A Theory of Vertically Composable Transactional Objects

Timos Antonopoulos, Paul Gazzillo, Eric Koskinen, and Zhong Shao

Yale University

Abstract. We introduce a methodology and formal model that captures the essence of vertically composable transactional objects. Vertical composition adds complexity to transactional systems. As such, we aim to unearth a clean semantic model that strikes a balance between anticipating future implementation methodologies yet, nonetheless, offering a formal treatment of effective existing implementations. To this end we adopt a layered approach and show that first-class treatment of computation reversibility leads to a natural form of vertical composition: a given upper layer in a hierarchy can use inverses to roll back its operations and a contention manager can ensure progress by, at any point, applying inverses on behalf of an executing transaction. The model's expressiveness is evident, for example, from the fact that we do not require that one object layer use the same implementation strategy (e.g. pessimism versus optimism) as another. Our main technical results are the first proofs of contextual refinement (stronger than serializability) and vertical composition for transactional objects. Our underlying semantics gives rise to a novel transactional variant of Herlihy's Universal Construction.

Our model is a generalization of many known TM implementations, including memory transactions, transactional boosting, some nested transactions, etc. Yet the model also anticipates new strategies and leads to a framework for constructing highly-concurrent systems in a modular way. To this end, we describe how it could be used to implement a highly-concurrent transactional file system out of linearizable base objects.

1 Introduction

The landmark linearizability paper of Herlihy and Wing [20] 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 [12,15,27,44,43,2,36] and implementations (e.g. `java.util.Concurrent`) that exploit this atomicity abstraction and allow one to build complex systems from sensible building blocks.

Linearizability doesn't, however, provide a clear methodology for how one can build larger objects that are themselves linearizable *on top of* these objects. Modern systems such as web servers, databases, and services-oriented architectures need to leverage concurrency in order to efficiently respond to client requests. Linearizable data-structures provide a starting point but they only really

offer base primitives; many layers of code must be built on top of them. Without an alternative at hand, programmers currently resort to traditional synchronization methods such as locking in these layers, leading to code that is difficult to understand and riddled with notorious concurrency bugs such as deadlock, livelock, reentrancy, etc. A similar concern holds in the distributed setting. Concurrency is needed for applications such as SDN controllers and replicated data structures, but layers are currently built on top of primitives in an ad-hoc fashion. Indeed, large-scale concurrent software is difficult to write and principled approaches to vertical composition over these layers of abstraction would be immensely helpful.

Let's say, for instance, that one has a linearizable concurrent hashtable CHT that supports `put`(k, v), `get`(k), `size`() and that one wants to implement `move`(k_1, k_2) which should atomically move a value from one key to another. On the one hand, extending CHT to support the additional method while maintaining linearizability would be difficult because it involves careful reasoning about how `move` inter-plays with the existing methods. Yet, on the other hand, this should be straight-forward because, logically speaking, it can be constructed from a combination of CHT operations.

Transactional boosting [19] and the theory of coarse-grained transactions [25] offered the first step in the direction of composing transactional objects. They showed that transactions need not consist of read/write memory operations but can, instead, consist of base operations that are methods of a highly-concurrent linearizable base object (for details, see [19]). The efficiency of boosting stems from the fact that these "constituent" operations of a transaction are already highly efficient. With boosting one can build transactions that could start to look a little like the methods of a higher-level atomic object called, say, a MoveableHashtable MHT. In this sense, boosting can be thought of as one layer of vertical composition, or, "1VC."

Now what if we want to generalize the vertical composition to higher levels: nVC? What if we, for example, want to use the (highly-concurrent) MoveableHashtable as the basis for implementing a highly-concurrent Filesystem object? We might use the MoveableHashtable keys to represent file-paths and values as the file contents as well as, say, a highly-concurrent linearizable tree to track the file-path hierarchy. As we move vertically we would like to maintain, as an invariant that the operations on every level be atomic, just as was true for the base operations all the way up to the top, where our filesystem operations `unlink`, `rename`, etc. operate with the appearance of atomicity. To date, although some implementations permit similar vertical compositions, there is no known theoretical framework or vertical composition theorems to achieve this.

In another recent trend, techniques have emerged for synthesizing data-structure commutativity conditions via abstraction refinement [5] or learning [13]. Others use commutativity conditions to synthesize transactional conflict management [14,7,10]. We leverage these orthogonal techniques in our work.

This paper. We present a theory that captures the essence of vertically composable transactional objects. We begin by establishing a syntax and methodology

for building *reversible atomic objects* (RAOs). In each layer of the vertical composition, an upper-level object implements an operation $O.m()$ with a transaction that consists of *constituent* operations on objects in the levels below:

$$O.m(\bar{x}) \triangleq \textbf{atomic}\{O_1.a(\bar{x}_1); O_2.b(\bar{x}_2); \dots\}$$

The object hierarchy forms a directed acyclic graph: method calls can only invoke operations on lower-level objects and the lowest level are linearizable base objects in a simple wrapper. Our methodology and model explicitly requires that, for every object operation, transaction begin/commit is directly aligned with the object's method invocation/response (respectively). We also require that each RAO method is associated with its commutativity specification (obtained using other techniques [14,5,13,10]).

Next, we take a first-class treatment of object method inverses and require that, by the time a method commits, it must have *constructed its own inverse* operation. These inverses empower both (*i*) the transaction layer above to undo the effects of its constituent operations and (*ii*) our novel contention manager to ensure progress by aborting operations on behalf of the thread. This treatment of inverses bares some similarity to so-called compensating actions in open nested transactions (see discussion in Section 9) and generalizes the inverses of boosting [19].

Interestingly, once inverses are established on the base linearizable objects, all higher-level inverses can be constructed automatically. At each level, the inverse can be constructed by assembling—in reverse order—the inverses of each constituent operation. Nonetheless, a programmer may instead choose to provide one manually, especially in cases where the inverse can be achieved with fewer base operations.

The main benefit of working with reversible atomic objects is that the model provide formal guarantees including contextual refinement[1] and vertical composition, leading to systems that are correct by construction. In this way we believe our approach lifts the Herlihy/Wing notion of atomic objects to a multi-layer strategy, with implementation flexibility at each level. Finally, semantic model based on a global logical log gives rise to a novel transactional variant of the Universal Construction [18].

Contributions. To the best of our knowledge, our work is the first formal treatment of vertical composition of transactional objects. 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. (Section 5)

[1] Contextual refinement is more powerful than serializability. It is becoming common to use contextual refinement for showing the correctness of transactional memory. Connections to serializability, opacity, TMS, etc. can be found in the literature [3,4].

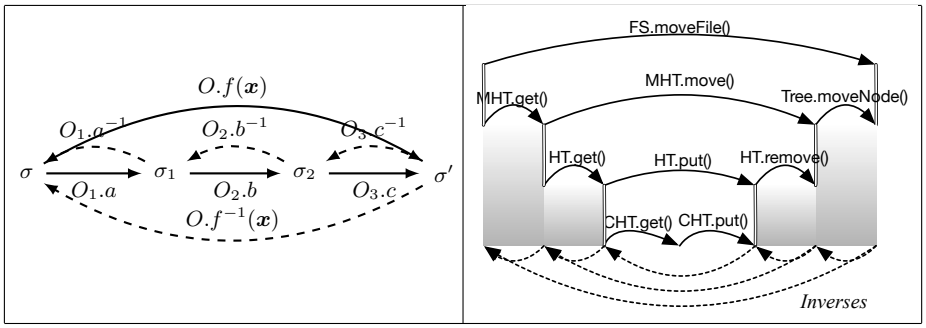

Fig. 1. (*Left*) Formal structure of reversible atomic objects. (*Right*) An example highly-concurrent filesystem reversible atomic object, constructed vertically from other reversible atomic objects. (Some detail omitted and replaced with gray boxes.)

- A proof that programs composed with implementations are a termination-sensitive contextual refinement of the same program composed instead 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 context that may, at any point, invert a transaction's uncommitted operations. (Section 7)
- A novel transactional variant of the universal construction, arising from our compositional treatment in terms of a single shared logical log. (Section 8.2)

To our knowledge there are no known models that allow this kind of nVC vertical composition of transactional objects. As we discuss in Section 8.1, the recent Push/Pull model can be viewed as 1VC. This model is expressive enough to cover a wide range of existing transactional implementations. In the next section we describe an instance of nVC, implementing a highly-concurrent transactional filesystem out of linearizable base objects.

2 An Illustration of reversible atomic objects

We now introduce our formal model and methodology by giving an example of how it might appear in the syntax of a suitable future programming language. We will take the running example of constructing a highly-concurrent linearizable filesystem. Formally, a *reversible atomic object* O implements an abstract atomic operation f, built from base operations, that it assumes already behave atomically. Such an object constructs an abstract state transformer $O.f(\boldsymbol{x}) : \Sigma \to \Sigma$ 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 the diagram on the left in Figure 1. The dotted lines are inverses, discussed later.

Filesystem operations touch multiple data structures that must be kept in sync with each other for integrity, a challenge for concurrent programs. Normally such filesystems are incredibly complex.

FileSystem, **MoveableHashtable**, and **Hashtable** Reversible Atomic Objects
Note: Code in gray can be automatically generated by a compiler.

```
1  class FileSystem[P, V] : RAO {
2    MoveableHashTable[P, V] mht
3    DirectoryTree[P] tree
4    ...
5    optim bool moveFile(p1, p2) = atomic{ conflict(wr:p1,wr:p2,wr:lca(p1,p2))
6      v = mht.get(p1) ↦ inv0
7      if (v is empty) { cmt_return (inv0, false) }
8      else { mht.move(p1, p2) ↦ inv1
9             tree.moveNode(p1, p2) ↦ inv2
10            cmt_return (atomic{inv2; inv1; inv0}, true)  }
11   }
12 }
```

```
1  class MoveableHashtable[K, V] : RAO {
2    Hastable[K, V] ht
3    ...
4    pess V get(k) = atomic{ conflict(rd:k)
5      v := ht.get(k) ↦ skip
6      cmt_return (skip, v)
7    }
8    pess bool move(k1, k2) = atomic{  conflict(wr:k1, wr:k2, wr:size)
9      v := ht.get(k1) ↦ inv1
10     v_old := ht.put(k2, v) ↦ inv2
11     ht.remove(k1) ↦ inv3
12     cmt_return (atomic{inv3; inv2; inv1}, ⊥)
13   }
14 }
```

```
1  class Hashtable[K, V] : RAO {
2    ConcurrentHashtable[K, V] cht
3
4    V get(k) { conflict(rd k)
5      x = cht.get(k)
6      cmt_return (skip, x)
7    }
8    V put(k, v) { conflict(wr k, wr:sz)
9      v_old = cht.get(k)
10     cht.put(k, v)
11     cmt_return (cht.put(k, v_old), v_old)
12   }
13   V remove(k) { conflict(wr k, wr:sz)
14     v_old := cht.get(k)
15     cht.remove(k)
16     cmt_return (cht.put(k, v_old), v_old)
17   }
18 }
```

Fig. 2. Illustration of a highly-concurrent file system implemented from reversible atomic objects. The code in gray could be generated automatically by a compiler: conflict specifications via [14,7,10,13,5] and inverses via dynamic instrumentation.

Figure 2 provides an illustration of the modularity of reversible atomic objects: a pseudo-code implementation of a highly-concurrent filesystem in under 50 lines of code. At the top level, we have a FileSystem reversible atomic object (RAO) that will provide standard POSIX-like file operations such as moveFile (a.k.a. rename), mkdir, rm, etc. FileSystem needs to be implemented efficiently so we implement it using a reversible atomic tree (for directory lookups) and reversible atomic hashtable (to store the payload file data for each full path): internal data-structures mht and tree, respectively. They will, in turn, be implemented on top of other RAOs. Ultimately, the lowest layer is formed by wrapping a linearizable concurrent base object, such as the HashTable RAO which wraps a linearizable hashtable. A diagram of how this implementation (call stack) correlates to our theoretical model is give on the right in Figure 1.

Consider now the implementation of moveFile in Figure 2. The code in light gray could be inserted by a compiler (discussed below). While the non-gray user code of this simple example looks straight-forward and the features of this method look familiar, there are many behind-the-scenes details to be appreciated. Our methodology and formal model is based on the following structural rules, which we will discuss in turn:

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

Aligning transactions with the method boundary. Reversible atomic objects impose a structure on the way in which transactions are used: they must be correlated with object methods. The body of moveFile *is* an atomic section, denoted **atomic{**...**}**. This object view is one of several ways in which we diverge from so-called nested transactions (see Section 9). 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 boolean) back to the caller. We will discuss **cmt_return** more later.

Conflict specification. All reversible atomic object methods must provide a conflict specification in the form of commutativity: the conditions under which an invocation of method $O.f(\boldsymbol{x})$ commutes with all other active operations on that object. In our formal model, this commutativity relation is defined in terms of observational equivalence (Section 3.6). These specifications do not have to be written by hand. A number of recent techniques have appeared in the literature for generating conflict specifications in the form of commutativity (via abstraction refinement [5] or learning [13]) and using those commutativity conditions to synthesize transactional conflict management algorithms such as abstract locking [14,7,10].

For the reader's benefit, we illustrate these conflict specifications using symbolic, dynamic *access points* [10]. This approach allows us to summarize conflict

by associating object methods with elements in a symbolic domain of so-called access points. Take, for example, moveFile in Figure 2. The synthesized [5,10] conflict specification for moveFile (Line 5) is $conflict(wr{:}p_1, wr{:}p_2, wr{:}lca(p_1, p_2))$. Intuitively, this specification means that moveFile may modify the contents of file p_1, may modify the contents of file p_2 or, taking an approximation, modify file paths in the directory subtree rooted at the least common ancestor of the two paths. There is also a conflict relation $\mathcal{C}$ and operations $O.f(\bar{x})$ and $O.g(\bar{y})$ are said to conflict whenever an access point of $O.f(\bar{x})$ is in relation $\mathcal{C}$ with an access point of $O.g(\bar{y})$. In addition to the wr prefix, there is also a rd prefix, as seen in MoveableHashTable. This permits distinction between read/read interactions and read/write interactions.

Reversible atomic objects leverage the declarative nature of access point specifications in order to have flexibility in how the runtime system manages conflict. In Figure 2 we illustrate how a user might guide this choice by annotating methods with keywords **pess** and **optim**. Consider the moveFile operation. In such a pessimistic case, the transactional system pauses the transaction until there are no conflicting concurrent operations with access points that are in conflict with access points $wr : p_1, wr : p_2, wr : lca(p_1, p_2)$. Concretely speaking, one way this can be accomplished is by translating access point specifications into locking schemes [19,14]. These schemes ensure that, if two transactions conflict, then they will both need to acquire at least one lock in common. When a caller invokes the moveFile method, it will acquire the appropriate synthesized locks so as to ensure it has exclusive access to the the file paths in question. In the optimistic case, an implementation might (logically) furnish each thread/transaction with its own local copy of the object that they can immediately mutate. Later, at commit time, the transactional system can check whether there is conflict between this operation's access points and those of recently committed transactions. Conflict at the FileSystem layer *has nothing to do* (directly) with conflict at the level of the constituent objects: MoveableHashtable and DirectoryTree. We will later discuss how layers relate and how conflict managers resolve conflict.

Invoking constituent operations. The implementation of an RAO method is free to perform methods on constituent reversible atomic objects: $O_1.a, O_2.b$, etc. The body of moveFile manipulates methods of 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.

A key element of reversible atomic objects is that they maintain the invariant that every operation has an inverse. Let's assume this invariant and note two things about an operation's invocation. First, at the location immediately preceding each operation, the compiler will insert code that captures the continuation [22]. This is a form of checkpoint that will be used in case the operation is inverted. Second, by way of the invariant each constituent operation, since it is also a reversible object, will return an inverse operation. These inverses

inv_0, inv_1, etc. are saved for two reasons. First,the inverse operation may be invoked by the environment contention manager (discussed below). Second, it may be used in the construction of the overall inverse operation for moveFile, as seen on Line 10 and discussed next.

Assembling the inverse. Before completing the operation $O.f(\boldsymbol{x})$ (via **cmt_return**), a reversible atomic object must prepare the inverse operation $O.f^{-1}(\boldsymbol{y})$ to maintain the invariant. Inverses can be generated automatically by assembling the inverses of the constituents, as we do for the inverse of moveFile on Line 10. In other cases the programmer might provide a smarter inverse. The inverses themselves, while atomic, do not themselves have inverses.

Inverse operations may be used at the next level up. In a transaction at that higher level, $O.f(\boldsymbol{x})$ 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 $O.f^{-1}(\boldsymbol{y})$ is still a valid inverse when there may be other concurrent operations? This inverse $O.f^{-1}(\boldsymbol{y})$ is a short-lived inverse and can only be used during the lifespan of the above transaction. Consequently, since the above transaction ensures that $O.f(\boldsymbol{x})$ is free of conflict with any concurrent operation, it is easy to show that one can commute $O.f(\boldsymbol{x})$ forward in time, until it is adjacent to $O.f^{-1}(\boldsymbol{y})$ and that these two then annihilate each other.

Committing and returning. 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 will already have ensured that the current transaction has the right-of-way (i.e. there are no concurrent operations that have access points in conflict with access points $wr : p_1, wr : p_2,$, or $wr : lca(p_1, p_2)$) so this commit event can happen immediately. An optimistic implementation, on the other hand, would need to perform commit-time conflict detection. 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.

The overall program consists of concurrent threads $P = (C_1 \mid\mid \cdots \mid\mid C_n)$ that are the clients of top-level reversible atomic objects. As with the lower layers, the clients call reversible atomic object methods, 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 3 is an example of threads all operating on the same FileSystem object. The Creator thread generates files, the Mover thread moves some of the files, and Printer reports the number of files. These threads can be scheduled in any order. Each takes advantage of the atomicity of the FileSystem object.

```
class Creator : Thread {
  run(FileSystem[K,V] fs) {
    for i := 1..100
      fs.addFile("file" + i, i * i); } }

class Printer : Thread {
  run(FileSystem[K,V] fs) {
    num_files = 0;
    while (num_files < 100)
      print fs.numFiles(); } }
```

```
class Mover : Thread {
  run(FileSystem[K,V] fs) {
    while (num_moved < 50) {
      for i := 1..100 step 2 {
        moved = fs.moveFile("file"+i, "b"+i);
        if (moved) num_moved++;
      }
    }
  }
}
```

Fig. 3. 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.

2.1 Across this layer: concurrency, 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 FileSystem. Transactional memory systems address contention with a so-called contention manager that implements some policy, deciding whom should be aborted [41,40,42]. If the contention manager is able to also know when deadlocks occur (e.g. [23]), then it can implement a policy that ensures overall progress.

In our formal model, the contention manager is an *environment context* that can not only control the scheduling of transactions [30], but also detect deadlock transactions and *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). Consider this sequence between transactions τ and τ':

τ begins; τ' begins; τ' completes mht.move(7,8); τ completes mht.move(5,6)

Now imagine that 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''.

In our model, logically speaking, threads are aware that the environment may take charge and invert operations on behalf of them. When a thread finds that the environment has inverted one or more of its operations (always in reverse order) back to some earlier program location, the thread must resume execution at that location via a previously captured continuation.

Contextual Refinement. The first formal result of this paper (Section 6) is that reversible atomic objects provide a contextual refinement guarantee. (Note: Contextual refinement is stronger than serializability [3,4].) If objects in the system follow the above criteria, abiding by the conflict specifications and establishing inverses then, for every (multi-threaded) program P, execution of P composed with an object's implementation C_O is a contextual refinement of P composed with the object's corresponding atomic specification S_O, denoted:

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

Our formalization (Section 5) 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 (logical) event log*. This does not mean that we would advocate that implementations use a physical log. Rather, in recent years it has been shown that a log can serve as a reasoning technique [24,16] technique by reducing the interaction between threads to a core essence of events. With the logical log treatment, we provide novel representations of transactional concepts including: schedule, transactions indicating they cannot make progress and contention management.

2.2 Below this layer: vertical composition.

FileSystem is built from two constituent reversible atomic objects, one of which is the MoveableHashtable, defined in the middle of Figure 2. 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 8 has three constituent operations: ht.get(k_1), ht.put(k_2, v), 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 (**pess**), again demonstrating the flexibility of permitting different transactional implementations at different vertical layers. Consider a pair of upper/lower levels such as MoveableHashtable and Hashtable. The conflict specification of the lower level (Hashtable) is used by the runtime system to govern the execution of the constituent operations of atomic sections in the upper level (MovableHashTable.get). This specification may be used differently depending on whether the upper level is pessimistic or optimistic.

Base objects. The RAO model is rooted on the linearizability (and performance!) of the *base* concurrent objects. This is accomplished via a wrapper such as Hashtable in Figure 2 that wraps a linearizable ConcurrentHashtable (cht). 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.

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: $\mathbf{k} : \mathbf{K}$, $\mathbf{v} : \mathbf{V}$. The conflict specification of put, for example, pertains to key $\mathbf{k}$ and the fact that the overall size of the cht (denoted as sz to distinguish from $size$) 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.

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 access points $wr\,\mathbf{k}$,$wr:sz$. This specification covers both cht.get($\mathbf{k}$) and cht.put($\mathbf{k}$,$\mathbf{v}$).

In Section 6.1 we show formally that objects can be vertically composed. Theorem 2 says that, for any two objects O and Q with implementations/specifications C_O/S_O and C_Q/S_Q, that

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

This relationship means that vertically composing object implementations preserves the contextual refinement guarantees against object specifications.

2.3 Transactional Universal Construction

Our compositional semantics that involves threads communicating via a logical log leads to a novel transactional variant of the so-called universal construction [18]. Our construction allows us to create a transactional concurrent version of any kind of sequential data-structure implementation D. Briefly, as in the original construction [18], each thread i maintains its own replica D_i of the data-structure. They then coordinate via a shared log that supports an atomic append/enqueue operation. This log is the authoritative history of the state of D, as a list of the operations that have been performed. Threads scan the log and replay the operations on their local D_i replica. When a thread wishes to perform an operation on D, it competes to append the name of the operation to the log.

In order to support transactions, a few extensions are needed. First, we first augment the type of log entries, so that events other than operations can also be recorded. Log entries have a transaction identifier, and threads also append begin and commit messages to the log. Inverses may be appended by a contention manager, signaling an abort.

The key abstraction is that threads, when attempting to append, perform an algorithm we call try_cmd. This algorithm attempts to append a new log entry. The log responds with either Success indicating that the entry has been appended, Conflict indicating that there is another uncommitted operation that conflicts, or Inverted indicating that some or all of the transaction's operations have been inverted. The try_cmd abstraction permits a range of transactional

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

Fig. 4. Events of the system.

policies from pessimistic to optimistic. The more eagerly a thread appends operations to the log, the more pessimistic it is. Alternative, a thread may optimistically perform all operations on a thread-local copy, and attempt to append all the operations at once just before committing.

3 Preliminaries

In this section we establish some formal preliminaries. Our formalism uses a *logical* notion of a global log of events (i.e. a history). In this section we describe how object transactions interact with the log, as well as inverses and conflicts.

3.1 States, Operations, Event Logs

We will work with a *state space* Σ. Operations are denoted by a, b, etc. and they are of type $\Sigma \to \Sigma$. We let **Ops** be a set of base operations.

Global Log and Threads. We will work with a globally shared system log ℓ of type Ev^*, which records a sequence of threads' events from a domain of events Ev. We define events in more detail in Section 3.3 and a list of them can be found in Figure 4. 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. 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 $\ell \cdot \langle(\tau, a)\rangle$, where $\langle(\tau, a)\rangle$ denotes the sequence that contains the single event (τ, a).

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

Observations. An observation $\mathit{obs}(\ell \cdot (\tau, a))$ 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 $\ell[0] \cdots \ell[i]$.

3.2 Objects

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

Given an object O and one of its methods $O.f(\boldsymbol{x})$, we define $\mathsf{spec}_{O.f(\boldsymbol{x})} : \mathbf{D}^* \to \mathbf{Ops}^*$. 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 $\mathsf{spec}_{O.f(\boldsymbol{x})}(x)$ would simply return the corresponding sequence according to the value of x. Our model does not preclude object methods that involve looping or even recursion, per se. We simply require that the method consist of a finite sequence of base operations. (After all, linearizability is termination sensitive.) Furthermore, we note that a method $O.f(\boldsymbol{x})$ cannot call another method $O.g(\boldsymbol{x})$ of the same object O. To simulate such functionality, one can use a different object on a higher layer that makes calls to both $O.f(\boldsymbol{x})$ and $O.g(\boldsymbol{x})$ as needed. On the other hand, helper methods could assist with defining or implementing (terminating) recursive calls.

Next, let $\mathsf{specLog}_{O.f(x)} : \mathcal{L} \times \mathbf{D}^* \to \mathcal{L}$ be a mapping that, given a log ℓ and a sequence of arguments $\boldsymbol{x}$ for the method $O.f(\boldsymbol{x})$, traverses the log ℓ and consults the function $\mathsf{spec}_{O.f(\boldsymbol{x})}$. It produces the correct sequence of events in the log corresponding to: invoking the method, executing the correct sequence of base operations and then appending the event of committing and returning.

3.3 Events

In addition to the base event (τ, a), there are other events that can be emitted by a thread transition $\xrightarrow{r\ \ell}$. The events are given in Figure 4. As mentioned above, event (τ, a) is an instance of thread τ performing operation a. The first event $(\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x}))$ models thread τ invoking an operation $O.f(\boldsymbol{x})$. If $O.f(\boldsymbol{x})$ is already an atomic event, then the next method generated by τ is a response event $(\tau, \mathsf{CmtRet}\ O.f(\boldsymbol{x}))$ whose observations give the operation's return value. Otherwise, $O.f(\boldsymbol{x})$ 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). There is no explicit abort event; we model abort by a series of cancellation steps.

3.4 Specifications and Implementations

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 $\ell_1 = \ell \cdot (\tau, \mathsf{lvk}\ O.f(\boldsymbol{x}))$ and $\ell_2 = \mathsf{specLog}_{O.f(\boldsymbol{x})}(\ell_1)$. Let $\ell_{2,p}$ be any prefix of ℓ_2 and $\ell_{2,s}$ 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.

3.5 Parameterized base operations

We require a prefix-closed predicate on logs $\mathsf{allowed}(\ell)$ that indicates whether $\forall i \in [0, len(\ell) - 1]$ that $obs_i(\ell)$ is valid according to the sequential specifications of the objects. For convenience we will also write $\ell\ \mathsf{allows}\ n$ 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 allowed extensions of ℓ_1, are also allowed extensions to 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_1, that we can't also make of ℓ_2. 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. This relation is intentionally asymmetric because contextual refinement (defined later) only requires inclusion. Notice that one can always define a trivial observation function such that all traces will be allowed, but such definitions arent useful. Consider an example of a simple natural number counter, initialized to 0 and the trace $(\tau, \mathsf{increment}) \cdot (\tau, \mathsf{decrement}) \cdot (\tau, \mathsf{decrement})$. Here we would set up an observation function *obs* that returns "fault" for the third operation.

We also require an abstract version of the 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:

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

3.6 Inverses and conflict

We assume that for every operation a, there is an *inverse* operation a^{-1}, which is to be exactly such that $\forall \sigma \in \Sigma,\ a^{-1}(a(\sigma)) = \sigma$. Unfolding the structure of a, we say that $O.f^{-1}(\boldsymbol{y})$ is the function such that $\forall \sigma,\ O.f^{-1}(\boldsymbol{y})(O.f(\boldsymbol{x})(\sigma)) = \sigma$. Notice that constructing an inverse operation f^{-1} may require arguments other than those passed to f. The inverse operations are constructed dynamically, and in the worst case, such as with some write operations, the original state σ might have to be passed as an argument to the inverse operation. Many existing implementations already have a requirement of inverses [19,33,35].

We define a *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 in one direction (left-moverness [28]) 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. We say $\log \ell_a$ *commutes* with $\log \ell_b$ with respect to $\log \ell$, if $\ell_a \overset{\ell}{\lhd} \ell_b$ and $\ell_b \overset{\ell}{\lhd} \ell_a$.

4 Vertical composition through abstraction

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

4.1 Abstract operations

The events in Figure 4 start in the direction of vertical composition: an abstract operation $O.f(\boldsymbol{x})$, with atomic semantics $S.f(\boldsymbol{x})$ is implemented via a series of transaction events involving base operations $a, b, ...$ We now discuss how an object implementation may construct abstract operations (mutations) and observations (return values) from the mutations and observations of these base operations. An object operation $O.f(\boldsymbol{x})$ is implemented with transactions in a particular way (unlike nested transactions). $O.f(\boldsymbol{x})$ consists of a transaction immediately within the body of the method:

$$f(\boldsymbol{x}) \triangleq \textbf{atomic}\{\ \ ...\ \ \textbf{cmt\_return}\ (inv, k);\}$$

Here, k is a depiction of the observation of the overall abstract operation $O.f(\boldsymbol{x})$. A thread calling this operation is modeled as the following event sequence:

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

The invocation of $O.f(\boldsymbol{x})$ 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 $(\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x})) \cdot \ell' \cdot (\tau, \mathsf{CmtRet}\ O.f(\boldsymbol{x}))$, where ℓ' comprises a sequence of base operations and abstract operation sequences. Furthermore, in such a case we call this abstract operation sequence, an $O.f(\boldsymbol{x})$ abstract operation sequence. We define the predicate $\mathsf{aos}_\tau(\ell, O.f)$ that holds for a segment ℓ when it is an $O.f(\boldsymbol{x})$ abstract operation sequence over the thread τ.

4.2 Well-formedness

We now give some well-formedness constraints on objects and threads. We first have a basic well-formedness constraint, requiring the objects to yield sensible event histories. We formalize this with an inductive predicate $\mathit{wfir}_{\tau,O.f}$ over logs. This predicate is omitted for lack of space but, 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 $O.f(\boldsymbol{x})$ must construct a corresponding abstract inverse $O.f^{-1}(\boldsymbol{y})$, 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. 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:

- $\mathsf{activeOps}_\tau(\ell) \subseteq \mathcal{T} \times \mathbf{Ops}$: the *subsequence* of events with basic operations in ℓ 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
- $\mathsf{activeOps}_{\neg\tau}(\ell) \subseteq \mathcal{T} \times \mathbf{Ops}$: the *subsequence* of events with basic operations in ℓ corresponding to all $\tau' \in \mathcal{T} \backslash \{\tau\}$ (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:

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

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

As an example, the second rule above intuitively 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.

Semantics

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

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

Fig. 5. The rules for Reversible Atomic Objects.

5 Compositional 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 a shared log ℓ. Threads invoke object operations $(\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x})))$. The implementation of these operations, provided by C_O or S_O generates events for base operations $a, b, ...$ 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 [24,6].

Definition 1 (RAO Game). *An RAO Game $\mathfrak{G} = (\mathcal{E}, V, \rightsquigarrow)$ is a game between a set of threads/transactions and an environment $\mathcal{E}$. 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 $T \subseteq \mathcal{T}$ and a mapping $\mathit{tm} : \mathcal{T} \to (\mathit{St} \times \mathit{Cd} \times R)$.*

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 5. 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 $\mathcal{E} : \mathcal{L} \to \mathcal{P}(\mathcal{T}) \to (\mathcal{L} \times \mathcal{T})$ 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 $\boldsymbol{ev} \cdot \triangledown$ or $\boldsymbol{ev} \cdot \mathsf{Term}$. These events are used to construct the 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 horizontal merge $\oplus_H$ is defined as:

$$(T_1, tm_1) \oplus_H (T_2, tm_2) \equiv \left(T_1 \cup T_2, \lambda\tau. \begin{cases} tm_1\ \tau \text{ if } \tau \in T_1 \\ tm_2\ \tau \text{ otherwise} \end{cases} \right)$$

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

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

where r_O contains the implementation of C_O which may consult the log ℓ in generating events. In particular, for any ℓ where $C_O(\ell)$ is defined, and for all st, c, we define $r_O(\ell)(st, c)$ to be equal to (st, c, ℓ'), where $\ell' = C_O(\ell)$.

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:

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

Here, r_S is such that for all $\ell \cdot (\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x}))$, and for all st, c,

$$r_S(\ell \cdot (\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x})))(st, c) = (st, c, \ell'),$$

where $\ell' = S_O(\ell \cdot (\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x})))$. Notice the difference between r_O above and r_S here, which is that r_S works only on the logs that end with the method invocation event. For what follows it is clear from the context when we are using the horizontal merge and when we are using the vertical composition, and thus use the symbol $\oplus$, without subscripts, for both cases. 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 programs instead composed with atomic specifications and that layers can be composed vertically. We begin by defining traces.

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

Definition 2 (Trace). *For a set of threads T, initial value tm^0, and environment $\mathcal{E}$ a trace $\Pi((T, tm^0), \mathcal{E})$ of the game is a sequence of the form*

$$\epsilon, \bot, (T, tm^0) \xrightarrow{\mathcal{E}} \ell_1, \tau_1, (T, tm^0) \xrightarrow{\tau_1} \ell_2, \bot, (T, tm^2) \xrightarrow{\mathcal{E}} \ell_3, \tau_3, (T, tm^2) \xrightarrow{\tau_3} ...$$

We lift observations to traces and 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 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 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 $\Pi(P \oplus C_O, \mathcal{E}_A) \preccurlyeq_{\widehat{obs}} \Pi(P \oplus S_O, \mathcal{E}_B)$.*

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 1. *For any object O we have*

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

Proof. The proof can be found in the Appendix.

6.1 Vertical composition of contextual refinement between implementations and specifications

Theorem 2. *Let O and Q be two objects. Then*

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

Proof. The proof can be found in the Appendix.

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 inverse (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 further require that a well-formed thread will take note of these inverse operations and act appropriately.

As an example, consider the following log:

$$\ell = (\tau_1, \mathsf{lvk}\ \_), (\tau_1, a), (\tau_1, \triangledown), (\tau_2, \mathsf{lvk}\ \_), (\tau_2, b), (\tau_2, \triangledown),\\ (\tau_3, \mathsf{lvk}\ \_), (\tau_3, c), (\tau_3, \triangledown), (\tau_1, \triangledown_e), (\tau_2, \triangledown_f), (\tau_3, \triangledown_g)$$

the last three $\triangledown$ events indicate that threads τ_1, τ_2, τ_3 (resp.) are stuck trying to perform operations e, f, g respectively. 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* [41,40,42] 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 $\mathsf{deadlocked}(\ell)$ is the set of deadlocked threads and $\mathsf{oldest}(T, \ell)$ indicates that thread whose lvk event is earliest in the log. Now, we can define the environment as:

$\mathcal{E}_{cm}(\ell, T)$:
  let pause $\ell\ \tau$ = rev mkSeq $\{(\tau, a^{-1}) \mid \forall (\tau, a) \in \mathsf{activeOps}_\tau(\ell)\}$ in
  $\lambda\ \ell\ T$.
    let T' = deadlocked(ℓ) in
    if $T' = \varnothing$ then choose(ℓ, T)
    else let τ = oldest(T',ℓ) in
          (concat [] (map (pause ℓ) $T' \backslash \tau$), τ)

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 that 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 Discussion

We now discuss existing transactional implementations and how they are captured by the reversible atomic object model. We then describe how our semantics gives rise to a novel transactional universal construction.

8.1 Instances and examples

To our knowledge there are currently no theories or systems that implement this kind of multi-level vertical composition (nVC) of transactional objects. We offer the first example of such a thing in Section 2.

Coarse grained transactions [25] and Push/Pull [24] offer formal models of 1VC: threads executing transactions over a single layer of atomic objects. These models provide detail of thread-local semantics, such as the following rules from Push/Pull: APPLY, UNAPPLY, PULL, UNPULL. These rules involve careful tracking of thread-local semantics via thread-local logs. These rules are abstracted away in RAO but could be seen as a way to enforce the particular shape of the thread-local state $\mathscr{st}$, code c, and transition relation r.

As for the shared state, the Push/Pull rule has a singled shared log G. The Push/Pull rule PUSH is taken to be a (τ, a) event in RAO, and UNPUSH is taken to be a (τ, a^{-1}) event. We can thus reconstruct the Push/Pull shared log G by replaying the history of RAO events, appending an entry to G when there is a (τ, a) event and dropping the entry when there is a (τ, a^{-1}) event.

Our treatment of inverses in RAO also permit us to express checkpoints. Herlihy & Koskinen [22] showed that checkpoints can be established in a (boosted [19]) transaction over object operations. Expressing this in a reversible atomic object is straight-forward: there is, by design, a checkpoint in between each constituent object operation. In the RAO semantics (and in implementations), one returns to a checkpoint, simply by performing inverse operations. This inherent checkpoint nature is exploited by the contention manager: it can strategically choose from among these checkpoints so that it only inverts what is needed to escape deadlock.

Many read/write STM systems (TL2 [9], TinySTM [11], McRT [39], etc.) can be viewed as transactional objects (1VC), where we take the memory to be a base object. These systems are typically either optimistic [9,11,39] or pessimistic [31]. Our formal framework abstracts over thread implementation, leaving the question of opacity [17] up to the threads: they may choose to view or ignore effects of uncommitted transactions. Some systems permit transactions to view these effects, establishing dependences between transactions [38].

8.2 Transactional Universal Construction

The semantics $[\![P]\!]_{\mathfrak{C}}$ in Section 6 is based on a logical log and gives rise to a novel transactional version of the universal construction. As in the original universal construction of Herlihy [18], our construction provides a theoretical means of building a concurrent object merely from a sequential implementation of the

same object. The essence of the idea is for each thread τ to maintain a replica D_τ of the data-structure D. The logical state of D is defined based on a shared log. This shared log supports a single wait-free enqueue operation. Threads coordinate by attempting to enqueue the operations they wish to perform on D. Due to concurrency, this is a competition and the winner's method invocation becomes the next log entry. Meanwhile, if a thread observes that it was beat to the punch, it replays those other threads' operations on its local replica. The idea is similar to state machine replication [37].

With the advent of transactions, the universal construction is more subtle in a few ways. During a transaction threads are performing *multiple* operations, so we must track, via transaction identifiers $\mathcal{T}$, which operations correspond to which transaction. Additionally, transactions must append entries for Ivk and CmtRet. To this end, extend the *type* of an entry in the queue to permit identifiers and these additional kinds of events. Moreover, to support reversibility, the queue also permits inverse operations. 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.

For lack of space, here we summarize the key abstraction of our transactional universal construction. More detail including a pseudo-code implementation can be found in Appendix B. The key is an API called try_enqueue defined as:

$$\begin{aligned} cmd &::= \{a, \mathsf{Ivk}, \mathsf{CmtRet}\} \\ rsp &::= \{\mathsf{Success}, \mathsf{Conflict}, \mathsf{Abort}\} \\ \mathsf{try\_enqueue} &: \mathcal{T} \times cmd \to rsp \end{aligned}$$

Transactions, when they wish to append an entry to the queue invoke try_enqueue. Commands *cmd* are either a constituent operation a, transaction begin, or a transaction commit (respectively). The outcome of this synchronized operation is one of three possibilities. Success means the invocation was appended to the log. Conflict means the invocation conflicts with an entry already in the queue. In this case, the thread may either wait or invert some of its own operations. Finally, Abort means some operations of the transaction have been inverted (by the contention manager). In this case the thread must update its local state.

9 Related Work

Transactional boosting. As discussed in Section 1, reversible atomic objects can be seen as a generalization of boosting that permits multiple levels of vertical composition, per-layer implementation flexibility. Moreover, our work provides a formal account.

Nested transactions. The desire for vertically composable methods is not a new idea. In a similar spirit, nested transactions [35,32,34,33] aim toward a different form of vertical composition. This line of work, growing out of the database community [34], is concerned with complexity database-style transactions that

have inherent nesting and a need for serializability. Some of works also propose using commutativity and inverses as part of a multi-level scheme.

The nested (and database) transaction work, many of which predate linearizability [20], are not centered around the idea of an atomic concurrent object. Instead, these works are tied to a viewpoint of a single, global shared memory. Thus, as one builds nested transactions of increasing depth, all layers share access to the same base memory and all transaction layers are handled by the same monolithic transactional implementation. Several issues arise if one tries to extend such a view to establish vertical composition including: (1) imprecise definition of layers, (2) building mechanisms for isolation/encapsulation introduces immense complexity, and (3) the global nature of the runtime precludes the ability to use different synchronization protocols at different levels. Perhaps since their focus has not been on developing verified systems, to date, they have not explored a framework of atomic objects.

We believe that, in contrast, our theory of vertically composable transactional objects has several benefits:

- *Clearer semantics.* RAO has a clean semantics, while also permitting a variety of implementation choices, and vertical composition (see Sections 3-7).
- *Expressive contention management.* 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.
- *Ease of implementation.* Reversible atomic objects make it more straightforward for a non-expert to build complex concurrent systems. Many details are abstracted away from the programmer and can be accomplished by a compiler or runtime.
- *Beyond transactions.* Our theory allows one to incorporate atomic objects that aren't even transactional.

Reversible atomic objects have syntactic nesting, but not all nested transactions are reversible atomic objects. An example of a nested transaction that is not a reversible atomic object is given in Appendix C.

Transactional object implementations. Two recent works have aimed at developing real-world implementations of transactional data-structures. Herman et al. [21] 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. [1] 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 doesn't provide a general theory for vertical composition.

Universal construction. The original construction is due to Herlihy [18]. Crain et al. [8] 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 appear to unearth a general methodology. There is also a known similarity between multi-core universal construction and state machine replication [37] in distributed systems.

Other works. The Push/Pull model provided a formal semantics for describing a range of transactional implementations [24]. At a technical level, the Push/Pull model uses thread-local logs for describing detailed thread-local behavior. We abstract away these details. 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 in Section 8.1, 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 [4] for the case where shared and local variables are rolled back when a transaction aborts. Others [26] describe a method of specifying and verifying TM algorithms. They specify some transactional algorithms in terms of I/O automata [29] and this choice of language enables them to fully verify those specifications in PVS. With these works, we share the spirit of a layered-approach to contextual refinement. However, we are instead working on the context of transactional objects. Ziv et al. [45] describe how to compose transactions with other kinds of concurrency control such as two-phase locking and two-phase commit.

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.

We believe that this model will form the basis of new strategies for building complex high-performance software in a clean, modular manner. To this end, we plan to investigate algorithms and for the various components of this framework, for example, automating the translation from user-level syntax to pessimistic and optimistic implementations. We will also develop implementations that can be used in realistic settings, where a performance evaluation can be made.

References

1. 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).
2. 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.
3. Attiya, H., Gotsman, A., Hans, S., and Rinetzky, N. A programming language perspective on transactional memory consistency. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing* (2013), ACM, pp. 309–318.
4. 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.
5. Bansal, K., Koskinen, E., and Tripp, O. Commutativity condition refinement. In *EC2* (2015).
6. 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.
7. Cherem, S., Chilimbi, T. M., and Gulwani, S. Inferring locks for atomic sections. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08)* (2008), pp. 304–315.
8. 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.
9. Dice, D., Shalev, O., and Shavit, N. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)* (September 2006).
10. 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).
11. 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.
12. Filipović, I., OHearn, P., Rinetzky, N., and Yang, H. Abstraction for concurrent objects. *Theoretical Computer Science 411*, 51 (2010), 4379–4398.
13. Gehr, T., Dimitrov, D., and Vechev, M. Learning commutativity specifications. In *International Conference on Computer Aided Verification* (2015), Springer, pp. 307–323.
14. Golan-Gueta, G., Ramalingam, G., Sagiv, M., and Yahav, E. Automatic scalable atomicity via semantic locking. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 31–41.
15. Gotsman, A., and Yang, H. Linearizability with ownership transfer. In *CONCUR* (2012).
16. Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V., and Costanzo, D. Certikos: An extensible architecture for building certified concurrent os kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

17. 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.
18. Herlihy, M. Wait-free synchronization. *ACM Trans. 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. 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.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. Liang, H., Hoffmann, J., Feng, X., and Shao, Z. Characterizing progress properties of concurrent objects via contextual refinements. In *CONCUR* (2013), pp. 227–241.
28. Lipton, R. J. Reduction: a method of proving properties of parallel programs. *Commun. ACM 18*, 12 (1975), 717–721.
29. 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.
30. 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.
31. Matveev, A., and Shavit, N. Towards a fully pessimistic STM model. In *Proceedings of the 2012 Workshop on Transactional Memory (TRANSACT12)* (2012).
32. 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.
33. Moss, J. E. B. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues* (2006), vol. 28.
34. Moss, J. E. B., Griffeth, N. D., and Graham, M. H. Abstraction in recovery management. In *ACM SIGMOD Record* (1986), vol. 15, ACM, pp. 72–83.

35. 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.
36. 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.
37. Pedone, F., Guerraoui, R., and Schiper, A. The database state machine approach. *Distributed and Parallel Databases 14*, 1 (2003), 71–98.
38. 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.
39. 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.
40. 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.
41. 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.
42. 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.
43. Vafeiadis, V. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.
44. 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.
45. 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 contains 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 where 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 $i \leqslant n$, 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 $\ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x}))$ is invoked before $\ell_2 \cdot (\tau, \mathsf{Ivk}\ Q.g(\boldsymbol{x}))$ in ℓ, if $\ell = \ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x})) \cdot \ell_1' \cdot (\tau, \mathsf{Ivk}\ Q.g(\boldsymbol{x})) \cdot \ell_3$ and $\ell_2 = \ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x})) \cdot \ell_1'$. Furthermore, given a log ℓ, we say that an event (τ, e) *belongs* to $\ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x}))$ if $\ell = \ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x})) \cdot \ell_2 \cdot (\tau, e) \cdot \ell_3$ and ℓ_2 does not contain an event $(\tau, \mathsf{CmtRet}\ O.f(\boldsymbol{x}))$. 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 $\ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x}))$, the constituent events of ℓ_2 belong to $\ell_2 \cdot (\tau', \mathsf{Ivk}\ Q.g(\boldsymbol{x}))$ and $\ell_1 \cdot (\tau, \mathsf{Ivk}\ O.f(\boldsymbol{x}))$ is invoked before $\ell_2 \cdot (\tau, \mathsf{Ivk}\ Q.g(\boldsymbol{x}))$.

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

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

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

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 5. *For all ℓ, ℓ', ℓ'',*

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

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 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 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 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 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 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 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 2 and 3, $\ell_I \preccurlyeq_{obs} \mathsf{ro}(\mathsf{ki}(\ell_I))$ and by Lemma 1, $\ell_I \preccurlyeq_{\widehat{obs}} \mathsf{ro}(\mathsf{ki}(\ell_I))$. Furthermore, by Lemma 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 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 5 we obtain $\ell_I \preccurlyeq_{\widehat{obs}} \ell_A$, as required.

Lemma 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 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 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 an environment $\mathcal{E}$ and a 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 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 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 Pseudo-code for the Universal Construction

The transactional universal construction is a theoretical result and we don't expect it to be used as the implementation strategy in real systems. Nonetheless, concretizing the algorithm is helpful. To this end, Figure 6 provides pseudo code UniversalTransactionalObject class. Here we have distinct API methods for operations (try_op) versus committing (try_commit), whereas our summary in Section 8.2 has them merged. Our implementation 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 4 defines try_op, which takes a transaction identifier and a method invocation. After constructing a new entry (Line 6), 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. Second, it checks for any inversions of the current transaction by the contention manager. Because inverses are applied in reverse order, the loop continues looking for inverses until the end of the log, saving the last one found. 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 is created by applying the operations from the log, as long as they are from committed operations and the current transaction.

With try_op defined, it is straightforward to append begin and commit events. The begin function (Line 33) creates a new transaction identifier for the thread and calls try_op to append the begin message [2]. The call cannot result in a Conflict or Abort, because there are no operations yet from the transaction. Similarly, try_commit (Line 38) 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, saving 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. 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.

```
1  public class UniversalTransactionalObject {
2    private Transaction[] committed;
3    private Node logHead;
4    public Result try_op(Transaction tx, Invoc invoc) {
5      Thread thread = tx.thread;
6      Node entry = new Node(th, tx, invoc);
7      Node current = thread.lastSeen;
8      do {
9        Result result = null;
10       while (current.next != null) {
11         current = current.next;
12         if (tx.isConflict(entry, current)) {
13           result = new Conflict();
14           break;
15         } else if (tx.isInverse(current))
16           abort = current;
17       }
18       if (abort != null) result = new Abort(abort)
19       if (result != null) break;
20       Node next = current.decideNext(entry);
21       if (next == entry) result = new Success();
22     } while (result == null);
23     thread.local = new SeqObject();
24     current = logHead;
25     while (current != null) {
26       if (committed[current.tx] or current.tx == tx)
27         thread.local.apply(current.invoc);
28       thread.lastSeen = current;
29       current = current.next;
30     }
31     return result;
32   }
33   public Transaction begin(Thread thread, ObjectID obj) {
34     Transaction tx = new Transaction(thread, obj);
35     try_op(tx, begin);
36     return tx;
37   }
38   public Result try_commit(Transaction tx) {
39     Result result = try_op(tx, commit);
40     if (result is Success) {
41       committed[tx] = true;
42     }
43     return result;
44   }
45 }
```

Fig. 6. The universal construction for transactional objects.

C Example of a nested transaction

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. The following is an example of a nested transactions that *aren't* reversible atomic objects.

```
1  FileSystem.moveFile(p1, p2) {
2    atomic {
3      v = ht.get(p1)
4      ht.put(p2, v)
5      ht.remove(p1)
6    }
7    directory.move(p1, p2);
8  }
9
10 Directory.move(p1, p2) {
11   atomic {
12     all_paths.add(p2)
13     all_paths.remove(p1)
14     atomic {
15       Node node = tree.find(p2.directory)
16       node.addChild(p2.filename)
17     }
18     atomic {
19       Node node = tree.find(p1.directory)
20       node.removeChild(p1.filename)
21     }
22   }
23 }
```

Here 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 manner (not shown here).