Composing Concurrency Control

Ofri Ziv  
Tel Aviv University, Israel  
ofriziv@tau.ac.il

Alex Aiken  
Stanford University, USA  
aiken@cs.stanford.edu

Guy Golan-Gueta  
Tel Aviv University, Israel  
ggolan@tau.ac.il

G. Ramalingam  
Microsoft Research, India  
grama@microsoft.com

Mooly Sagiv  
Tel Aviv University, Israel  
msagiv@tau.ac.il

Abstract
Concurrency control poses significant challenges when composing computations over multiple data-structures (objects) with different concurrency-control implementations. We formalize the usually desired requirements (serializability, abort-safety, deadlock-safety, and opacity) as well as stronger versions of these properties that enable composition. We show how to compose synchronization protocols in a way which preserves these properties. Our approach generalizes well-known synchronization protocols (such as two-phase-locking and two-phase-commit) and leads to new synchronization protocols. We apply this theory to show how we can safely compose optimistic and pessimistic concurrency control and demonstrate the practical value of such a composition. For example, we show how we can execute a transaction that accesses two objects, one controlled by an STM and another by locking.

Categories and Subject Descriptors  D.1.3 [Programming Techniques]: Concurrent Programming

Keywords  composable concurrency

1. Introduction
Current approaches to designing concurrent software tend to assume that whatever concurrency control mechanism is selected, that one mechanism will be used exclusively throughout the system. In reality, large systems are usually assembled from existing small ones that were often designed separately, and most real systems that we are familiar with have a variety of concurrency control protocols that are occasionally intermingled in complex ways. In this thesis, we develop a theory that allows us to reason about and prove the correctness of compositions of different concurrency control protocols. Our results provide a framework for understanding when and why certain combinations of concurrency control strategies are correct or incorrect. In addition, while the focus of this thesis is on the theoretical foundations, our approach opens up the possibility of intentionally designing systems that mix different concurrency control protocols to take advantage of their different strengths with respect to performance and semantics.

Motivation  Figure 1 shows a simplified version of the Intruder benchmark from Jstamp (a Java version of the STAMP library [6]).

```
main() {
    while p = T1() do { T2(p); T3(); }
}
T1() {
    packet = packet_q.deq();
    return packet ;
}
T2(packet) {
    if flows_m.has_key(packet.flowID) {
        l = flows_m[packet.flowID];
        l.add(packet);
    } else {
        l = new List(packet);
        flows_m[packet.flowID] = l;
    }
    if packet.fragments == list.size() {
        s = l.to_string();
        flows_m.remove(packet.flowID);
    }
}
T3() {
    data = complete_q.deq();
    print isAttack(data);
}
```

Figure 1. An example with three atomic transactions. Intervals of the form $[\cdots]$ denote “synchronization windows” which are explained in the sequel.
**Intruder** is a Java program that consists of many threads, each executing the main procedure which invokes three other procedures \( T1, T2, \) and \( T3 \), each of which is required to execute atomically, as a software transaction. These transactions manipulate three shared data structures \( \text{packet}_g \) (a queue), \( \text{complete}_q \) (another queue), and \( \text{flows}_m \) (a map).

We first consider two well-known ways of ensuring that each of the procedures \( T1, T2, \) and \( T3 \) executes atomically. **Pessimistic Locking:** One approach is to associate a distinct lock with each of the data-structures (\( \text{packet}_g, \text{flows}_m \) and \( \text{complete}_q \)). \( T1 \) is realized by invoking lock and unlock operations on \( \text{packet}_g \) before and after the deq operation (at the points in the code indicated by \([1]\) and \([3]\)). Similarly, for \( T3 \), we replace \([5/2]\) by lock/unlock operations on \( \text{complete}_q \)’s lock. Since \( T2 \) accesses two data-structures, we use the two-phase-locking protocol within \( T2 \). This protocol requires each thread to execute in two phases: a “growing” phase where the thread may acquire locks but not release locks, followed by a “shrinking” phase in which the thread may release locks but cannot acquire locks. Two-phase-locking can be enforced in \( T2 \) by replacing \([5/2]\) by lock/unlock operations on \( \text{flows}_m \) and \([5/2]\) by lock/unlock operations on \( \text{complete}_q \).

**Optimistic STM:** An alternative approach is to use an optimistic STM [17] that wraps each transaction with a lock. This protocol within \( T1 \) requires each thread to execute in two phases: a “growing” phase where the thread may acquire locks but not release locks, followed by a “shrinking” phase in which the thread may release locks but cannot acquire locks. Two-phase-locking can be enforced in \( T2 \) by replacing \([5/2]\) by lock/unlock operations on \( \text{flows}_m \) and \([5/2]\) by lock/unlock operations on \( \text{complete}_q \).

**Locking + Optimistic STM:** We could also consider a third, more aggressive and speculative, approach: We could choose the concurrency control solution for each data-structure independently. For instance, since operations on \( \text{packet}_g \) and \( \text{complete}_q \) involve a relatively high fraction of write operations, we can choose to use lock-based concurrency for these data structures. Since accesses to \( \text{flows}_m \) are read-heavy we may want to use an optimistic STM. This solution can be achieved by replacing \([5/2]\) by lock/unlock operations on \( \text{complete}_q \)’s lock. Explain how we can compose distinct choices of concurrency control for different parts of a program is the contribution of this thesis.

There are compelling reasons to consider the use of multiple concurrency control protocols within the same transaction, as explained below.

**Convenience and Expressiveness.** A common scenario is that a project needs to use an existing library whose concurrency control properties have already been chosen. Perhaps the library uses locks, or perhaps the library is not even thread-safe and needs to be wrapped in STM atomic blocks, or perhaps the library is based on lock-free data structures. A flexible approach that allows us to combine different concurrency control protocols is obviously very convenient.

**Performance.** As Figure 2 illustrates, combining locking with STM yields better performance than using just a locking-based or an STM-based solution for the Intruder benchmark. In fact, parallelism yields almost no benefits with the locking-based or STM-based approaches, because using either choice exclusively is a very poor fit for some operations: The queue data-structures (\( \text{packet}_g \) and \( \text{complete}_q \)) are more efficient with locking as they have high contention writes to the head and tail of the queue, whereas the map data-structure \( \text{flows}_m \) has an overall very low contention and so is better suited to optimistic concurrency. In general, the ability to compose different concurrency control strategies leads to more implementation choices, with potential performance benefits.

Here is another example to motivate flexible composition of concurrency protocols and the need for a general theory to enable it. Since \( T3 \) consists of a single operation on \( \text{complete}_q \), it is safe to use an off-the-shelf optimized linearizable [21] queue implementation (without any extra concurrency control) for this queue? The answer depends on the choices made for the other data structures. Since \( T2 \) accesses both \( \text{flows}_m \) and \( \text{complete}_q \), the choices made for \( \text{flows}_m \) and \( \text{complete}_q \) must be compatible with each other. A pessimistic \( \text{flows}_m \) that guarantees never to abort a transaction is compatible with a linearizable \( \text{complete}_q \), but an optimistic \( \text{flows}_m \) that may abort is not compatible with a linearizable \( \text{complete}_q \).

**Contributions** The main contribution of our work is a theory for correctly composing concurrency control protocols. In Section 2, we introduce a uniform formalism and framework that allows us to precisely formulate the problem. We refer to a shared data-structure used within a transaction, associated with its own concurrency control protocol, as an object. The concurrency control protocol on an object prescribes how a client (a transaction over that single object) must execute a sequence of operations on that object to ensure that the sequence appears to execute atomically.

In Section 3, we present the first key component of our theory. Protocols usually guarantee serialization windows: an interval in the sequence of operations performed on the underlying object such that any point in the interval can serve as a serialization point for the transaction (for that object). Intuitively, a transaction that accesses multiple objects is serializable if the serialization windows for all accessed objects overlap, guaranteeing they have at least

![Figure 2. The performance of the Intruder benchmark under three synchronizations. The dotted line represents two-phase locking. The dashed line represents ScalaSTM [4]. The solid line represents a combination (composition) of the previous two techniques. Measurements were performed on a Xeon E5620 machine with 2 quadcore CPUs, each core featuring two hardware threads (i.e., 16 hardware threads overall) on a workload of 10,000 flows, each of up to 512 bytes.](image-url)
one serialization point in common. An interesting aspect of our formalism is that it generalizes and subsumes two-phase-locking (which can be seen as a composition of multiple objects that each use their own locks) as well as two-phase-commit (which can be seen as a composition of multiple objects that use protocols that may abort).

In Section 4, we address the challenges introduced by protocols that may abort. In Section 5, we consider the problem of avoiding deadlocks.

In Section 6 we consider a more subtle and challenging issue. A transaction that reads an inconsistent state (i.e., a state not expected when all transactions truly execute atomically) may become non-terminating because of this inconsistency. Avoiding such behavior requires a stronger property called opacity [15], which requires that all transactions, including those that may be aborted, see only a consistent state. In this section, we show that the overlapping window condition presented earlier is insufficient to guarantee opacity. We present stronger conditions that are sufficient to ensure that the composed protocol enjoys the opacity guarantee. These conditions are sufficient, e.g., to show that if every serial execution of a client of a protocol terminates, then every interleaved execution will terminate as well.

Section 7 presents a preliminary evaluation of the benefits of composing different kinds of concurrency control protocols in a single application. As mentioned previously, our primary aims are foundational and we do not attempt to undertake a thorough evaluation of the benefits of mixing concurrency control strategies. Our purpose with these experiments is merely to demonstrate for the reader that combining different concurrency control strategies has potential applications to improve software development (such as providing a way to combine STMs with I/O operations) while also opening up more possible implementation strategies with a wider range of performance characteristics. Section 8 discusses related work.

Proofs and other extra material can be found in the technical report [25].

2. Preliminaries

In this section we formalize the standard notions of histories, object specifications, synchronization protocols and serializability.

Objects and Events. We refer to data-structures such as the maps and queues in Figure 1 as objects. An object o exposes a set of methods. An operation is a tuple of the form m(e1, ..., eM) where m is an object method name and e1, ..., eM are its arguments. For an operation with no arguments we omit the parentheses and write only m. An event is a tuple ⟨t, o, op, r⟩ where t is a transaction identifier, o is an object, op is an operation, and r is a return value (r is omitted when the operation does not have a return value). An event captures both an operation invocation as well as its return value. This is a convenient simplification — where necessary, we will distinguish the invocation and return events. We assume that each object is a unit of encapsulation: no mutable state is shared by different objects, and the state of an object can be accessed only through its methods.

Histories. A sequential history (or history) h is a finite sequence of events (along with a unique id for each event to identify it, which we usually omit). A subhistory of h is a subsequence of the events of h. We use the following notations:

- h | o is the subhistory of h consisting of all events in h on object o.
- h | O is the subhistory of h consisting of all events on objects in the set of objects O.
- h | t is the subhistory of h consisting of all events executed by transaction t.
- h | T is the subhistory of h consisting of all events executed by some transaction t ∈ T.
- h \ t is the subhistory of h obtained by omitting all events executed by transaction t.
- hh′ is the concatenation of history h′ to the end of history h.
- α(h) is the set of objects appearing in history h.
- α(P) is the set of objects appearing in the set of histories P.
- Γ(h) is the set of transactions t such that h | t is non-empty.

A t-history is a history where all events are executed by transaction t (a h-history may also be empty). We say two histories h and h′ are equivalent histories if for every transaction t, h | t = h′ | t. A history is non-interleaved if its transactions are not interleaved — if h = (h1 | t1)(h2 | t2)···(hn | tn) where Γ(h) = {t1, ..., tn}.

Let S1 and S2 be two sets of histories such that α(S1) ∩ α(S2) = ∅. We define S1 • S2 to be the maximal set such that {h | (h \ α(S1)) ∩ (h \ α(S2)) = ∅}. This definition represents the unconstrained composition of S1 and S2 and reflects the fact that different objects share no state and operations on one object do not affect other objects.

Specifications and Protocols. We define a specification S to be a prefix-closed set of histories. A history is legal (with respect to S) if the specification contains it. We denote the specification of a single object o by Ho which must satisfy α(Ho) = {o}. Note that a specification combines both preconditions (to be satisfied by clients that use an object) and postconditions (guarantees provided by the object implementation).

Example 1. Consider a sequential queue Q that provides methods enq and deq to, respectively, enqueue and dequeue an item from the queue. The specification HQ of the queue captures the FIFO semantics of the queue in the obvious way (but ignores the transaction identifiers). As another example, the specification Hl of a lock L provides methods acquire and release can capture the usual semantics of locks: Hl consists of all prefixes of histories with strictly alternating acquire and release events, with the acquiring transaction releasing a lock before any transaction can acquire the lock again.

Let h[q1 → a][q2 → a] denote the history obtained from h by replacing every occurrence of objects q1 and q2 in h by o. We extend this notation to any set of histories. We define the composite object o1 × o2 to be a new object o whose specification is given by (Ho1 × Ho2)[o1 → o][o2 → o]. The composite object exposes the set of methods exposed by each of its underlying primitive objects. The composition is defined only if the objects have no method names in common. Every (composite) object represents a set of primitive objects. We say that two objects are disjoint if the underlying sets of primitive objects are disjoint.

Let O = {o1, ..., on} be a set of objects. We define the specification HO by:

\[ H_O = H_{o_1} * H_{o_2} * \cdots * H_{o_n}. \]

We define a (synchronization) protocol P over a set of objects O (with given specifications) to be a prefix-closed subset of HO. Thus, a protocol enforces additional constraints on how the objects may be used beyond the individual specifications of the objects.

Example 2. We now formalize a (well-known) locking protocol that utilizes a lock L to synchronize concurrent accesses of the
Figure 3. Serialization point mapping examples. $h_\text{m}$, $h_{\text{m'}}$ are the histories induced by applying the functions $m$, $m'$ (defined in Example 3) on the history $h$, respectively.

sequential queue $Q$. Let $QL$ denote the composite object $Q \times L$. We define the protocol $SLQ$ (Single Lock Queue) to be the set of all histories in $H_{QL}$ that are prefixes of well-locked histories, where a multi-transaction history $h$ is said to be well-locked if for every transaction $t$ in $h$, $h \mid t$ is well-locked, and a single-transaction history $h$ is said to be well-locked if it consists of an acquire event, followed by a sequence of enq/deq events, followed by a release event.

3. Abort-Free Serializability

We now present the first component of our theory for composing protocols, which generalizes the essence of two-phase locking. We will restrict ourselves to histories without abort transactions in this section.

Definition 1 (Serializability). A history $h$ is said to be abort-free serializable with respect to a set of histories $S$ if there is a non-interleaved history $h'$ in $S$ that is equivalent to $h$. We say that a protocol $P$ over a set of objects $O$ guarantees abort-free serializability if every history in $P$ is abort-free serializable (with respect to $H_O$).

Single-Object Histories. We begin by considering histories and transactions over a single object. We first formalize the notion of a serialization point for a transaction. Intuitively, it is a point in time where the transaction appears to take place atomically (inspired by the notion of a linearization point in linearizability).

Definition 2 (Serialization Point Mapping). Let $t$ be a history such that $|\alpha(h)| = 1$ and let $\Gamma(h) = \{t_1, \ldots, t_k\}$. Let $m$ be a function that maps every transaction in $h$ to an event in $\Gamma(h)$. Function $m$ is said to be a serialization point mapping for $h$ (with respect to a specification $S$) if $(h[t_1])(h[t_2]) \cdots (h[t_k])$ is legal for any permutation $(t_{i_1}, \ldots, t_{i_k})$ of $\Gamma(h)$ that satisfies $m(t_{i_1}) \leq_h m(t_{i_2}) \leq_h \cdots \leq_h m(t_{i_k})$.

In other words, a function $m$ is a serialization point mapping for a history $h$ if any non-interleaved ordering of the transactions of $h$ consistent with the order induced by $m$ is legal.

Example 3. Figure 3 shows a history $h \in Q$ (defined in Example 1) containing 3 transactions $t_1$, $t_2$, $t_3$, and two functions $m$, $m'$ from $h$ transactions to its events such that: $m(t_1) = (t_1, q, \text{enq} 3)$, $m(t_2) = (t_2, q, \text{enq} 2)$, $m(t_3) = (t_3, q, \text{deq} 1)$. And $m'(t_1) = (t_1, q, \text{enq} 1)$, $m'(t_2) = (t_2, q, \text{enq} 2)$, $m'(t_3) = (t_3, q, \text{deq} 1)$. The histories $h_m$, $h_{m'}$ are obtained by applying $m$, $m'$ on $h$'s transactions, respectively. Since $h_m \in H_Q$, $m$ is a serialization point mapping for $h$. On the other hand $h_{m'} \notin H_Q$ (as the deq operations invalidate the queue specification), therefore $m'$ is not a serialization point mapping for $h$.

Given any history $h = [e_1, \ldots, e_n]$, we define a window in $h$ to be a pair of events $[e_i, e_j]$ where $e_i \leq_h e_j$. A window identifies a non-empty sequence of consecutive events in $h$. Two windows $[e_i, e_j]$ and $[e_{i'}, e_{j'}]$ in a history are said to overlap if there exists an event $e_k$ such that $e_i \leq_h e_k \leq_h e_j$ and $e_{i'} \leq_h e_k \leq_h e_{j'}$. Protocols typically guarantee that every transaction executes atomically at any point within the window. We formalize this notion below.

Definition 3 (Single-History Serialization Window-Mapping). Let $h$ be a history. A window mapping for $h$ is a function $w$ that maps every transaction in $h$ to a window in $h$. We say that $w$ is a serialization window mapping for $h$ (with respect to a specification $S$) if every function $m$ that maps every transaction in $h$ to some event in $w(t)$ is a serialization point mapping for $h$ (with respect to $S$).

Note that a serialization window mapping may be seen as a representation of a set of valid serialization-orderings of the transactions in a history. Looking back at protocol $Q$ and the history $h$ in $Q$ (described in Figure 3), the function $w_5$ defined by $w_5(t_1) = [3, 3]$, $w_5(t_2) = [1, 2]$, $w_5(t_3) = [4, 5]$ is a serialization window mapping for $h$. The function $w_6'$ defined by $w_6'(t_1) = [1, 5]$, $w_6'(t_2) = [1, 2]$, $w_6'(t_3) = [3, 5]$ is not a serialization window mapping for $h$, as $m'$, defined earlier, maps $h$'s transactions to events in $w_6'$ windows but is not a serialization point mapping.

Example 4. Every history $h$ in the Single Lock Queue protocol has a serialization window mapping $w$ defined by $w(t) = [e_i, e_j]$ where:

- $e_i$ is the acquire method executed by $t$;
- if $h[t]$ contains a release method, then $e_j$ is the release method executed by $t$, otherwise $e_j$ is the last event in $h$.

The following result holds trivially:

Theorem 1. Any history with a serialization point mapping (or a serialization window mapping) is serializable.

We now extend the preceding formalism and results (presented for single histories) to protocols (sets of histories).

Definition 4 (Window-Serializable Protocols). We say that $(P, w)$ is a window-serializable protocol if $P$ is a protocol and $w$ is a function that maps every history $h \in P$ to a single-history serialization window mapping for $h$. In this case, we refer to $w$ as a serialization window mapping for $P$. We will abbreviate $w(h)(t)$ to $w(t)$.

An example of window-serializable protocol is the Single Lock Queue with the window mapping function presented in Example 4.

Theorem 2. If $(P, w)$ is a window-serializable protocol, then $P$ is serializable.

Composing Window-Serializable Protocols. Let $(P_1, w_1)$ and $(P_2, w_2)$ be window-serializable protocols for two distinct objects $o_1$ and $o_2$, respectively. We now consider how we can ensure serializability of histories over $o_1$ and $o_2$.

Consider any history $h \in P_1 \times P_2$ (the unconstrained composition of $P_1$ and $P_2$). Such a history correctly follows the protocols $P_1$ and $P_2$ for both objects $o_1$ and $o_2$. However, this is not sufficient to guarantee that $h$ is serializable. It simply implies that we can find a suitable serialization ordering for each object independently, but
there may not be a single consistent serialization ordering that is valid for both objects.

We abuse notation and extend \( w_1, w_2 \) to be a (partial) window-mapping function for \( P_1 \times P_2 \). Specifically, if \( h \in P_1 \times P_2 \) and \( w_i(h|_o, t) \) is defined then \( w_i(h, t) = w_i(h|_o, t) \) (otherwise \( w_i(h, t) \) is undefined).

**Definition 5 (Overlapping Mapping).** Let \( h \) be a history (over the two disjoint objects \( o_1, o_2 \)). Let \( w_{o_1} \) and \( w_{o_2} \) be two serialization window mappings for \( h \mid o_1 \) and \( h \mid o_2 \), respectively. We say that \( h \) has overlapping windows if for every transaction \( t \) in \( h \) that accesses both \( o_1 \) and \( o_2 \), the windows \( w_{o_1}(t) \) and \( w_{o_2}(t) \) overlap.

**Theorem 3.** Any history that has overlapping windows is serializable.

We now show how we can compose two window-serializable protocols to obtain another window-serializable protocol over the composite object.

**Definition 6 (Overlapping Protocol Composition).** Let \( (P_1, w_1) \) and \( (P_2, w_2) \) be two window-serializable protocols for two distinct objects \( o_1 \) and \( o_2 \). We define \( (P_1, w_1) \otimes (P_2, w_2) \) to be the pair \( (P, w) \), where \( P \) is the set of all \( h \in P_1 \times P_2 \) such that \( h \) has overlapping windows (with respect to \( w_1 \) and \( w_2 \)) and \( w \) is a window mapping function for \( P \) defined as follows:

\[
  w(h, t) = \begin{cases} 
    w_1(h, t) & \text{if } o(h|t) = \{o_1\} \\
    w_2(h, t) & \text{if } o(h|t) = \{o_2\} \\
    w_1(h, t) \cap w_2(h, t) & \text{otherwise}
  \end{cases}
\]

**Theorem 4.** Let \( (Q_1, w_1) \) and \( (Q_2, w_2) \) be window-serializable protocols (for disjoint objects). Then, \( (Q_1, w_1) \otimes (Q_2, w_2) \) is a window-serializable protocol.

Note that any subset of a window-serializable protocol is also window-serializable. Theorem 4 provides a sufficient condition to prove a protocol is window-serializable. We can now deduce a protocol \( P \) is window-serializable if it is a subset of the overlapping composition of two protocols. Furthermore, Definition 6 also provides us with an overlapping mapping for \( P \). Intuitively, one can think of 2PL protocol on two objects as a composition of two Single-Lock protocols.

**Example 5 (Two-phase locking).** Let \( QL_1, QL_2 \) be two different objects \( Q \times L \) (each with a different pair of lock and queue). The two-phase locking protocol (2PL) consists of the prefixes of every history \( h \in H_{QL_1,QL_2} \) that for every transaction \( t \) and object \( o \) the following conditions are satisfied: (i) \( h|t \) does not contain a release that precedes an acquire; (ii) \( h|o \) is well-locked.

**Theorem 5.** Let \( (SL_1, w_1) \) and \( (SL_2, w_2) \) be two window-serializable Single Lock protocols on \( QL_1 \) and \( QL_2 \), respectively. \( (2PL, w) = (SL_1, w_1) \otimes (SL_2, w_2) \), where \( w \) is the window mapping induced by \( (SL_1, w_1) \otimes (SL_2, w_2) \).

**Visibility Status.** To compose protocols, in the presence of aborts, we need extra information about the protocols. This takes the form of a function that lets us constructively identify the existentially quantified set \( T \) of transactions in the above definition of serializability. A status mapping, for a protocol \( P \), is a function that maps every \( (h, t) \) pair where \( h \in P \) and \( t \in h \) to one of three values invisible, visible, or dual, with the restriction that it map committed transactions to visible or dual and aborted transactions to invisible. Essentially, the status invisible indicates that the transaction should be excluded, and the status visible indicates that the transaction should be included, while the status dual indicates that it is

**Multi-Object Histories.** The preceding approach can be directly used to create window-serializable protocols over more than two objects by iteratively composing protocols over each individual object (since the overlapping composition of two window-serializable protocols is a window-serializable protocol over the composite object). Such an approach yields, for example, the 2PL protocol over \( n \) objects \( QL_1, \ldots, QL_n \), each protected by a lock.

**4. Abortable Serializability**

Optimistic protocols permit potentially conflicting transactions to execute concurrently and resolve this conflict by aborting one of these transactions. Such aborts complicate protocol composition. Consider a transaction over two objects in which one of the objects aborts the transaction. The only viable option in this case is to have the other object abort the transaction as well. This imposes constraints on the histories we should permit if one of the protocols cannot support aborts at arbitrary points during execution. We formalize these constraints in this section.

**4.1 Theory**

An operation invocation may return a special value aborted to indicate that the transaction has been aborted. An event is said to be an aborted event if its return value is aborted. \( t \) is said to be an aborted transaction in \( h \) if \( h \mid t \) contains an aborted event. We assume a special (last) operation that is called by each transaction to indicate its completion. A transaction is said to be complete if it has called this operation and it is said to be running otherwise. A history is said to be complete if all its transactions are complete. We refer to a completed, but not aborted, transaction as a committed transaction.

**Definition 7 (Serializability).** A history \( h \) is said to be serializable (with respect to a specification \( S \)) if there exists some set of transactions \( T \) that includes all of \( h \)’s committed transactions but none of its aborted transactions such that \( h|T \) is abort-free serializable. A protocol \( P \) is said to be serializable (with respect to \( S \)) if every history in \( P \) is serializable.

**Example 6.** We now describe an optimistic map object OM. This object provides the usual map operations get and add, which, respectively, retrieves the value associated with a key and updates the value associated with a key in the map. In addition, the map provides the methods begin-transaction and commit. The commit method returns a value success or aborted. The operations get and add can also return the value aborted in addition to the usual values. This allows early detection of inconsistency and is important, as will become clear later. In a serializable protocol OMP over OM every transaction is required to first execute the begin-transaction method, then execute any sequence of get and add, and end by invoking the commit method. An implementation of such a protocol can be realized by wrapping an STM such as TL2 [8] around a sequential map implementation, but these details are beyond the scope of this thesis.
okay to either include or exclude the transaction. We further require that (a) the status of a transaction can be determined using just its sub-history: i.e., for any $h_1, h_2 \in P$, if $h_1 | t = h_2 | t$, then $f(h_1, t) = f(h_2, t)$, and (b) The status of a visible transaction does not change: for any $h_1, h_2 \in P$. If $h_1$ is a prefix of $h_2$ and $f(h_1, t)$ is visible, then $f(h_2, t)$ must also be visible.

We now generalize our earlier definition of window serializable protocols: in the generalized setting, the window mapping function needs to define a serialization window only for transactions labelled visible or dual.

**Definition 8.** Given a status mapping function $f$ and a history $h$, we define the set $\text{witness}(h, f)$ to be the set of all $T \subseteq \Gamma(h)$ such that $T$ includes all the visible transactions, but none of the invisible transactions.

**Definition 9.** We say that $(P, w, f)$ is an abortable ws-protocol if $P$ is a protocol, $f$ is a status mapping function for $P$, and $w$ a serialization window mapping for the set of histories $\{h | T \subseteq \Gamma(h) : h \in P; T \in \text{witness}(h, f)\}$.

**Lemma 1.** If $(P, w, f)$ is an abortable ws-protocol, then $P$ is serializable.

**Example 7.** The definitions of $w$ and $f$ for an STM-based implementation of an object depend on the STM. For a TL2-based implementation of the optimistic map OM, the following definitions work. The status mapping function $f$ is fairly simple: $f(h, t)$ is visible if $t$ has successfully committed in $h$, and invisible otherwise. Every completed history $h$ has a serialization window mapping $w$ defined by $w(t) = [e_i, e_j]$ where:

- If $t$ is a read-only transaction (no add operation was executed by $t$), then $e_i$ is the begin-transaction executed by $t$ and $e_j$ is the first get method executed by $t$ or the last event in $h$ if no get method was invoked by $t$.
- Otherwise, if $t$ executed a successful commit, then $e_i$ and $e_j$ are both the commit event executed by $t$.

Let $\text{OMP}'$ denote $(\text{OMP}, w, f)$. Then $\text{OMP}'$ is an abortable ws-protocol. The rationale for this definition is the way TL2 works: TL2 validates real operations when they occur and again before commit. Write operations, however, update only a local copy — the write operations are validated and the original locations updated only at commit time. However, read-only transactions are not validated again at commit time as an optimization.

**Composition.** Now we define the conditions for composing two abortable protocols. Let $(P_1, w_1, f_1)$ and $(P_2, w_2, f_2)$ be two abortable ws-protocols for two different objects $o_1$ and $o_2$, respectively. We abuse notation and extend each $f_i$ to be a (partial) status mapping function for $P_1 \times P_2$. Specifically, we define $f_i(h, t)$, where $h \in P_1 \times P_2$ and $t \in h$, to be the status determined by $f_i$ if $t$ accesses $o_i$ and $(f_i(h, t)$ is dual if $t$ does not access $o_i$.

We say that a history $h \in P_1 \times P_2$ is compatible if for every transaction $t$ in $h$, (a) Either $f_1(h, t) = f_2(h, t)$ or one of $f_1(h, t)$ and $f_2(h, t)$ must be dual, and (b) If $w_1(h, t)$ and $w_2(h, t)$ are both defined, then they overlap.

Let $P$ denote the set of all $h \in P_1 \times P_2$ such that $h$ is compatible. We define a window-mapping function $w$ as follows: $w(h, t)$ is $w_1(h, t) \cap w_2(h, t)$ if both $w_1(h, t)$ and $w_2(h, t)$ are defined, $w(h, t)$ is $w_i(h, t)$ if $w_i(h, t) = \text{dual}$ for $i = 1, 2$, and $w(h, t)$ is undefined otherwise. We define a status-mapping function $f$ (for compatible histories only) as follows: $f(w, t) = f_1(w, t)$ if $f_2(w, t)$ is dual, $f(w, t)$ is $f_2(w, t)$ if $f_1(w, t)$ is dual, and $f(w, t) = f_1(w, t)$ ($= f_2(w, t)$) otherwise.

We define $(P_1, w_1, f_1) \circ (P_2, w_2, f_2)$ to be $(P, w, f)$.

**Theorem 6.** If $(Q_1, w_1, f_1)$ and $(Q_2, w_2, f_2)$ are abortable ws-protocols (for disjoint objects), then $(Q_1, w_1, f_1) \circ (Q_2, w_2, f_2)$ is an abortable ws-protocol.

Note that any subset of an abortable ws-protocol is also an abortable ws-protocol. Theorem 6 provides a sufficient condition to prove a protocol is an abortable ws-protocol.

Note that the status compatibility condition (in the composition) implicitly encodes the constraint that if one object in a transaction aborts, then the other object must abort too. (Otherwise, when the transaction ends, one object will be labelled invisible, while the other object will be labelled visible, and such a history is not compatible and is not permitted in the composed protocol.) This typically requires objects to support an explicit (client-initiated) abort operation. Many optimistic protocols can easily support such a client-initiated abort. However, pessimistic protocols, such as SLQ, typically do not support aborts. So, how can we compose such protocols? We now show how we can explain the correctness of composing optimistic and pessimistic protocols in examples such as Figure 1 using our preceding theorem.

**Example 8.** We define a status-mapping function $f$ for the SLQ protocol as follows: $f(h, t)$ is dual if $t$ has performed no queue operation (enq or deq) and visible otherwise. The window-mapping function $w$ is defined for SLQ as usual (see Example 4). Let SLQ denote $(\text{SLQ}, w, f)$. It can be shown that SLQ is an abortable ws-protocol. This definition captures the fairly simple intuition that a pessimistic object can indeed support an abort by a transaction $t$ if $t$ has not performed any updates yet. While simple, it suffices to enable the composition of protocols in examples such as Figure 1, as shown below.

The following is an example of a protocol composed of TL2 and Single Lock protocols:

**Example 9 (OM+SLQ).** Consider the abortable ws-protocols SLQ and $\text{OMP}'$ defined in Example 8 and Example 7. We now present a sufficient condition for correctness of transactions over these two objects. Consider a set of histories $\mathcal{H} \subseteq \text{SLQ} \times \text{OM}$ such that every $h$ in $\mathcal{H}$ satisfies the following conditions for every transaction $t$ that accesses both objects: (i) if $h | t$ is aborted, it contains no invocations of enq or deq methods (on SLQ). (ii) Any commit event (on OM) in $h | t$ occurs between an acquire event (on SLQ) and a release event (on SLQ) or the end of the history if it contains no release event.

We claim that $\mathcal{H}$ is a subset of the $\text{OMP}' \circ \text{SLQ}'$ protocol. Hence, $\mathcal{H}$ itself is an abortable ws-protocol.

### 4.2 Applications

Using our theorem we can prove the correctness of using the OM and SLQ protocols in Figure 1. Assume we use a separate instance of SLQ for $\text{packet}_G$ and $\text{complete}_G$ and OM for $\text{flows}_M$. We replace $\delta$ by the OM begin-transaction method, and all other $[i,j]$, by SLQ acquire/release methods. We replace $\gamma$ by an invocation of the OM commit method: if the commit is successful, execution continues normally; if the commit aborts, we skip the subsequent conditional statement to directly execute $\delta$. (Note that when an aborted transaction ends, it must be retried as well. This is straightforward and we will ignore this aspect in this thesis.) Such an implementation guarantees that for every transaction accessing both the map and the queue objects, the OM serialization window is contained in the SLQ serialization window and all the histories are compatible. Therefore, our composed protocol is an abortable ws-protocol.

The well-known two-phase-commit protocol is another instance of our protocol composition. For certain simple optimistic transaction implementations, the serialization window of a transaction
consists of a single event, namely the commit event. It is not possible to compose two such protocols since there is no way to ensure that two different single-point windows overlap. Protocols that are designed to support two-phase-commit split the commit operation into two operations. The first operation validates that the transaction is consistent and can be committed (and acquires necessary locks to ensure this) and returns. The second operation completes the commit. For such protocols, the serialization window extends from the first operation (if it is successful) to the second operation, permitting a non-trivial composition of two such protocols. In this setting, our composition corresponds to exactly the two-phase-commit protocol.

A similar extension can be used to compose two instances of the TL2 protocol by splitting its commit operation into multiple operations. However, this splitting is more involved since TL2 does not acquire locks for reads.

Another useful application that our theorem enables is the use of an abortable protocol (e.g. STM) in transactions performing I/O operations.

**Example 10.** Consider a data-structure and a log file used in a program with a read transaction, which reads from the data-structure, and an update transaction, which writes to the data-structure and adds a line in the log file. Assuming the data-structure supports parallel reads, for a read-dominant program we would prefer using TL2 STM to synchronize access to the data-structure and a lock for the log file. To guarantee serializability, we acquire the lock before the STM commit. The subsequent log file operation is executed only if the STM commits successfully. In either case, the lock is released finally.

The above example is an abstraction of real systems, like levelDB [1] and journaling file-systems [26].

### 5. Progress Guarantees: Deadlock Avoidance

We now consider the problem of avoiding deadlocks in a composed protocol. To formalize the problem, we first refine our notion of an event, as in [21], and assume that histories may contain invocation events and response events, where an invocation event represents an invocation of an operation, and a response event represents a response to an invocation. For each transaction \( t \) in a history \( h \), \( h[t] \) applies a sequence of events, alternately issuing an invocation and then receiving the associated response. A response event matches an invocation event if it follows it in \( h \) and both have similar transaction identifier and object. (The event defined in Section 2 is essentially an invocation event immediately followed by a matching response event.)

We write \( \text{RUN}(h) \), to denote the set of running transactions at the end of history \( h \). We say that history \( h \) is a complete history, if \( \text{RUN}(h) = \emptyset \); otherwise we say that \( h \) is an incomplete history.

**Definition 10 (Unblocked transaction).** We say that a running transaction \( t \) in a history \( h \) is currently blocked (with respect to a set of histories \( S \) ) if \( h[t] \) ends with an invocation event and there is no matching response event \( e \) such that \( h \in S \). Otherwise, we say that \( t \) is currently unblocked in \( h \). We say that \( t \) is unblocked in \( h \) (with respect to a set of histories \( S \) ) if for any t-history \( h_t \) such that \( h_t \in S \), \( t \) is currently unblocked in \( h_t \).

The above notion of an unblocked transaction is analogous to the notion of obstruction-freedom: if no other transaction intervenes, then an unblocked transaction can keep executing without getting blocked.

**Definition 11 (Unblocked history).** An incomplete history \( h \) is existentially unblocked (with respect to a set of histories \( S \) ) if there exists some \( t \in \text{RUN}(h) \) such that \( t \) is unblocked in \( h \). An incomplete history \( h \) is universally unblocked (with respect to a set of histories \( S \) ) if for every \( t \in \text{RUN}(h) \), \( t \) is unblocked in \( h \).

**Definition 12 (Unblocked Protocol).** A protocol \( P \) is existentially unblocked if every incomplete history \( h \in P \) is existentially unblocked with respect to the protocol specification \( S \). A protocol \( P \) is universally unblocked, if every incomplete history \( h \in P \) is universally unblocked with respect to the protocol specification \( S \).

The above unblocked properties are satisfied by many existing concurrency control protocols. The existentially unblocked property is satisfied by tree-locking [2, 29], DAG-locking [2, 29], domination-locking [10], two-phase locking in which the locks are acquired in a consistent order [31], and by the synchronization described in [11]. All of these protocols ensure that (at least) one of the running transactions can safely continue alone without waiting for the other running transactions.

The universally unblocked property is satisfied by every serializable obstruction-free protocol [20] because obstruction-free protocols guarantee that each running transaction can make progress until completion — examples for such protocols are described in [17]. It is also satisfied by protocols like boosting [19] and TL2, because these protocols use timeouts to ensure that any transaction that waits too long is aborted (notice that in our framework an aborted transaction is a completed transaction).

**Theorem 7.** If \( P_1 \) is an existentially unblocked protocol and \( P_2 \) is a universally unblocked protocol, then \( P_1 \ast P_2 \) is an existentially unblocked protocol.

Our formalism can be extended to guarantee that a composition of two existentially unblocked protocols creates an existentially unblocked protocol. This can be achieved by introducing ordered overlapping composition, which enforces serialization window order in addition to the overlapping window requirement.

### 6.Opacity

A subtle issue with optimistic protocols is that, unless special care is taken, only committed transactions are guaranteed to be consistent. Speculative transactions may observe an inconsistent state and only subsequently detect that they should rollback. However, these inconsistencies may cause transactions to behave in unexpected ways (taking "impossible" branches, executing non-terminating loops, throwing an exception, etc.) even before the inconsistency is detected.

Many STMs take extra care to prevent inconsistent reads and guarantee that transactions always observe a consistent state (a property formalized as opacity [15]). E.g., the TL2 and LSA [27] algorithms use a global time-stamp to efficiently validate a transaction after each read, guaranteeing consistency for all intermediate states.

This problem re surfaced when we compose protocols, as illustrated by the example in Figure 4. In this example, we protect variable \( x \) using an STM like TL2 and variable \( y \) using a lock. The main program executes two transactions \( T_1 \) and \( T_2 \) in parallel. The invariant \( x == y \) is preserved by transaction \( T_1 \). Yet, transaction \( T_2 \) can observe a state where this invariant is violated, if it reads the old value of \( x \) (before \( T_1 \) commits) and the new value of \( y \) (after \( T_1 \) releases the lock). In this example, we have an interleaving history that is non-terminating even though all non-interleaving histories terminate.

Note that this example satisfies the requirements of the abortable ws-protocol composition from the previous section! This shows that we need something more if we wish to guarantee opacity for the composed protocol.
6.1 Formalism

Several challenges motivate the following formalism. First, we desire to formalize the desired guarantees in a general way so that it applies to both pessimistic and optimistic protocols. Second, we need to strengthen these guarantees so that we can inductively compose protocols with such guarantees and ensure that the resultant protocol has the same guarantee.

A candidate guarantee is to ensure that every transaction sees all the effects of another transaction, while still guaranteeing some notion of consistency. In this section we assume an event is an invocation event followed by its matching response event. We now abstractly describe several challenges motivate the following formalism. First, we define a relation that captures the notion that two histories are almost the same: the only asymmetry is that one or more elements of one history may abort, potentially producing behaviors not seen in a serial history. We say that a history is a serial history if for every non-empty history \( h \approx \) \( h' \setminus \Gamma(h) \) and every \( i \), the sub-history executed by transaction \( i \) in \( h' \setminus \Gamma(h) \) is complete in \( h \).

Property 2 says that actions of a dependable transaction are “left movers” (with respect to actions of other transactions), while property 3 is a slightly stronger property asserting that the actions of a dependable transaction do not interact (conflict) with any possible concurrent actions by other transactions. (Note that one can also derive \( h \approx h') \setminus \Gamma(h) \) in the case of 3 using 2.)

We now define a relation \( \approx \) that captures the notion that two histories look equivalent from the perspective of actions of a transaction. For any non-empty history \( h \), let \( \text{Abort}(h) \) denote \( h \) with its last event’s return value replaced by \( \	ext{abort} \). Define \( \text{Abort}^r(h) \) to be the set \( \{ h \} \cup \{ \text{Abort}(h) \} \) if \( h \approx \) \( h') \), otherwise \( \text{Abort}^r(h) \) is a non-empty prefix of \( h \).

Let \( h_1 \approx h_2 \) be histories. We say that \( h_1 \approx h_2 \) if for any \( t \)-history \( h_1 \), we have \( h_1 h_2 \approx S \Rightarrow h_1 h_2 \approx S \) and \( h_1 h_2 \approx S \Rightarrow h_1 h_2 \approx S \). This relation says that the behavior of \( t \) after \( h_1 \) and \( h_2 \) are almost the same: the only asymmetry is that aborts may prevent some possible behaviors after \( h_1 \).

Definition 14 (Safe transaction). A transaction \( t \) in a history \( h \) is said to be a safe transaction (with respect to a set of histories \( S \)) if there exists some set \( T \) of dependable transactions in \( h \) such that \( h \approx (h[T], h[t]) \) where \( h[T] \) is any legal serialization of \( h[T] \). We say that \( t \) safely-depend on \( T \) (in \( h \)) in this case.

Definition 15 (Dependable). An abortable ws-protocol \((P, w, f)\) is said to be dependable if every visible transaction is dependable: i.e., for any \( h \in P \) and transaction \( t \) in \( h \), if \( f(h, t) \) is visible, then \( t \) is dependable.

Definition 16 (Read window). Let \((P, w, f)\) be an abortable ws-protocol. Let \( h = h_1 h_2 \) be a history in \( P \) containing an event \( e \). Event \( e \) is said to be a read-point for a transaction \( t \) if \( t \) safely-depends on \( V \) for any \( V \in \text{witness}(h_1, f) \). (See Definition 8.) A read-window \( w \) in \( h \) is said to be a read-window for \( t \) if every event \( e \) in \( w \) is a read-point for \( t \). A read-window mapping \( r \) is a function that maps every pair \((h, t)\) (where \( h \in P \) and \( t \) is a transaction in \( h \)) to a read-window \( w \) for \( t \) in \( h \).

Definition 17. We say that \((P, w, f, r)\) is an opaque ws-protocol if \((P, w, f)\) is a dependable abortable ws-protocol and \( r \) is a read-window mapping for \((P, w, f)\).

Example 11. Every transaction \( t \) in every history \( h \) in the OMP' abortable ws-protocol (defined in Example 7) has a read window mapping \( r \) defined by \( r(t) = [e_i, e_j] \) where:

- If \( \text{commit} \) was executed successfully by \( t \), \( e_i \) and \( e_j \) are both the commit method executed by \( t \);
- Otherwise, \( e_i \) is the begin-read transaction executed by \( t \) and \( e_j \) is the first read method executed by \( t \) or the last event in \( h \) if no read was made by \( t \).

For 2PL protocol the read window mapping and the serialization window mapping are identical.

6.2 The Opacity Theorem

We now formalize the claim that an opaque ws-protocol ensures that every transaction sees only a “consistent state”.

Given a protocol \( P \) for an object \( o \), a client \( C \) of \( P \) is (the semantics of) a program that executes multiple, concurrent, transactions against \( o \) and satisfies the protocol. A formal definition of a client appears in the technical report. Mathematically, \( C \) is a subset of \( P \) (which satisfies certain properties detailed in the technical report).

Recall that a non-interleaved history \( h \) is of the form \( h_1 h_2 \cdots h_k \) where each \( h_i \) represents the sub-history executed by a different transaction \( t_i \). Note that \( t_i \) may be incomplete in \( h \). We say that the non-interleaved history \( h \) above is a serial history if for every \( 1 \leq i \leq k \), transaction \( t_i \) is complete in \( h \). In other words, for every prefix \( h_p \) of a complete history \( h \) we have \( \text{RUN}(h_p) \leq 1 \).

The following theorem shows that if \( P \) is an opaque ws-protocol, then every transaction in an interleaved history of \( C \) behaves as it does in some serial history of \( C \). It is important to distinguish the actual serial histories that are possible with a given implementation (say, of an STM) from the set of serial histories permitted by the specification of the object \( o \). E.g., in a real implementation, a serial history will not abort, while an interleaved history may abort, potentially producing behaviors not seen in a real serial history. However, the specification of an STM object allows aborts even in a serial history, thus explaining the behavior seen in an interleaved history.

Theorem 8 (Opacity Theorem). Let \( C \) be a client of an opaque ws-protocol \( P \), such that every serial history in \( C \) is a prefix of a complete serial history in \( C \). For every history \( h \in C \) and every transaction \( t \) in \( h \), there exists a serial history \( h_s \in C \) such that \( h[t = h_s] \).

\footnote{first read operation with no preceding write operation to the read location}
Many desired results follow from the above theorem. Thus, if no serial history throws an exception, no interleaved history will throw an exception. If no serial history contains a non-terminating loop, then no interleaved history contains a non-terminating loop.

6.3 Composition of Opaque Protocols

We now extend the composition operator we defined for abortable ws-protocols to compose opaque ws-protocols, by essentially adding the extra requirement that only histories that have overlapping read-windows are permitted.

Let \((P_1, w_1, f_1, r_1)\) and \((P_2, w_2, f_2, r_2)\) be opaque ws-protocols for two disjoint objects \(o_1\) and \(o_2\). Let \((P', w, f) = (P_1, w_1, f_1) \odot (P_2, w_2, f_2)\). Let \(r_2\) denote the natural extension of function \(r_1\) to histories in \(P'\) (where \(r_2(h, t)\) is determined by ignoring operations on \(o_2\) and is undefined if \(t\) does not access \(o_1\)). \(r_2\) is similarly defined. We say that a history \(h\) in \(P'\) has overlapping read-windows if for every transaction \(t\) in \(h\) that accesses both \(o_1\) and \(o_2\), the windows \(r_1(h, t)\) and \(r_2(h, t)\) overlap. Let \(P\) denote the set of histories \(h\) in \(P'\) that have overlapping read-windows. We define a read-window mapping function \(r\) as follows: \(r(h, t) = r_2(h, t)\) if both \(r_1(h, t)\) and \(r_2(h, t)\) are defined, and \(r(h, t) = r_1(h, t)\) if \(r_1(h, t) = o\) (for \(i = 1, 2\)).

We define the composed protocol \((P_1, w_1, f_1, r_1) \odot (P_2, w_2, f_2, r_2)\) to be \((P, w, f, r)\).

Theorem 9. If \(Z_1 = (Q_1, w_1, f_1, r_1)\) and \(Z_2 = (Q_2, w_2, f_2, r_2)\) are opaque ws-protocols (for disjoint objects), then \(Z_1 \odot Z_2\) is an opaque ws-protocol.

7. Evaluation

We have implemented and evaluated several instances of hybrid transactions that combine STM implementations and locking. Our implementations make use of DEUCE [8, 23] and ScalaSTM [4]. The API that both STMs expose is the ability to execute an arbitrary function atomically.

We now describe how we combine locking with the execution of the atomic function (by the STM) in a way that satisfies the requirements of our composition theory:

- Overlapping window: The serialization window of the atomic function is effectively the end of the atomic function (assuming no read-only transactions). We can ensure overlapping windows by acquiring the locks within the atomic function and releasing them after the STM atomic function returns.

- Compatible status: No writes to the object guarded by the 2PL are permitted within the atomic function.

- Aborts: We implemented a trivial abort function for 2PL (to release the 2PL locks) to be called by the STM abort function. While ScalaSTM allows such an extension to its abort procedure using the afterRollback API, with DEUCE we had to add such a functionality ourselves.

- Deadlock avoidance: We use Java tryLock to implement 2PL (as an universally unblocked protocol) and if acquiring a lock fails, we initiate STM abort (and release all locks acquired so far).

- Opacity: No reads from the object guarded by 2PL are permitted within the atomic function.

We have implemented various versions of the Intruder and Vacation benchmarks from Jstamp and a composite data-structure that combines in-memory data-structure and a disk-based log. In the technical report we discuss the performance benefits which achieved by our composition implementation comparing to traditional synchronization synchronization protocols. As described in Section 1, these results show how hybrid protocols can yield better performance than pure locking or STM approaches.

8. Related Work

Quite a few authors have previously explored opportunities for combining optimistic and pessimistic concurrency control [18, 24]. A distinguishing aspect of our work is that we present a general theory of correctness for composing abstract protocols. Much of the previous work [18, 22, 24] has focused on composition of specific protocols in specific systems. Our composition theorem can be used to justify the correctness of many protocol combinations (e.g., Lock-free STMs with 2PL, tree-locking with 2PL, etc.)

Combining optimistic and pessimistic concurrency control has been previously explored by Herlihy in the context of distributed databases [18]. Herlihy’s setting assumes the presence of a transaction manager that assigns logical timestamps to transactions as well as a common validation protocol that has some knowledge of all of the different concurrency control mechanisms. Within this framework it is provable that any concurrent transactions made up of mixed optimistic and pessimistic mechanisms is serializable. In [24] the STM must be aware of the locks and have a specific validation procedure for verifying that the optimistic operations do not violate the serializability of the lock operations. In contrast to the above, we present our results in a more abstract setting.

The work in [12, 14] is also motivated by the same abstract problem of composing transactions that use different protocols. Our work focuses on abstract properties that enable such composition, while [12, 14] address composition of a specific set of protocols. [12] also addresses compiler support for automating composition, which we do not address.

[22] presents another instance of a specific composition that combines lock-based STMs with lock inference. An alternative approach for composition is proposed in [5, 9, 13], which serializes different transactions simultaneously accessing the same shared memory using different synchronization protocols. A methodology for composing lock-free objects is described in [7]. Their approach is applicable to CAS-based algorithms, and relies on implementation details of the composed objects.

Adaptive runtime techniques for choosing between optimistic and pessimistic synchronization protocols for every transaction are described by [28, 30].

I/O operations are handled in a limited way in STMs. In [16], I/O is handled using a buffering mechanism which wraps the native libraries. As noted in [16], this solution is very limited and cannot support transactions dependent on the output of this operation. Finally, [3], describes an STM with two modes of operation: restricted (which limits the operations that can be done) and unrestricted (which allows any operation). The STM transition between the modes is done automatically, moving to unrestricted only when an operation violates the restricted mode. Only one running transaction at a time is allowed in an unrestricted mode (as in critical sections), which limits the degree of parallelism. Our method can exploit more parallelism opportunities by using an STM on the parts of a transaction that do not invoke I/O combined with a pessimistic (non-abortable) protocol. This ensures that transactions which include I/O execute atomically.

Linearizability [21] relates to atomicity of a single operation, while our thesis focuses on the atomicity of a sequence of operations (transactions), with a single-operation being a special case. Our formalism is based on abstract specifications of objects, as in linearizability (with a set of histories being used to model a specification). The protocols and the compositions we consider also preserve the real-time ordering of non-overlapping transactions, even though we do not explicitly discuss this. This is a simple consequence when the serializability window of every transaction is contained within the transaction.
Acknowledgments

The research leading to these results has received funding from the European Research Council under the European Union’s Seventh Framework Program (FP7/2007-2013) / ERC grant agreement n [321174-VSSC].

References


