old_list2);
          }
          new_list2.add(f2);

          Operation op2 = new Operation(ht.getObjectID(), ht.PUT,
                  new Object[] { d2, new_list2 });
          Operation inv2 = new Operation(ht.getObjectID(), ht.
                  PUT_INV, new Object[] { d2, old_list2 });
          add_inverse(inv2);
          Result result2 = try_op(tx, op2.getObjectID(), op2, inv2);
          if (! result2.isSuccess()) {
            do_inverses(tx);
            continue;
          }
          ht.put(d2, new_list2);
        }

        retval = true;
      } else {
        retval = false;
      }

      Result cresult = try_commit(tx, this.getObjectID());
      if (cresult.isSuccess()) {
        return retval;
      } else {
        do_inverses(tx);
        continue;
      }
    }
  }
}
```

MoveableHashtable.java

```
class MoveableHashtable<K, V> extends ObjectDefinition {
  public static final String typeName = "MHT";

  // opcodes
  public static final int PUT = 1;
  public static final int PUT_INV = 2;
  public static final int REMOVE = 3;
```

```java
public static final int REMOVE_INV = 4;
public static final int MOVE = 5;
public static final int MOVE_INV = 6;

protected final Hashtable<K, V> ht;

public MoveableHashtable(Log log, ObjectID objectID, ThreadID thread
        , Hashtable<K, V> ht) {
  super(log, objectID, thread);
  this.ht = ht;
}

public String opName(int opcode) {
  switch (opcode) {
  case PUT:
    return "PUT";
  case PUT_INV:
    return "PUT_INV";
  case REMOVE:
    return "REMOVE";
  case REMOVE_INV:
    return "REMOVE_INV";
  case MOVE:
    return "MOVE";
  case MOVE_INV:
    return "MOVE_INV";
  default:
    return null;
  }
}

@SuppressWarnings("unchecked")
public boolean isConflict(Operation op1, Operation op2) {
  return ((K) (op1.args[0])).equals(((K) (op2.args[0])));
}

public V get(K k) {
  sync();
  return ht.get(k);
}

public int size() {
  sync();
  return ht.size();
}

public V put(K k, V v) {
  while (true) {
    Transaction tx = begin(this.getThreadID(), this.getObjectID());
    V v_old = ht.get(k);

    Operation op = new Operation(ht.getObjectID(), ht.PUT, new
          Object[] { k, v });
    Operation inv = new Operation(ht.getObjectID(), ht.PUT_INV,
          new Object[] { k, v_old });
    add_inverse(inv);
    Result result = try_op(tx, op.getObjectID(), op, inv);
    if (! result.isSuccess()) {
      do_inverses(tx);
      continue;
    }
    ht.put(k, v);

    Result cresult = try_commit(tx, this.getObjectID());
    if (cresult.isSuccess()) {
      return v;
    } else {
      do_inverses(tx);
    }
  }
}

public boolean move(K k1, K k2) {
  while (true) {
    Transaction tx = begin(this.getThreadID(), this.getObjectID());
    V v = ht.get(k1);
    V v_old = ht.get(k2);

    if (null != v) {
      Operation op1 = new Operation(ht.getObjectID(), ht.PUT, new
            Object[] { k2, v });
      Operation inv1 = new Operation(ht.getObjectID(), ht.PUT_INV,
            new Object[] { k2, v_old });
      add_inverse(inv1);
      Result result1 = try_op(tx, op1.getObjectID(), op1, inv1);
      if (! result1.isSuccess()) {
        do_inverses(tx);
      }
      ht.put(k2, v);


      Operation op2 = new Operation(ht.getObjectID(), ht.REMOVE,
            new Object[] { k1 });
      Operation inv2 = new Operation(ht.getObjectID(), ht.
            REMOVE_INV, new Object[] { k1, v_old });
      add_inverse(inv2);
      Result result2 = try_op(tx, op2.getObjectID(), op2, inv2);
      if (! result2.isSuccess()) {
        do_inverses(tx);
        continue;
      }
      ht.remove(k1);
    }

    Result cresult = try_commit(tx, this.getObjectID());
    if (cresult.isSuccess()) {
      return null != v;
    } else {
      do_inverses(tx);
      continue;
    }
  }
}
}
```

FileSystem.java

```java
import java.util.Set;
import java.util.HashSet;

class FileSystem extends ObjectDefinition {
  public static final String typeName = "FS";

  // opcodes
  public static final int ADD = 1;
  public static final int ADD_INV = 2;
  public static final int DEL = 3;
  public static final int DEL_INV = 4;
  public static final int MOVE = 5;
  public static final int MOVE_INV = 6;

  protected final MoveableHashtable<String, Integer> mht;
  protected final DirectoryTree dt;

  public FileSystem(Log log, ObjectID objectID, ThreadID thread,
        MoveableHashtable<String, Integer> mht, DirectoryTree dt) {
    super(log, objectID, thread);
    this.mht = mht;
    this.dt = dt;
  }

  public String opName(int opcode) {
    switch (opcode) {
    case ADD:
      return "ADD";
    case ADD_INV:
      return "ADD_INV";
    case DEL:
      return "DEL";
    case DEL_INV:
      return "DEL_INV";
    case MOVE:
      return "MOVE";
    case MOVE_INV:
      return "MOVE_INV";
    default:
      return null;
    }
  }

  protected Set<String> collectKeys(Operation op) {
    Set<String> dirs = new HashSet<String>();
    switch (op.opcode) {
    case ADD:
    case DEL:
      dirs.add(DirectoryTree.dirname(((String) (op.args[0]))));
    case MOVE:
      dirs.add(DirectoryTree.dirname(((String) (op.args[1]))));
      break;
    default:
      break;
    }
    return dirs;
  }

  @SuppressWarnings("unchecked")
  public boolean isConflict(Operation op1, Operation op2) {
    Set<String> dirs1 = collectKeys(op1);
    Set<String> dirs2 = collectKeys(op2);

    // there is a conflict if set of dirs is disjoint
    Set<String> all = new HashSet<String>();
    all.addAll(dirs1);
    all.addAll(dirs2);
    return all.size() < (dirs1.size() + dirs2.size());
  }

  public Set<String> dirs() {
    sync();
    return dt.dirs();
  }

  public Set<String> list(String d) {
    sync();
    return dt.list(d);
  }

  public Integer get(String path) {
    sync();
    return mht.get(path);
  }

  public int size() {
    sync();
    return mht.size();
  }

  /**
   * Overwrites an existing file.
   */
  public Integer addFile(String path, Integer v) {
    while (true) {
      Transaction tx = begin(this.getThreadID(), this.getObjectID());
      Integer v_old = mht.get(path);

      Operation op1 = new Operation(mht.getObjectID(), mht.PUT,
            new Object[] { path, v });
      Operation inv1 = new Operation(mht.getObjectID(), mht.
            PUT_INV, new Object[] { path, v_old });
      add_inverse(inv1);
      Result result1 = try_op(tx, op1.getObjectID(), op1, inv1);
      if (! result1.isSuccess()) {
        if (result1.isConflict()) {
          do_inverses(tx);
        }
        continue;
      }
      mht.put(path, v);

      String d = DirectoryTree.dirname(path);
      String f = DirectoryTree.basename(path);
      Operation op2 = new Operation(dt.getObjectID(), dt.ADD, new
            Object[] { d, f });
      Operation inv2 = new Operation(dt.getObjectID(), dt.ADD_INV,
            new Object[] { d, f });
      add_inverse(inv2);
      Result result2 = try_op(tx, op2.getObjectID(), op2, inv2);
      if (! result2.isSuccess()) {
        do_inverses(tx);
        continue;
      }
      dt.add(d, f);

      Result cresult = try_commit(tx, this.getObjectID());
      if (cresult.isSuccess()) {
        return v;
      } else {
        do_inverses(tx);
        continue;
      }
    }
  }

  public boolean moveFile(String path1, String path2) {
    while (true) {
      Transaction tx = begin(this.getThreadID(), this.getObjectID());

      {
        Operation op1 = new Operation(mht.getObjectID(), mht.MOVE,
              new Object[] { path1, path2 });
        Operation inv1 = new Operation(mht.getObjectID(), mht.
              MOVE_INV, new Object[] { path2, path1 });
        Result result1 = try_op(tx, op1.getObjectID(), op1, inv1);
        if (! result1.isSuccess()) {
          do_inverses(tx);
          continue;
        }
        mht.move(path1, path2);
      }

      {
        Operation op2 = new Operation(dt.getObjectID(), dt.MOVE,
              new Object[] { path1, path2 });
        Operation inv2 = new Operation(dt.getObjectID(), dt.
              MOVE_INV, new Object[] { path2, path1 });
        Result result2 = try_op(tx, op2.getObjectID(), op2, inv2);
        if (! result2.isSuccess()) {
          do_inverses(tx);
          continue;
        }
      }
      boolean retval = dt.move(path1, path2);

      Result cresult = try_commit(tx, this.getObjectID());
      if (cresult.isSuccess()) {
        return retval;
      } else {
        do_inverses(tx);
        continue;
      }
    }
  }
}
```

Main.java

```
import java.util.ArrayList;
import java.util.Set;
import java.util.HashSet;

class Main {
  public static void example() throws InterruptedException {
    Log log = new Log();

    // initialize the RAOs' unique IDs
    ObjectID htID = new ObjectID(Hashtable.typeName);
    ObjectID mhtID = new ObjectID(MoveableHashtable.typeName);
    ObjectID htdtID = new ObjectID(Hashtable.typeName);
    ObjectID dtID = new ObjectID(DirectoryTree.typeName);
    ObjectID fsID = new ObjectID(FileSystem.typeName);

    // for each thread
    ArrayList<ThreadID> threads = new ArrayList<ThreadID>();
    ArrayList<MoveableHashtable<String, Integer>> mhts = new
          ArrayList<MoveableHashtable<String, Integer>>();
    ArrayList<FileSystem> fss = new ArrayList<FileSystem>();
    for (int i = 0; i < 3; i++) {
      ThreadID thread = new ThreadID();
      Hashtable<String, Integer> ht = new Hashtable<String, Integer
            >(log, htID, thread);
      MoveableHashtable<String, Integer> mht = new
            MoveableHashtable<String, Integer>(log, mhtID, thread, ht)
            ;
            Hashtable<String, Set<String>> htdt = new Hashtable<String, Set<String>>(log, htdtID, thread);
            DirectoryTree dt = new DirectoryTree(log, dtID, thread, htdt);
            FileSystem fs = new FileSystem(log, fsID, thread, mht, dt);
            threads.add(thread);
            mhts.add(mht);
            fss.add(fs);
        }

        Thread creator = new Thread() {
                public void run() {
                    for (int i = 1; i <= 100; i++) {
                        String filename = "/file" + i;
                        fss.get(0).addFile(filename, i);
                        System.out.println("created " + filename);
                    }
                }
            };

        Thread mover = new Thread() {
                public void run() {
                    int num_moved = 0;
                    while (num_moved < 50) {
                        for (int i = 2; i <= 100; i += 2) {
                            String filename = "/file" + i;
                            String new_filename = "/evens/file" + i;
                            boolean moved = fss.get(1).moveFile(filename, new_filename);
                            if (moved) {
                                System.out.println("moved " + filename + " to " + new_filename);
                                num_moved++;
                            }
                        }
                        System.out.println("moved " + num_moved);
                    }
                }
            };

        Thread printer = new Thread() {
                public void run() {
                    int size = 0;
                    while (size < 100) {
                        size = fss.get(2).size();
                        System.out.println("size " + size);
                    }
                }
            };

        creator.start();
        mover.start();
        printer.start();
        creator.join();
        mover.join();
        printer.join();

        // // print the full log for debugging
        // for (LogNode cur = log.getHead(); null != cur; cur = cur.next.get()) {
        // System.out.println(cur.data);
        // }

        Set<String> dirs = new HashSet<String>(fss.get(0).dirs());
        for (String dir : dirs) {
            System.out.println("DIRNAME: " + dir);
            Set<String> files = fss.get(0).list(dir);
            int num = 0;
            System.out.print("CONTENTS: ");
            for (String file : files) {
                System.out.print(file + ", ");
                num++;
            }
            System.out.println("\nCOUNT:" + num);
            System.out.println();
        }
    }

    public static void main(String args[]) throws InterruptedException {
        example();
    }
}
```