The Push/Pull model of transactions

Eric Koskinen[1] and Matthew Parkinson[2]

[1]New York University
[2]Microsoft Research

Abstract

We present a general theory of serializability, unifying a wide range of transactional algorithms, including some that are yet to come. To this end, we provide a compact semantics in which concurrent transactions *push* their effects into the shared view (or *unpush* to recall effects) and *pull* the effects of potentially uncommitted concurrent transactions into their local view (or *unpull* to detangle). Each operation comes with simple side-conditions given in terms of commutativity (Lipton's left-movers and right-movers [24]).

The benefit of this model is that most of the elaborate reasoning (coinduction, simulation, subtle invariants, etc.) necessary for proving the serializability of a transactional algorithm is already proved within the semantic model. Thus, proving serializability (or opacity) amounts simply to mapping the algorithm on to our rules, and showing that it satisfies the rules' side-conditions.

1 Introduction

Recent years have seen an explosion of research on methods of providing `atomic` sections in modern programming languages, typically implemented via transactional memory (TM). The `atomic` keyword provides programmers with a powerful concurrent programming building block: the ability to specify when a thread's operations on shared memory should appear to take place instantly when viewed by another thread.

To support such a construct, we must be able to reason about atomicity. Implementations typically achieve this by dynamically detecting conflicts between concurrent threads. This can be done tracking memory operations in hardware [15, 17, 16] or software [14, 6, 8, 25, 4]. Meanwhile, an alternate approach exploits abstract-level notions of conflict over linearizable data-structure operations such as commutativity [11, 28, 21, 20]. Both levels of abstraction also chose between optimistic execution, pessimistic execution, or mixtures of the two. Finally, there are multiple notions of correctness, and circumstances under which one may be preferable to another.

Unfortunately, we lack a unified way of formally describing this myriad of models, implementations and correctness criteria. This leads to confusion when trying to understand comparative advantages/disadvantages and how/when models can be combined or are interoperable. For example, with hardware support for transactions now available (*e.g.* Intel Haswell [17]), we need to understand how one can combine *memory-level* hardware transactions for unstructured memory operations with *abstract-level* data-structure operations (*e.g.* transactional boosting [11]). Today, at best, we have two custom semantics for reasoning about the models individually, but no unified view.

We present a simple calculus that illuminates the core of transactional memory systems. In our model concurrent transactions *push* their effects into the shared log (or *unpush* to roll-back) and *pull* in the effects of potentially uncommitted concurrent transactions (or *unpull* to detangle). Moreover, transactions can push or pull operations in non-chronological orders, provided certain commutativity (Lipton left/right-movers [24]) conditions hold. The benefit of this semantic model is that most of the elaborate reasoning (coinduction, simulation relations, subtle invariants, etc.) necessary for proving the correctness of a transactional algorithm is contained within the semantic model, and need only be proved once.

Our work formulates an expressive class of transactions and we have applied it to a wide range of TM systems including: optimistic read/write software TMs [6, 8], hardware transactional memories (Intel [17], IBM [16]), pessimistic TMs [25, 4, 11], hybrid optimistic/pessimistic TMs such as irrevocability [34], open nested transactions [28], and abstract-level techniques such as boosting [11].

Our choice of expressiveness includes transactions that are not opaque [10]: transactions *may* share their uncommitted effects. This choice carves out a design space for implementations to take advantage of the full spectrum of possibilities (*e.g.* dependent transactions [30]) and is relatively unrestrictive in terms of TM correctness criteria. However, despite expressive power, the model also gives the appropriate criteria to ensure serializability [29]. Meanwhile, we can also identify restrictions on the model for which opacity is recovered.

In our experience we have found that our model provides a mathematically rigorous foundation for intuitive concepts (*e.g. push* and *pull*) used in colloquial conversations contrasting TM systems.

Contributions. Our work includes the following:

- A general model of concurrent transactions in Section 4, capable of expressing a wide range of implementations, with only a few intuitive rules. Our model is parameterized by a coinductive sequential specification and the operational semantics of the programming language.
- We have proved that this model is serializable, discussed in Section 5. To cope with the non-monotonic nature of the model (arising from *unpush*, *unpull*, etc.), we devised a novel preservation invariant that is closed under rewinding both the local and global logs. The serializability proof shows simulation with an uninterleaved machine.
- We have shown conditions under which we can restrict the model to obtain a sub-model that satisfies opacity.
- In Section 6 we describe how our model accounts for the serializability of many transactional memory systems that range from software to hardware, pessimistic to optimistic, read/write-conflict to abstract-conflict, nesting and non-opaque features such as dependent transactions.

Limitations. The work presented in this paper models safety properties of transactions (*i.e.* serializability, opacity). A direction for future work is to consider liveness/progress issues.

Related work. In previous work [20] we provided a formal semantics for transactions that perform abstract-level data-structure operations. Our work involves two *separate* semantics: one for pessimistic transactions and one for optimistic transactions. There are several distinctions of our work: (i) The model presented in this paper is more expressive because it permits mixtures of these two flavors. This is particularly useful when combining hardware transactions [17, 16] with and abstract-level reasoning [11] for data-structures. (ii) In our mode, transactions may observe the effects of uncommitted, non-commutative transactions as seen in dependent transactions [30] and open nesting [28]. (iii) We have a simulation result which involves nontrivial formal groundwork such as a coinductive definition of state equality and left-mover.

Lesani *et al.* [22] describe a method of specifying and verifying TM algorithms. They specify some transactional algorithms in terms of I/O automata and this choice of language enables them to fully verify those specifications in PVS. In our work, we have aimed at a more abstract goal: to uncover the fundamental nature of transactions in the form of a general-purpose model. We leave the goal of full algorithm verification to future work.

There are other works in the literature that are focused on a variety of orthogonal semantic issues, including the privatization problem [32, 26, 1], correctness criteria such as dynamic/static/hybrid atomicity [33], and message passing within transactions [23]. These works are concerned with models that are restricted to read/write STMs and limited in expressive power (e.g. restricted to opacity [10]). Semantics also exist for other programming models that are similar to transactions [2, 3] but are not serializable. Finally, Cohen *et al.* [5] described some small hand proofs for particular transactional memory algorithms.

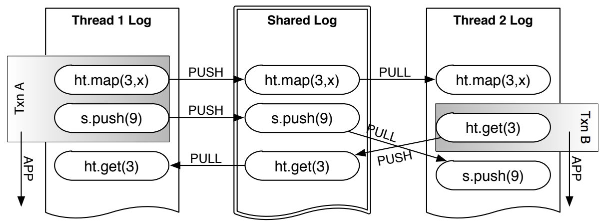

Figure 1: A diagram of Push/Pull

2 Overview

In this paper we distill the essence of reasoning about transactional implementations into a semantic model we call Push/Pull transactions. The model consists of a few simple rules—named push, pull, etc.—that correspond to natural stages in a transactional memory algorithm. For example, after a transaction applies an effect locally it then may push this effect out into the shared view, where other transactions may pull the effect into their local view.

The Push/Pull model has no concrete state, only a shared log of the object operations that have been applied, as well as per-thread local logs. An illustration is given in Figure 1. The full formal detail of the model is given in Section 4. We will now discuss this model informally.

Once a transaction (logically) applies an operation in its local log via the app rule, it may push the operation to the shared log. Note that, at this stage, the transaction may not have committed. Meanwhile, other threads may pull the operation into their local log. The pull case enables transactions to update their local view with operations that are permanent (that is, that correspond to committed transactions) or even to view the effects of another uncommitted transaction (*e.g.* for early conflict detection [14] or to establish a dependency [30]). Push/Pull also includes an unpull rule which discards a transaction's knowledge of an effect due to another thread, and an unpush rule which removes a thread's operation from the shared view, perhaps implemented as an inverse. The unAPPly rule is useful for rewinding a transaction's local state. Finally, there is a simple commit rule cmt that, roughly, stipulates that all operations must have been pushed and all pulled operations must have been committed.

Different algorithms will use different combinations of these rules (*cf.* Section 6). Push/Pull is expressive enough to describe a wide range of transactional implementations, all with only a few simple, tangible rules. Pessimistic algorithms [11, 25, 4] push immediately after a local app, optimistic algorithms [6, 8] push their operations on commit, and hybrid [34] algorithms do a mixture of the two. Opaque [10] transactions do not pull uncommitted effects. Non-opaque algorithms, such as dependent transactions [30], permit a transaction to pull in uncommitted effects. From different patterns of Push/Pull rule usage one can derive correctness proofs for many transactional memory algorithms.

Example. Consider the transactional boosting [11, 12] hashtable implementation given in Figure 2. Recall that a boosted transaction uses a linearizable base object (in this case a `ConcurrentSkipListMap`), along with abstract locking to ensure that only commutative operations occur concurrently. In this example a thread executing the `atomic` block in `put` or `get` acquires a lock corresponding to the `key` of interest. In this way, no two transactions will conflict because if they try to access the same `key` one will block. Within the `put` method there are two scenarios depending on whether `key` is already defined in the `map` and, consequently, there are two cases for how to handle an abort. Finally, `put` ends by updating `map` and unlocking the `abstractLock`.

We can describe this algorithm intuitively, in terms of rules in the Push/Pull model. We have decomposed the code accordingly in Figure 2. After the transaction begins, it implements a pull implicitly because, in transactional boosting, modifications are made directly to the shared state so the local view is the same as the shared view. Skipping the abort cases for the moment, the transaction then performs an

```
import java.util.concurrent.ConcurrentSkipListMap
class BoostedSkipListMap[Key,Value] {
  val abstractLock = new AbstractLock()
  val map = new ConcurrentSkipListMap[Key, Value]()

  def put(key: Key, value: Value, t = Tx.current) {
   atomic {
     abstractLock lock key

      if (map contains key) {
        var oldValue = map(key)
        Tx.onAbort( () =>
           map.put(key, oldValue)
           abstractLock unlock key
        )
      } else {
        Tx.onAbort( () =>
           map.remove(key)
           abstractLock unlock key
        )
      }
      map.put(key,value)
      abstractLock unlock key
    }
  }

  def get(key: Key) {
    atomic {
      abstractLock lock key
      Tx.onExit(() => abstractLock.unlock(key) )
      key k = map.get(key)
      return k
    }
  }
}
```

```
import java.util.concurrent.ConcurrentSkipListMap
class BoostedSkipListMap[Key,Value] {
  val abstractLock = new AbstractLock()
  val map = new ConcurrentSkipListMap[Key, Value]()

  def put(key: Key, value: Value, t = Tx.current) {
     atomic {
          BEGIN
          abstractLock lock key;

          PULL(map)
          (implicit because shared access to map)

          if (map contains key) {
             var oldValue = map(key)
             Transaction.onAbort( () =>
                 UNPUSH(map.put(key,value)) and
                 UNAPP(map.put(key,value))
                 map.put(key, oldValue)
                 abstractLock unlock key

             )

          } else {
             Transaction.onAbort( () =>
                 UNPUSH(map.put(key,value)) and
                 UNAPP(map.put(key,value))
                 map.remove(key)
                 abstractLock unlock key
               )
          }
          APPLY(map.put(key,value)) and
          PUSH(map.put(key,value))
          map.put(key, value)

          CMT
          abstractLock unlock key

     }
  }

  def get(key:Key) { . . . }
}
```

Figure 2: On the left, an implementation of *transactional boosting* [11] which uses abstract locking and commutativity to safely access a shared Set, implemented as a `ConcurrentSkipList`. On the right, decomposition of the boosting implementation into Push/Pull rules such as app, push, pull, cmt, as well as unapp, unpush used for aborting a transaction. Each of these rules comes with a correctness criteria (see Figure 5) which, if proved to hold, implies that the implementation is serializable.

APPly (effecting the local view) and a push (sending the effect to the shared view) by updating the `map`. If the transaction aborts, it performs the opposite of these two Push/Pull rules in reverse: unpush and then unAPPly by performing the appropriate inverse operation. The abstract locking of boosting ensures that the only operations in the shared log belong to committed transactions or else commute with all other uncommitted operations in the shared log.

Proofs of serializability. Each rule in Push/Pull comes with a few *correctness criteria*. In Section 5 we prove that if an implementation satisfies these criteria, then it is serializable. In this sense we have done the hard work of reasoning about transactional memory algorithms. The Push/Pull model encapsulates the difficult components involved in a correctness argument (*e.g.* simulation proofs, coinduction, etc.) while, on the outside, offering rules that are simple and intuitive. For a user to prove the correctness of their algorithm they must simply:

1. Demarcate the algorithm into fragments: PUSH, PULL, etc.
2. Prove the implementation satisfies the respective correctness criteria.

Proofs of correctness criteria typically do not involve elaborate simulation relations or coinductive reasoning, but rather algebraic (*i.e.* commutative) properties of sequential code.

Synchronization in Figure 2 is accomplished using abstract locks to ensure that only commutative operations proceed in parallel. One correctness criterion of PUSH(`map.put(key,value)`) is that the `put` operation must be able to commute with (more precisely: move to the right of) concurrent, uncommitted operations. We can show that this correctness criterion holds by showing that the two sequential sequences:

```
map.put(key1,value1); map.put(key2,value2)
```

vs.

```
map.put(key2,value2); map.put(key1,value1)
```

lead to the same final state, provided that `key1` ≠ `key2`. Moreover, such proofs involving commutativity can been aided by recent works in the literature [7, 18]. Note that, without PUSH/PULL, the full formal argument would say that the data structure is atomic because there is a simulation relation between any configuration of concurrent transactions and a sequential history. The benefit of the PUSH/PULL semantic model is that the simulation relation has been identified and the difficult aspects of the correctness argument have already been proved. We believe that our work will cleanup transactional correctness proofs because it suffices to show that the implementation satisfies the correctness criteria given in our proof rules.

3 Language and Atomic Semantics

In this section we describe a generic language of transactions and define an idealized semantics for concurrent transactions called the atomic semantics in which there are no interleaved effects on the shared state. We later introduce the PUSH/PULL semantics and show that it simulates the atomic semantics.

Language. We assume a set M of method calls (*e.g.* `ht.put('a',5)`). Threads execute code from a programming language that includes transactions `tx` c, method names such as m, and a `skip` statement. Our first trick is to abstract away the threads' programming language c with two functions:

step(c)**:** Pair $(m, c') \in \mathsf{step}(c)$ if m is a next reachable method in the reduction of c, with remaining code c'.

fin(c)**:** This predicate is true if there is a reduction of c to `skip` that does not encounter a method call.

These two functions allow us to obtain a simple semantics, despite an expressive input language, by introducing functions to resolve nondeterminism between method operation names and at the end of a transaction. We assume that code is well-formed in that a single operation name m is always contained within a transaction (orthogonal, is this issue of isolation [26]).

Example 1. *One could use the generic language:*

$$c \ ::= \ c_1 \texttt{ + } c_2 \mid c_1 \texttt{ ; } c_2 \mid (c)^* \mid \texttt{skip} \mid \texttt{tx}\, c \mid m$$

This grammar additionally consists of nondeterministic choice, sequential composition, and nondeterministic *looping. The corresponding functions for this example are:*

$$
\begin{array}{lcl}
\mathit{step}(\texttt{skip}) & \equiv & \varnothing \\
\mathit{step}(c_1 \ \texttt{;}\ c_2) & \equiv & (\mathit{step}(c_1)\ \texttt{;}\ c_2) \cup (\mathit{fin}(c_1)\ \texttt{;}\ \mathit{step}(c_2)) \\
\mathit{step}(c_1 \ \texttt{+}\ c_2) & \equiv & \mathit{step}(c_1) \cup \mathit{step}(c_2) \\
\mathit{step}((c)^*) & \equiv & \mathit{step}(c)\ \texttt{;}\ (c)^* \\
\mathit{step}(\texttt{tx}\ c) & \equiv & \mathit{step}(c) \\
\mathit{step}(m) & \equiv & \{(m, \texttt{skip})\} \\
\mathit{fin}(\texttt{skip}) & \equiv & \mathit{true} \\
\mathit{fin}(c_1 \ \texttt{;}\ c_2) & \equiv & \mathit{fin}(c_1) \wedge \mathit{fin}(c_2) \\
\mathit{fin}(c_1 \ \texttt{+}\ c_2) & \equiv & \mathit{fin}(c_1) \vee \mathit{fin}(c_2) \\
\mathit{fin}((c)^*) & \equiv & \mathit{true} \\
\mathit{fin}(\texttt{tx}\ c) & \equiv & \mathit{fin}(c) \\
\mathit{fin}(m) & \equiv & \mathit{false}
\end{array}
$$

$$
\begin{array}{rcl}
S\ \texttt{;}\ c_1 & \equiv & \{(m, c_1; c_2) \mid (m, c_1) \in S\} \\
B\ \texttt{;}\ S & \equiv & \{(m, c_1) \mid B \wedge (m, c_1) \in S\}
\end{array}
$$

Thus, if $c = \texttt{tx}\ (\texttt{skip}\ \texttt{;}\ (c_1\ \texttt{+}\ (m\ \texttt{+}\ n))\ \texttt{;}\ c_2)$*, then one path through* c *reaches method* n *with a continuation of* c_2*. Hence,* $(n, c_2) \in \mathit{step}(c)$*.*

To make things more concrete, we instantiate our semantics with the above language through the remainder of this paper. We ignore nested transactions[1], however our model permits threads to roll backwards to any execution point [19] (thus modeling the partial abort nature of nested transactions).

Operations and logs. State is represented in terms of logs of operation records. An operation record (or, simply, an "operation") $op = \langle m, \sigma_1, \sigma_2, id\rangle$ is a tuple consisting of the operation name m, a thread-local pre-stack σ_1 (method arguments), a thread-local post-stack σ_2 (method return values), and a unique identifier id. We assume a predicate $\mathsf{fresh}(id)$ that holds provided that id is globally unique (details omitted for lack of space). In the atomic semantics defined below, the shared state $\ell : \mathsf{list}\ op$ is an ordered list of operations (more information is needed in the PUSH/PULL semantics, discussed later).

Parameter 3.1 (Sequential specification: $\mathsf{allowed}$)**.** *The sequential specification is a predicate on operation lists:* $\mathsf{allowed}\ \ell$*. We require that it be prefix closed.*

For convenience we will also write ℓ allows $\langle m, \sigma_1, \sigma_2, id\rangle$ which simply means $\mathsf{allowed}\ \ell \cdot \langle m, \sigma_1, \sigma_2, id\rangle$. For example, if we have a simple TM based on memory read/write operations we might specify $\mathsf{allowed}\ \ell \cdot \langle \texttt{a := x}, [x \mapsto 5], [x \mapsto 5, a \mapsto 5], id\rangle$, but $\neg\mathsf{allowed}\ \ell \cdot \langle \texttt{a := x}, [x \mapsto 5], [x \mapsto 5, a \mapsto 3], id\rangle$ or more elaborate specifications that involve multiple tasks.

Ultimately, we expect the $\mathsf{allowed}$ predicate to be induced by the implementation's operations on the state, $[\![op]\!] : \mathcal{P}(\mathsf{State} \times \mathsf{State})$, and the initial states, I. If we give a denotation to logs as $[\![\ell \cdot op]\!] \equiv [\![\ell]\!]; [\![op]\!]$, and $[\![\epsilon]\!] \equiv I$, where $S; R \equiv \{s' \mid \exists s \in S.(s, s') \in R\}$. Then we can define $\mathsf{allowed}\ \ell$ simply by checking if the denotation is non-empty, ($[\![\ell]\!] \neq \varnothing$).

We define a precongruence over operation logs $\ell_1 \preccurlyeq \ell_2$ coinductively, by requiring that all $\mathsf{allowed}$ extensions of the log ℓ_1, are also $\mathsf{allowed}$ extension to the log ℓ_2. This definition will ultimately be used in the simulation between PUSH/PULL and an atomic machine. We use a coinductive definition so that the precongruence can be defined up to all infinite suffixes.

Definition 3.1 (Shared log precongruence $\preccurlyeq$)**.** *For all* ℓ_1, ℓ_2*,*

$$
\frac{\mathit{allowed}\ \ell_1 \Rightarrow \mathit{allowed}\ \ell_2 \quad \forall op.\ (\ell_1 \cdot op) \preccurlyeq (\ell_2 \cdot op)}{\ell_1 \preccurlyeq \ell_2}
$$

We use a double-line here to indicate greatest fixpoint.

[1] For a discussion, see [19].

$$\boxed{(c,\sigma_1),\ell_1 \Downarrow \sigma_2,\ell_2}$$

$$\frac{\begin{array}{l}(m_1,c_2) \in \mathsf{step}(c_1)\\ \ell_1 \text{ allows } \langle m_1,\sigma_1,\sigma_2\rangle\\ (c_2,\sigma_2),\ell_1 \cdot [\langle m_1,\sigma_1,\sigma_2\rangle] \Downarrow \sigma_3,\ell_2\end{array}}{(c_1,\sigma_1),\ell_1 \Downarrow \sigma_3,\ell_2}\ \textsc{bsstep} \qquad \frac{\mathsf{fin}(c)}{(c,\sigma_1),\ell_1 \Downarrow \sigma_1,\ell_1}\ \textsc{bsfin}$$

$$\boxed{c_1,\sigma_1,\ell \xrightarrow{a} c_2,\sigma_2,\ell}$$

$$\frac{(c,\sigma_1),\ell_1 \Downarrow \sigma_2,\ell_2}{\mathtt{tx}\ c,\sigma_1,\ell_1 \xrightarrow{a} \mathtt{skip},\sigma_2,\ell_2}\ \textsc{am\_runtx}$$

$$\frac{}{c_1\ \mathtt{+}\ c_2,\sigma_1,\ell \xrightarrow{a} c_1,\sigma_1,\ell}\ \textsc{am\_nondet}\mathrm{L} \qquad \frac{}{c_1\ \mathtt{+}\ c_2,\sigma_1,\ell \xrightarrow{a} c_2,\sigma_1,\ell}\ \textsc{am\_nondet}\mathrm{R}$$

$$\frac{}{((c))^*,\sigma_1,\ell \xrightarrow{a} (c\ \ \mathtt{;}\ \ ((c)^*))\ \mathtt{+}\ \mathtt{skip},\sigma_1,\ell}\ \textsc{am\_loop}$$

$$\frac{c_1,\sigma_1,\ell_1 \xrightarrow{a} c_1',\sigma_2,\ell_2}{c_1\ \ \mathtt{;}\ \ c_2,\sigma_1,\ell_1 \xrightarrow{a} c_1'\ \ \mathtt{;}\ \ c_2,\sigma_2,\ell_2}\ \textsc{am\_semi} \qquad \frac{}{\mathtt{skip}\ \ \mathtt{;}\ \ c_1,\sigma_1,\ell_1 \xrightarrow{a} c_1,\sigma_1,\ell_1}\ \textsc{am\_semiskip}$$

$$\boxed{\mathbf{A}_1,\ell_1 \xrightarrow{a}{}^{*} \mathbf{A}_2,\ell_2}$$

$$\frac{}{\mathbf{A}_1,\ell_1 \xrightarrow{a}{}^{*} \mathbf{A}_1,\ell_1}\ \textsc{ams\_refl} \qquad \frac{c_1,\sigma_1,\ell_1 \xrightarrow{a} c_2,\sigma_2,\ell_2}{\mathbf{A}_1 \cdot (\sigma_1,c_1) \cdot \mathbf{A}_2,\ell_1 \xrightarrow{a}{}^{*} \mathbf{A}_1 \cdot (\sigma_2,c_2) \cdot \mathbf{A}_2,\ell_2}\ \textsc{ams\_one}$$

$$\frac{}{\mathbf{A}_1 \cdot (\sigma_1,\mathtt{skip}) \cdot \mathbf{A}_2,\ell_1 \xrightarrow{a}{}^{*} \mathbf{A}_1 \cdot \mathbf{A}_2,\ell_1}\ \textsc{ams\_end} \qquad \frac{\begin{array}{l}\mathbf{A}_1,\ell_1 \xrightarrow{a}{}^{*} \mathbf{A}_2,\ell_2\\ \mathbf{A}_2,\ell_2 \xrightarrow{a}{}^{*} \mathbf{A}_3,\ell_3\end{array}}{\mathbf{A}_1,\ell_1 \xrightarrow{a}{}^{*} \mathbf{A}_3,\ell_3}\ \textsc{ams\_trans}$$

Figure 3: Atomic semantics of concurrent threads.

Informally, the above definition says that there is no sequence of observations we can make of ℓ_2, that we can't also make of ℓ_1. This is more general than just considering the set of states reached from executing the first log is included in the second: unobservable state differences are also permitted.

Atomic semantics. We define a simple atomic semantics, given in Figure 3 in which transactions are executed instantly, without interruption from concurrent threads. The semantics is a relation $\xrightarrow{a}{}^{*}$ over pairs consisting of a list of concurrent threads $\mathbf{A}$ and a shared state ℓ. A single thread $(\sigma, c) \in \mathbf{A}$ is a local stack and code c. The relation $\xrightarrow{a}{}^{*}$ is reflexive, transitive, and permits a thread to complete (rules AMS_REFL, AMS_TRANS, AMS_END, respectively).

According to the rule AMS_ONE, a single thread can be reduced using the $\xrightarrow{a}$ relation which is defined inductively over the structure of c. The rules for `+` , `;` , c^* and `skip` AM_NONDETL, AM_NONDETR, AM_LOOP, AM_SEMI, and AM_SEMISKIP are standard. The rule AM_RUNTX atomically executes the entire transaction `tx` c via the $\Downarrow$ reduction (*i.e.* big step semantics). The big step semantics $\Downarrow$ uses $\mathsf{step}()$ and $\mathsf{fin}()$ (rules BSSTEP and BSFIN, respectively) to scan through the nondeterminism in `tx` c to find a next operation name m or a path to `skip` denoting the end of the transaction. BSSTEP can be taken provided that the operation *op* is permitted by the sequential specification and that c_2 can be entirely reduced.

4 The Push/Pull Model

In this section we describe Push/Pull, an expressive model of serializable transactions. Concurrent threads execute the language described in the previous section but now transaction interleavings are possible. Moreover, we describe reductions APP, UNAPP, PUSH, UNPUSH, PULL, UNPULL, CMT, which can be made by a

$$\frac{\mathbf{T}_1, G_1 \rightarrow^* \mathbf{T}_2, G_2 \qquad \mathbf{T}_2, G_2 \rightarrow^* \mathbf{T}_3, G_3}{\mathbf{T}_1, G_1 \rightarrow^* \mathbf{T}_3, G_3}\ \text{MS\_TRANS} \qquad \frac{}{\mathbf{T}_1, G_1 \rightarrow^* \mathbf{T}_1, G_1}\ \text{MS\_REFL} \qquad \frac{\mathbf{T}_1, G_1 \xrightarrow{\mathsf{rt}} \mathbf{T}_2, G_2}{\mathbf{T}_1, G_1 \rightarrow^* \mathbf{T}_2, G_2}\ \text{MS\_ONE}$$

$$\frac{T_1, G_1 \xrightarrow{\mathsf{rt}} T_2, G_2}{\mathbf{T}_1 \cdot T_1 \cdot \mathbf{T}_2, G_1 \xrightarrow{\mathsf{rt}} \mathbf{T}_1 \cdot T_2 \cdot \mathbf{T}_2, G_2}\ \text{MS\_SELECT} \qquad \frac{}{\mathbf{T}_1 \cdot \{\mathtt{skip}, \sigma_1, L_1\} \cdot \mathbf{T}_2, G_1 \rightarrow^* \mathbf{T}_1 \cdot \mathbf{T}_2, G_1}\ \text{MS\_END}$$

Figure 4: The machine reductions of PUSH/PULL. See Figure 5 for individual steps.

given transaction to control how its effects are shared with the environment or view the effects made by the environment.

As in the atomic semantics, the PUSH/PULL semantics has a reflexive, transitive reduction $\mathbf{T}, G \rightarrow^* \mathbf{T}', G'$ that reduces a list of threads $\mathbf{T} : \mathsf{list}\ (c \times \sigma \times L)$ and a global log G to $\mathbf{T}', G'$. L and G are local and global operation logs, respectively, described in more detail below.

The reductions of the form $\mathbf{T}, G \rightarrow^* \mathbf{T}', G'$ are given in Figure 4. As in the atomic machine, $\rightarrow^*$ is transitive (MS_TRANS). and reflexive (MS_RELF). The MS_END rule removes a completed thread (*i.e.* a thread that has reached `skip`) from the list of threads. The MS_SELECT rule reduces a single thread via the $\rightarrow$ relation. Finally, the MS_ONE rule incorporates this reduction into the $\rightarrow^*$ relation.

The single-thread reduction relation $\xrightarrow{\mathsf{rt}}$ is defined inductively over c and has three types:

$$\xrightarrow{\mathsf{rt}} \ ::= \ \xrightarrow{\mathsf{struct}} \mid \xrightarrow{\mathsf{fwd}} \mid \xrightarrow{\mathsf{back}}$$

The structural $\xrightarrow{\mathsf{struct}}$ reductions depend on the language. For the example language mentioned earlier, there are $\xrightarrow{\mathsf{struct}}$ rules for nondeterministic choice, nondeterministic looping and sequential composition as in Figure 6 (NONDETL, NONDETR, LOOP, SEMI, SEMISKIP). The four $\xrightarrow{\mathsf{fwd}}$ rules APP, CMT, PUSH, and PULL pertain to transactions making forward progress and the $\xrightarrow{\mathsf{back}}$ rules UNAPP, UNPUSH, UNPULL pertain to transactions rewinding.

Figure 5 lists the seven proof rules that form the core of PUSH/PULL. These rules pertain to a thread performing a transaction `tx` c and manipulate the local stack, local log, and shared log in various ways. The local log $L : \mathsf{list}\ (op \times l)$ is a list of operations, along with an additional flag l per operation, as to the status of the operation:

$$\begin{array}{rcll} l & ::= & \mathsf{npshd}\ c & \text{(local operation)} \\ & \mid & \mathsf{pshd}\ c & \text{(local operation shared to global view)} \\ & \mid & \mathsf{pld} & \text{(some other txn's operation)} \end{array}$$

The npshd and pshd flags save the code c that was active when the log entry was created. There is also a global log $G : \mathsf{list}\ (op \times g)$ with flag g that distinguishes between operations that have or have not been committed: $g\ ::=\ \mathsf{gUCmt} \mid \mathsf{gCmt}$. Each proof rule comes with criteria, labeled as APP criterion (i), APP criterion (ii), etc. Note that we lift $\in, \smallsetminus, \subseteq$ to logs as follows, using id for equality:

$$\begin{array}{rcl} \langle m_1, \sigma_1, \sigma_1', id_1 \rangle \in L & \equiv & \exists i.\ \mathsf{let}\ L[i] = [\langle m_2, \sigma_2, \sigma_2', id_2 \rangle, l]\ \mathsf{in}\ id_1 = id_2 \\ G \smallsetminus L & \equiv & \mathsf{filter}\ (\lambda\ (\langle m_1, \sigma_1, \sigma_1', id_1 \rangle, g).\ \langle m_1, \sigma_1, \sigma_1', id_1 \rangle \notin L)\ G \\ L \subseteq G & \equiv & \forall i.\ \mathsf{let}\ L[i] = [\langle m_1, \sigma_1, \sigma_1', id_1 \rangle, l]\ \mathsf{in}\ \langle m_1, \sigma_1, \sigma_1', id_1 \rangle \in G \end{array}$$

Here the notation $L[i]$ refers to the ith list element of L.

The APP rule. APP is similar to the BSSTEP rule in the atomic semantics: it applies if there is a nondeterministic path in code c_1 that reaches a method m_1 (with continuation code c_2). APP criterion (ii) specifies that method m_1 must be allowed by the sequential specification with post-stack σ_2. If so, the new operation

$\textsc{Push/Pull}$ **Rules**

$$\frac{\begin{array}{ll}(i)\text{–} & (m_1,c_2)\in \mathsf{step}(c_1)\\ (ii)\text{–} & L_1\ \mathsf{allows}\ \langle m_1,\sigma_1,\sigma_2\rangle\\ (iii)\text{–} & \mathsf{fresh}(id)\end{array}}{\{\texttt{tx}\ c_1,\sigma_1,L_1\},G_1 \xrightarrow{\mathsf{fwd}} \{\texttt{tx}\ c_2,\sigma_2,L_1\cdot[\langle m_1,\sigma_1,\sigma_2,id\rangle,\mathsf{npshd}\ \ c_1]\},G_1}\ \textsc{app}$$

$$\frac{}{\{\texttt{tx}\ c_1,\sigma_1,L_1\cdot[\langle m_1,\sigma_2,\sigma_3,id\rangle,\mathsf{npshd}\ \ c_2]\},G_1 \xrightarrow{\mathsf{back}} \{\texttt{tx}\ c_2,\sigma_2,L_1\},G_1}\ \textsc{unapp}$$

$$\frac{\begin{array}{ll}(i)\text{–} & op \blacktriangleleft \lfloor L_1\rfloor_{\mathsf{npshd}}\\ (ii)\text{–} & \lfloor G_1\rfloor_{\mathsf{gUCmt}} \setminus \lfloor L_1\cdot L_2\rfloor_{\mathsf{pshd}} \blacktriangleleft op\\ (iii)\text{–} & G_1\ \mathsf{allows}\ op\end{array}}{\{\texttt{tx}\ c_1,\sigma_1,L_1\cdot[op,\mathsf{npshd}\ \ c_2]\cdot L_2\},G_1 \xrightarrow{\mathsf{fwd}} \{\texttt{tx}\ c_1,\sigma_1,L_1\cdot[op,\mathsf{pshd}\ \ c_2]\cdot L_2\},G_1\cdot[op,\mathsf{gUCmt}]}\ \textsc{push}$$

$$\frac{\begin{array}{ll}(i)\text{–} & \mathsf{allowed}\ G_1\cdot G_2\\ (ii)\text{–} & \lfloor L_2\rfloor_{\mathsf{pshd}} \blacktriangleleft op\end{array}}{\{\texttt{tx}\ c_1,\sigma_1,L_1\cdot[op,\mathsf{pshd}\ \ c_2]\cdot L_2\},G_1\cdot[op,g]\cdot G_2 \xrightarrow{\mathsf{back}} \{\texttt{tx}\ c_1,\sigma_1,L_1\cdot[op,\mathsf{npshd}\ \ c_2]\cdot L_2\},G_1\cdot G_2}\ \textsc{unpush}$$

$$\frac{\begin{array}{ll}(i)\text{–} & op\notin L\\ (ii)\text{–} & L\ \mathsf{allows}\ op\\ (iii)\text{–} & op \blacktriangleleft \lfloor L\rfloor_{\mathsf{pshd}}\cup\lfloor L\rfloor_{\mathsf{npshd}}\end{array}}{\{\texttt{tx}\ c_1,\sigma_1,L\},G_1\cdot[op,g]\cdot G_2 \xrightarrow{\mathsf{fwd}} \{\texttt{tx}\ c_1,\sigma_1,L\cdot[op,\mathsf{pld}]\},G_1\cdot[op,g]\cdot G_2}\ \textsc{pull}$$

$$\frac{(i)\text{–}\quad \mathsf{allowed}\ L_1\cdot L_2}{\{\texttt{tx}\ c_1,\sigma_1,L_1\cdot[op,\mathsf{pld}]\cdot L_2\},G \xrightarrow{\mathsf{back}} \{\texttt{tx}\ c_1,\sigma_1,L_1\cdot L_2\},G}\ \textsc{unpull}$$

$$\frac{\begin{array}{ll}(i)\text{–} & \mathsf{fin}(c_1)\\ (ii)\text{–} & L_1\subseteq G_1\\ (iii)\text{–} & \lfloor L_1\rfloor_{\mathsf{pld}}\subseteq\lfloor G_1\rfloor_{\mathsf{gCmt}}\\ (iv)\text{–} & \mathsf{cmt}(G_1,L_1,G_2)\end{array}}{\{\texttt{tx}\ c_1,\sigma_1,L_1\},G_1 \xrightarrow{\mathsf{fwd}} \{\texttt{skip},\sigma_1,[]\},G_2}\ \textsc{cmt}$$

The $\textsc{cmt}$ rule involves predicate cmt, defined as follows:

$$\mathsf{cmt}(G_1,L_1,G_2)\ \equiv$$
$$G_2\ =\ \mathsf{map}\left(\lambda\ (op,g).\ \begin{cases}(op,\mathsf{gCmt}) & \text{if } op\in\lfloor L_1\rfloor_{\mathsf{pshd}}\\ (op,g) & \text{otherwise}\end{cases}\right)G_1$$

Figure 5: The $\textsc{Push/Pull}$ rules. Notations $\setminus,\notin,\cdot,\subseteq$ are all lifted to lists where equality is given by ids, as discussed below. We will refer to the premise criteria of each rule as, for example, "$\textsc{push}$ criterion (ii)." Standard rules for reducing nondeterminism in the input language are displayed in Figure 6. Criteria that are written in gray font are not strictly necessary. See inline discussion.

$$\frac{}{\{c_1\ \texttt{+}\ c_2,\sigma_1,L\},G \xrightarrow{\mathsf{struct}} \{c_1,\sigma_1,L\},G}\ \textsc{nondet}\mathrm{L} \qquad \frac{}{\{c_1\ \texttt{+}\ c_2,\sigma_1,L\},G \xrightarrow{\mathsf{struct}} \{c_2,\sigma_1,L\},G}\ \textsc{nondet}\mathrm{R}$$

$$\frac{}{\{((c))^*,\sigma_1,L\},G \xrightarrow{\mathsf{struct}} \{(c\ \ ;\ \ ((c)^*))\ \texttt{+}\ \texttt{skip},\sigma_1,L\},G}\ \textsc{loop}$$

$$\frac{\{c_1,\sigma_1,L_1\},G_1 \xrightarrow{\mathsf{rt}} \{c_1',\sigma_2,L_2\},G_2}{\{c_1\ \ ;\ \ c_2,\sigma_1,L_1\},G_1 \xrightarrow{\mathsf{rt}} \{c_1'\ \ ;\ \ c_2,\sigma_2,L_2\},G_2}\ \textsc{semi} \qquad \frac{}{\{\texttt{skip}\ \ ;\ \ c_1,\sigma_1,L_1\},G_1 \xrightarrow{\mathsf{struct}} \{c_1,\sigma_1,L_1\},G_1}\ \textsc{semiskip}$$

Figure 6: $\textsc{Push/Pull}$ transaction reductions for standard input language features. Notice that the type of the $\textsc{semi}$ reduction is inductive.

is appended to the local log L_1 with fresh operation id_1 (formalization of fresh in APP criterion (*iii*) is omitted). Intuitively, APP applies some next method m_1 locally but does not yet share it by sending it to the global log; it is marked as such with flag npshd. The APP rule also records the pre-code c_1 in the local log so that the transaction can later be reversed (*i.e.* aborted or undone). Indeed, the rule UNAPP moves backwards by taking the last item in the local log and, provided that it is still npshd, recalls the previous local stack and code.

The PUSH rule. A transaction may choose to share its effects with the global view via the PUSH rule. This reduction changes an operation's flag from npshd to pshd in the local log and appends the operation to the global log, provided three conditions hold. These conditions use the notion of *left-mover* which is an algebraic property of operations and due to Lipton [24]. We provide a novel coinductive definition of left-mover that builds upon log precongruence (*i.e.* a form of observational equivalence):

Definition 4.1 (Left-mover [24], over logs)**.** *For all* op_1, op_2

$$op_1 \blacktriangleleft op_2 \;\equiv\; \forall \ell.\ \ell \cdot \{op_1, op_2\} \preccurlyeq \ell \cdot \{op_2, op_1\}.$$

Intuitively, operation op_1 can move to the left of operation op_2 provided that whenever we are allowed to do $op_1 \cdot op_2$, we are also allowed to do $op_2 \cdot op_1$ and the resulting log is the same (precongruent). The proof of serializability involves several fairly straight-forward lemmas pertaining to allowed and left/right moverness, omitted for lack of space.

PUSH criterion (i) specifies that the pushed operation *op is able to move to the left of all unpushed operations in the local log.* This, intuitively, means that we can publish *op* as if it was the next thing to happen after all the operations published thus far by the current transaction. Formally, we lift $\blacktriangleleft$ to lists and define projections such as $\lfloor L_1 \rfloor_{\mathsf{npshd}}$ using:

$$\lfloor L_1 \rfloor_l \;\equiv\; \mathsf{map\ fst\ (filter}\ (\lambda(op, l').\ l = l')\ L_1)$$

and similar for $\lfloor G \rfloor_{\mathsf{gUCmt}}$.

> *Application example:* To our knowledge, all existing implementations satisfy this trivially because operations are PUSHed in the same order that they are APPlied.

PUSH criterion (ii) is that *all uncommitted operations in the shared log* $\lfloor G \rfloor_{gUCmt}$*—except those due to the current transaction–can move to the right of the current operation op.* This condition ensures that if the transaction commits at any point, it can serialize before all concurrent uncommitted transactions. (Recall that we have lifted $\blacktriangleright$ to lists where equality is given by the operation IDs and the order is determined by the first operand (in this case, G_1).)

> *Application examples:* A boosted transaction immediately performs a PUSH at the linearization point because it modifies the shared state in place. Optimistic STMs don't perform PUSH until commit-time (unless there is some early conflict detection [13] which involves a form of PUSH). In boosted [11] and open nested [28] transactions, a commutativity requirement is sufficient to ensure this condition.

PUSH criterion (iii) is that *op* is allowed by the sequential specification of the global log. (Here we have lifted allowed to global logs.)

The UNPUSH rule. An operation *op* that has been PUSHed to the shared log can be UNPUSHed. This amounts to swapping the local flag from pshd to npshd and removing the corresponding global log entry for *op*. PUSH criterion (i) ensures that G_2 does not depend on *op* and PUSH criterion (ii) is that everything pushed chronologically after *op* could still have been pushed if *op* hadn't been pushed. Note that PUSH criterion (i) is not strictly necessary because we can prove that it must hold whenever an UNPUSH occurs.

> *Application:* When a boosted transaction aborts (*e.g.* due to deadlock) it must undo its effects on the shared state. This is modeled via the UNPUSH rule and typically implemented via inverse operations (such as `remove` on an element that had been `add`ed).

The PULL **rule.** Transactions can learn about the published effects of other transactions by PULLing operations from the global log into their local logs. An operation *op* can be pulled from the global log provided that is wasn't pulled before (PULL criterion (*i*)) and that the local log allows it (PULL criterion (*ii*)) according to the sequential specification. A transaction can only learn about the shared state through PULLing. In most applications, a transaction will PULL operations in chronological order. However, there are many examples for which this is not true. In a transaction that operates over two shared data-structures a and b, it may PULL in the effects on a even if they occurred after the effects on b because the transaction is only interested in modifying a. When the PULL rule occurs, the operation is appended to the local log L and marked as pld.

Finally, PULL criterion (*iii*) is that *everything that the current transaction has currently done locally must be able to move to the right of op.* This ensures that the transaction can behave as if the pulled effect preceded the transaction. We have marked this criterion in gray, indicating that it is not strictly necessary. One could imagine allowing transactions to PULL uncommitted, conflicting effects. However, we don't believe such behaviors to be particularly interesting or realistic.

> *Application*: Many traditional STMs are opaque [10] (transactions cannot view the effects of other uncommitted transactions). Such systems never execute PULL operations marked as gUCmt and can only view operations that have been marked gCmt.
> *Application*: Some (non-opaque) transaction A may become *dependent* [30] on another transaction B if the effects of B are released to A before B commits. This is captured by B performing a PUSH of some effects that are then PULLed by A even though B has not committed.

The UNPULL **rule.** A PULLed operation may be removed from the local log. UNPULL criterion (*i*) is that the local log is $\mathsf{allowed}$ without operation *op*. Informally, this means that the transaction must not have done anything that depended on *op*. Without this criterion the local log might become invalid with respect to the sequential specification.

> *Applications*: Breaking dependencies [30].

The CMT **rule.** If there is a path through `tx` c that reaches `skip` (CMT criterion (*i*)), then the transaction can commit. There are three additional conditions: CMT criterion (*ii*) is that the local log L_1 must be contained within the global log G_1, indicating that *all of the transaction's operations have been pushed.* CMT criterion (*iii*) says that *all pulled operations correspond to transactions that have been committed.* Finally, CMT criterion (*iv*) is that the global log is updated to G_2 in which all of the transaction's operations are marked as committed. This is achieved with the $\mathsf{cmt}(G_1, L_1, G_2)$ predicate, defined at the bottom of Figure 5. The CMT rule serves as the instantaneous moment when all of a transaction's effects become permanent. Note that a transaction does not have to PULL all committed operations. Instead, transactions check whether they conflict with other transactions' operations each time they PUSH an operation.

5 Serializability

In this section we present the proof of serializability of PUSH/PULL. The proof is achieved by a simulation between a PUSH/PULL machine and the atomic semantics.

5.1 Mnemonics

There are a few helpful mnemonics to keep in mind for this proof. The first is a relationship between $\blacktriangleleft$ and $\preccurlyeq$. When there is a relationship of the form $op_1 \blacktriangleleft op_2$, and we have $\ell \cdot op_1 \cdot op_2$ then, after swapping the ops, we have the relationship $\ell \cdot op_1 \cdot op_2 \ \preccurlyeq\ \ell \cdot op_2 \cdot op_1$. That is, the order of the operations in the expression $op_1 \blacktriangleleft op_2$ is the order they will be in the log on the left-hand side of $\preccurlyeq$. Generally speaking, we will refer to logs on the right-hand side of $\preccurlyeq$ as the hypothetical log (the log of the atomic machine we simulate).

5.2 Lemmas

We develop lemmas that later help us establish the simulation relation between Push/Pull and the atomic semantics.

Lemma 5.1. *For all ℓ_1, ℓ_2, op, we have that $\ell_2 \blacktriangleleft op \;\wedge\; \mathsf{allowed}\ \ell_1 \cdot \ell_2 \cdot op \;\Rightarrow\; \mathsf{allowed}\ \ell_1 \cdot op$.*

Proof. First we consider the case where ℓ_2 is a singleton operation op'. This holds easily from the definition of $op' \blacktriangleleft op$ and prefix closure of $\mathsf{allowed}$. We then proceed by induction on ℓ_2. ☐

Lemma 5.2 (Transitivity of precongruence)**.** *For all ℓ_a, ℓ_b, ℓ_c, we have that $\ell_a \preccurlyeq \ell_b \wedge \ell_b \preccurlyeq \ell_c \Rightarrow \ell_a \preccurlyeq \ell_c$*

Proof. By coinduction, using the fact that $\mathsf{allowed}\ \ell_a \Rightarrow \mathsf{allowed}\ \ell_c$ holds by transitivity of implication. ☐

Lemma 5.3 (Precongruence over append)**.** *For all ℓ_a, ℓ_b, ℓ_c, we have that $\ell_a \preccurlyeq \ell_b \Rightarrow \ell_a \cdot \ell_c \preccurlyeq \ell_b \cdot \ell_c$.*

Proof. By induction on ℓ_c, using the definition of $\preccurlyeq$. ☐

Lemma 5.4 (Precongruence and bsstep)**.** *For all $c, \sigma, \sigma', \ell_1, \ell_1'$ and ℓ_2,*

$$(c,\sigma),\ell_1 \Downarrow \sigma',\ell_1' \;\wedge\; \ell_2 \preccurlyeq \ell_1 \quad\Rightarrow\quad \exists \ell_2'.(c,\sigma),\ell_2 \Downarrow \sigma',\ell_2' \;\wedge\; \ell_2' \preccurlyeq \ell_1'$$

Proof. We proceed by rule induction on the derivation of $(c,\sigma),\ell_1 \Downarrow \sigma',\ell_1'$. In the base case, $\ell_1' = \ell_1$. ☐

5.3 Invariants

In order to prove simulation, numerous invariants were necessary. We say that a predicate $P(T,G)$ is *invariant w.r.t.* $I_1, ..., I_n$ provided that

$$\begin{array}{ll} \forall \mathbf{T}\ G\ \mathbf{T}'\ G'. & \mathbf{T},G \xrightarrow{\mathsf{rt}} \mathbf{T}',G' \;\Rightarrow \\ & (\forall T \in \mathbf{T}.\ P(T,G) \wedge I_1(T,G) \wedge ... \wedge I_n(T,G)) \;\Rightarrow \\ & (\forall T' \in \mathbf{T}'.\ P(T',G')) \end{array}$$

Lemma 5.5. *For all $\mathbf{T}, G$, if $P(T,G)$ is invariant and $\mathbf{T},G \longrightarrow^* \mathbf{T}',G'$, then $P(T',G')$ for $\mathbf{T}'$.*

Proof. Induction on $\longrightarrow^*$. ☐

We use the following lemma to prove properties are invariant.

Lemma 5.6. *If for all reductions $T,G \longrightarrow T',G'$,*

1. *if $P(T,G)$ and $\bigwedge_{i\in[1,n]} I_i(T,G)$ implies $P(T',G')$; and*

2. *for all $\hat{T}$, if $P(T,G)$ and $\bigwedge_{i\in[1,n]} I_i(T,G)$ and $P(\hat{T},G)$ and $\bigwedge_{i\in[1,n]} I_i(\hat{T},G)$ implies $P(\hat{T},G')$*

then P is invariant wrt $I_1, \ldots, I_n$.

Part of proving that serializability holds, involves some log manipulations that arise from the commutativity conditions imposed by the Push/Pull model. This includes a few invariants. First, we have an invariant that ensures that the $\mathsf{pshd}/\mathsf{npshd}$ flags in the local log are consistent with whether or not the op is in the global log:

Lemma 5.7. *The following is invariant:*

$$\begin{array}{c} I_{LG}(\{c,\sigma,L\},G) \;\equiv\; \forall [\langle m,\sigma,\sigma',id\rangle, l] \in L \\ \left\{ \begin{array}{lcl} l = \mathsf{pshd}\ c & \Rightarrow & \langle m,\sigma,\sigma',id\rangle \in G \\ l = \mathsf{npshd}\ c & \Rightarrow & \langle m,\sigma,\sigma',id\rangle \notin G \end{array} \right\} \end{array}$$

Proof. By Lemma 5.6 and case analysis on reduction relation. ☐

Next, we have two invariants that ensure your uncommitted operations can move to the right of other transactions' uncommitted operations:

Lemma 5.8. *The following is invariant wrt I_{LG}:*

$$\begin{array}{rl} I_{\mathsf{slideR}}(\{c,\sigma,L\},G) \;\equiv & \forall [\langle m_1,\sigma_1,\sigma_1',id_1\rangle, \mathsf{pshd}] \in L \textit{ and } [\langle m_2,\sigma_2,\sigma_2',id_2\rangle, \mathsf{pshd} \mid \mathsf{npshd}] \notin L. \\ & G = G_1 \cdot [(\langle m_1,\sigma_1,\sigma_1',id_1\rangle, \mathsf{gUCmt})] \cdot G_2 \cdot [(\langle m_2,\sigma_2,\sigma_2',id_2\rangle, g)] \cdot G_3 \\ \Rightarrow & \langle m_1,\sigma_1,\sigma_1',id_1\rangle \blacktriangleleft \langle m_2,\sigma_2,\sigma_2',id_2\rangle \end{array}$$

Here the notion $\mathsf{pshd} \mid \mathsf{npshd}$ *means that the label can be either* pshd *or* npshd*, but not* pld*.*

Proof. To prove that I_{slideR} is invariant there are two cases to consider:

1. *Let* $T \equiv \{c,\sigma,L\} \in \mathbf{T}$ *be the thread that took a step to* $T' \equiv \{c',\sigma',L'\} \in \mathbf{T}'$*. Assume* $I_{\mathsf{slideR}}(\{c,\sigma,L\},G)$*.*
   Claim: $I_{\mathsf{slideR}}(\{c',\sigma',L'\},G')$*.*

   Pf. All cases (APP, UNAPP, PUSH, UNPUSH, PULL, UNPULL, CMT) are trivial.

2. *Let* $\{c,\sigma,L\} \in \mathbf{T}$ *be the thread that took a step and let* $\{\hat{c},\hat{\sigma},\hat{L}\} \in \mathbf{T}$ *be another thread. Assume* $I_{\mathsf{slideR}}(\{c,\sigma,L\},G)$ *and* $I_{\mathsf{slideR}}(\{\hat{c},\hat{\sigma},\hat{L}\},G)$*.*
   Claim: $I_{\mathsf{slideR}}(\{\hat{c},\hat{\sigma},\hat{L}\},G')$*.*

   Pf. Proof by case analysis on the step taken by $\{c,\sigma,L\}$, cases APP, UNAPP, UNPUSH, PULL, UNPULL, CMT are trivial. For case PUSH we rely on PUSH criterion (ii).

□

Lemma 5.9. *The following is invariant wrt I_{LG} and I_{slideR}:*

$$I_{\mathsf{slidePushed}}(\{c,\sigma,L\},G) \quad \equiv \quad G \;\preccurlyeq\; (G \setminus \lfloor L \rfloor_{\mathsf{pshd}}) \cdot (G \cap \lfloor L \rfloor_{\mathsf{pshd}})$$

Note that $\setminus$ *and* $\cap$ *preserves the order of their first arguments.*

Proof. By induction on L, using Lemma 5.8. □

The next invariants intuitively mean that if a transaction PUSHes operations out of order, the resulting log bares some precongruence to a log in which the operations were pushed in the correct order.

Lemma 5.10. *For all c,σ,L,G, the following is invariant:*

$$\begin{array}{rl} I_{\mathsf{reorderPUSH}}(\{c,\sigma,L_1 \cdot L_2\},G) \;\equiv & \forall [\langle m_1,\sigma_1,\sigma_1',id_1\rangle, \mathsf{pshd} \mid \mathsf{npshd}] \in L_1 \textit{ and } [\langle m_2,\sigma_2,\sigma_2',id_2\rangle, \mathsf{pshd} \mid \mathsf{npshd}] \in L_2 \\ & G = G_1 \cdot [(\langle m_2,\sigma_2,\sigma_2',id_2\rangle, \mathsf{gUCmt})] \cdot G_2 \cdot [(\langle m_1,\sigma_1,\sigma_1',id_1\rangle, \mathsf{gUCmt})] \cdot G_e \\ & \Rightarrow \langle m_2,\sigma_2,\sigma_2',id_2\rangle \blacktriangleleft \langle m_1,\sigma_1,\sigma_1',id_1\rangle \end{array}$$

Proof. To prove that $I_{\mathsf{reorderPUSH}}$ is invariant there are two cases to consider:

1. *Let* $t \equiv \{c,\sigma,L\} \in \mathbf{T}$ *be the thread that took a step to* $T' \equiv \{c',\sigma',L'\} \in \mathbf{T}'$*. Assume* $I_{\mathsf{reorderPUSH}}(\{c,\sigma,L\},G)$*.*
   Claim: $I_{\mathsf{reorderPUSH}}(\{c',\sigma',L'\},G')$*.*

   Pf. Cases APP, UNAPP, PULL, UNPULL, CMT are trivial. Case UNPUSH uses UNPUSH criterion (ii) and case PUSH uses PUSH criterion (i).

2. *Let* $\{c,\sigma,L\} \in \mathbf{T}$ *be the thread that took a step and let* $\{\hat{c},\hat{\sigma},\hat{L}\} \in \mathbf{T}$ *be some other thread. Assume* $I_{\mathsf{reorderPUSH}}(\{c,\sigma,L\},G)$ *and* $I_{\mathsf{reorderPUSH}}(\{\hat{c},\hat{\sigma},\hat{L}\},G)$*.*
   Claim: $I_{\mathsf{reorderPUSH}}(\{\hat{c},\hat{\sigma},\hat{L}\},G')$*.*

   Pf. Trivial because $I_{\mathsf{reorderPUSH}}$ only pertains to a transaction's own operations. □

The following Lemma indicates a log equivalence between one in which operations have been pushed in non-chronological order, and a log in which they have been pushed chronologically.

Lemma 5.11. *The following is invariant wrt $I_{\mathsf{reorderPUSH}}$*

$$\begin{array}{l} I_{\mathsf{chronPush}}(\{c,\sigma,L\},G) \quad \equiv \\ (G \smallsetminus \lfloor L \rfloor_{\mathsf{pshd}}) \cdot (G \cap \lfloor L \rfloor_{\mathsf{pshd}}) \preccurlyeq (G \smallsetminus \lfloor L \rfloor_{\mathsf{pshd}}) \cdot \lfloor L \rfloor_{\mathsf{pshd}} \end{array}$$

Proof. We prove a slightly stronger property

$$\begin{array}{l} (G \smallsetminus \lfloor L_1 \cdot L_2 \rfloor_{\mathsf{pshd}}) \cdot (G \cap \lfloor L_1 \cdot L_2 \rfloor_{\mathsf{pshd}}) \preccurlyeq \\ (G \smallsetminus \lfloor L_1 \cdot L_2 \rfloor_{\mathsf{pshd}}) \cdot (G \cap \lfloor L_1 \rfloor_{\mathsf{pshd}}) \cdot \lfloor L_2 \rfloor_{\mathsf{pshd}} \end{array}$$

By induction on the size of L_2. □

The next invariants intuitively mean that any operation $[\langle m_1, \sigma_1, \sigma_1', id_1 \rangle, \mathsf{npshd}\ c_1]$ can move to the left of some $[\langle m_2, \sigma_2, \sigma_2', id_2 \rangle, \mathsf{pshd}\ c_2]$ provided that the first operation is earlier than the second operation in the local log.

Lemma 5.12. *The following is invariant*

$$I_{\mathsf{localOrder}}(\{c,\sigma,L\},G) \equiv \left\{ \begin{array}{l} L = L_1 \cdot [\langle m_2, \sigma_2, \sigma_2', id_2 \rangle, \mathsf{npshd}\ c_2] \cdot L_2 \cdot [\langle m_1, \sigma_1, \sigma_1', id_1 \rangle, \mathsf{pshd}\ c_1] \cdot L_3 \\ \Rightarrow \langle m_1, \sigma_1, \sigma_1', id_1 \rangle \blacktriangleleft \langle m_2, \sigma_2, \sigma_2', id_2 \rangle \end{array} \right\}$$

Proof. To prove that $I_{\mathsf{localOrder}}$ is invariant there are two cases to consider:

1. *Let $t \equiv \{c,\sigma,L\} \in \mathbf{T}$ be the thread that took a step to $T' \equiv \{c',\sigma',L'\} \in \mathbf{T}'$. Assume $I_{\mathsf{localOrder}}(\{c,\sigma,L\},G)$. Claim: $I_{\mathsf{localOrder}}(\{c',\sigma',L'\},G')$.*

   Pf. Cases APP, UNAPP, PULL, UNPULL, CMT are trivial. Case PUSH uses PUSH criterion (i). Case UNPUSH uses UNPUSH criterion (i).

2. *Let $\{c,\sigma,L\} \in \mathbf{T}$ be the thread that took a step and let $\{\hat{c},\hat{\sigma},\hat{L}\} \in \mathbf{T}$ be some other thread. Assume $I_{\mathsf{localOrder}}(\{c,\sigma,L\},G)$ and $I_{\mathsf{localOrder}}(\{\hat{c},\hat{\sigma},\hat{L}\},G)$. Claim: $I_{\mathsf{localOrder}}(\{\hat{c},\hat{\sigma},\hat{L}\},G')$.*

   Pf. Trivial because $I_{\mathsf{localOrder}}$ only pertains to a transaction's own operations. □

The following lemma indicates that we can reorder a given thread's operations to match the order they were applied in the local log.

Lemma 5.13. *The following is invariant wrt $I_{\mathsf{localOrder}}$:*

$$\begin{array}{l} I_{\mathsf{localReorder}}(\{c,\sigma,L\},G) \quad \equiv \\ (G \smallsetminus \lfloor L \rfloor_{\mathsf{pshd}}) \cdot \lfloor L \rfloor_{\mathsf{pshd}} \cdot \lfloor L \rfloor_{\mathsf{npshd}} \preccurlyeq (G \smallsetminus \lfloor L \rfloor_{\mathsf{pshd}}) \cdot \lfloor L \rfloor_{\mathsf{pshd}}^{\mathsf{npshd}} \end{array}$$

Proof. We prove the stronger property

$$\begin{array}{l} (G \smallsetminus \lfloor L_1 \cdot L_2 \rfloor_{\mathsf{pshd}}) \cdot \lfloor L_1 \cdot L_2 \rfloor_{\mathsf{pshd}} \cdot \lfloor L_1 \cdot L_2 \rfloor_{\mathsf{npshd}} \preccurlyeq \\ (G \smallsetminus \lfloor L \rfloor_{\mathsf{pshd}}) \cdot \lfloor L_1 \rfloor_{\mathsf{pshd}}^{\mathsf{npshd}} \cdot \lfloor L_2 \rfloor_{\mathsf{pshd}} \cdot \lfloor L_2 \rfloor_{\mathsf{npshd}} \end{array}$$

by induction on L_1. □

5.4 Commit preservation invariant

The heart of the simulation requires that we prove an invariant of the system that the shared log is equivalent to what it would be if concurrently executing transactions removed their effects and applied them atomically. More precisely, imagine that at a given moment there is a shared log G, and a given thread $T = \{c, \sigma, L\}$ atomically marks all of its pushed operations as committed, reaching a shared log of G_{post}. Note that T may still have unpushed operations $\lfloor L \rfloor_{\mathsf{npshd}}$. The invariant states that, there is a precongruence between the shared log reached by completing T from $G_{\mathsf{post}} \cdot \lfloor L \rfloor_{\mathsf{npshd}}$ and the shared log that would have been reached if T rewound itself and atomically ran the entire transaction from G (that is, $G \smallsetminus L$, *i.e.* the previous shared log, with all operations belonging to T filtered out).

As described so far, the commit preservation invariant would look like the following:

$$\begin{array}{l} \forall G_{\mathsf{post}}.\ \mathsf{cmt}(G, L, G_{\mathsf{post}}) \Rightarrow \\ \forall \sigma', \ell_a.\ (c, \sigma), G_{\mathsf{post}} \cdot \lfloor L \rfloor_{\mathsf{npshd}} \Downarrow \sigma', \ell_a \Rightarrow \\ \ \exists \ell_b.\ \ \mathsf{otx}(\{c, \sigma, L\}), G \smallsetminus L \Downarrow \sigma', \ell_b \quad \wedge \quad \ell_a \preccurlyeq \ell_b \end{array}$$

where otx rewinds the transaction to its original state/code, recorded in L as follows:

$$\begin{array}{lcl} \mathsf{otx}(\{c, \sigma, []\}) & \equiv & \{c, \sigma, []\} \\ \mathsf{otx}(\{c, \sigma, [\langle m_1, \sigma_1, \sigma_1', id_1 \rangle, \mathsf{npshd}\ `c] \cdot L\}) & \equiv & \{`c, \sigma_1, []\} \\ \mathsf{otx}(\{c, \sigma, [\langle m_1, \sigma_1, \sigma_1', id_1 \rangle, \mathsf{pshd}\ `c] \cdot L\}) & \equiv & \{`c, \sigma_1, []\} \\ \mathsf{otx}(\{c, \sigma, [\langle m_1, \sigma_1, \sigma_1', id_1 \rangle, \mathsf{pld}] \cdot L\}) & \equiv & \mathsf{otx}(\{c, \sigma, L\}) \end{array}$$

Partial rewind. This is not enough to give us the simulation result as the property is not an invariant. As the system makes steps, which undo operations from the logs, the property must be closed with respect to these *backwards* steps. Thus we need the above to hold after any partial rewinding of the local log and/or partial removal of other transactions' uncommitted operations in the shared log.

Definition 5.1 (Self-rewind)**.** *A transaction's* self-rewind *denoted* $\{c, \sigma, L\}, G \ \circlearrowleft_{\mathit{self}}\ \{`c, `\sigma, `L\}, `G$ *is defined as:*

$$\frac{}{\{c, \sigma, L \cdot \langle m, \sigma_1, \sigma_2, id, \mathit{npshd}\ `c \rangle\}, G \ \circlearrowleft_{\mathit{self}}\ \{`c, \sigma_1, L\}, G}\ PRU$$

$$\frac{}{\{c, \sigma, L \cdot \langle m_1, \sigma_1, \sigma_1', id_1, \mathit{pshd}\ `c \rangle\}, G_1 \cdot [\langle m_1, \sigma_1, \sigma_1', id_1 \rangle, \mathit{gUCmt}] \cdot G_2 \ \circlearrowleft_{\mathit{self}}\ \{`c, \sigma_1, L\}, G_1 \cdot G_2}\ PRP$$

$$\frac{\{c, \sigma, L\}, G \ \circlearrowleft_{\mathit{self}}\ \{c, \sigma, L'\}, `G}{\{c, \sigma, L \cdot \cdot \langle m_1, \sigma_1, \sigma_1', id_1, \mathit{pld} \rangle\}, G \ \circlearrowleft_{\mathit{self}}\ \{c, \sigma, L'\}, `G}\ PRM \qquad \frac{}{\{c, \sigma, L\}, G \ \circlearrowleft_{\mathit{self}}\ \{c, \sigma, L\}, G}\ PRR$$

The self-rewind allows us to cope with the fact that a transaction may have PULLed operations from another uncommitted transaction. In particular, we preserve the fact that the transaction *may* be able to detangle from the uncommitted transaction, and atomically commit. In the definition above of $\circlearrowleft_{\mathsf{self}}$, the first two cases describe a transaction rewinding its log to an operation that has been APPlied but may or may not have been PUSHed. The transactional may pass beyond a PULLed operation via the third rule. Finally, the relation is reflexive.

The *shared log* partial rewind permits uncommitted operations of other transactions to be dropped from the shared log, and is defined as follows:

$$\frac{G_1 \ \circlearrowleft_L\ `G_1 \quad \langle m, \sigma, \sigma', id \rangle \notin L \quad G_2 \ \circlearrowleft_L\ `G_2}{G_1 \cdot [\langle m, \sigma, \sigma', id \rangle, \mathsf{gUCmt}] \cdot G_2 \ \circlearrowleft_L\ `G_1 \cdot `G_2} \qquad \frac{}{G \ \circlearrowleft_L\ G}$$

We can now state the commit preservation invariant as follows:

Definition 5.2 (Commit preservation invariant)**.** *For all G,*

$$
\mathit{cmtpres}(c,\sigma,L,G) \;\equiv\; \begin{cases}
\forall \text{‘‘}G.\; G \;\circlearrowleft_L\; \text{‘‘}G. \;\Rightarrow & (0)\\
\forall \{\text{‘}c,\text{‘}\sigma,\text{‘}L\}.\; \{c,\sigma,L\},\text{‘‘}G \;\circlearrowleft_{\mathit{self}}\; \{\text{‘}c,\text{‘}\sigma,\text{‘}L\},\text{‘}G \;\Rightarrow & (1)\\
\forall G_{\mathit{post}}.\; \mathit{cmt}(\text{‘}G,\text{‘}L,G_{\mathit{post}}) \;\Rightarrow & (2)\\
\forall \sigma',\ell_a.\; (\text{‘}c,\text{‘}\sigma),G_{\mathit{post}}\cdot\lfloor \text{‘}L\rfloor_{\mathit{npshd}} \Downarrow \sigma',\ell_a \;\Rightarrow & (3)\\
\;\exists \ell_b.\;\; \mathit{otx}(\{\text{‘}c,\text{‘}\sigma,\text{‘}L\}),G \smallsetminus L \Downarrow \sigma',\ell_b \;\;\wedge\;\; \ell_a \preccurlyeq \ell_b & (4)
\end{cases}
$$

Intuitively, this invariant means that under any dropping of others' uncommitted operations (Line 0) and after partially rewinding $\circlearrowleft_{\mathsf{self}}$ your local transaction to some local log ‘L (Line 1), if you are now able to atomically "commit" by swapping your commit flags (Line 2) and running the rest of your transaction (Line 3), then the shared log reached ℓ_a, is contained within a shared log ℓ_b that would have been reached if the thread appended its entire transaction to G atomically.

Lemma 5.14. *For all ℓ,ℓ', $\ell \preccurlyeq \ell' \;\Rightarrow\; (\mathit{cmtpres}(\{c,\sigma,L\},G) \Rightarrow \mathit{cmtpres}(\{c,\sigma,L'\},G))$*

Proof. Trivial. □

The following lemma indicates that partial rewind behaviors are included in the Push/Pull transition system:

Lemma 5.15. *Invariance of*

$$
I_{\subseteq}(\{c,\sigma,L\},G) \;\;\equiv\;\; \left\{ \begin{array}{lcl} (\{c,\sigma,L\},G & \circlearrowleft_{\mathit{self}} & \{\text{‘}c,\text{‘}\sigma,\text{‘}L\},\text{‘}G) \;\Rightarrow \\ (\{c,\sigma,L\},G & \xrightarrow{\mathsf{rt}} & \{\text{‘}c,\text{‘}\sigma,\text{‘}L\},\text{‘}G) \end{array} \right\}
$$

Proof. Case analysis. □

Lemma 5.16. *$\mathit{cmtpres}$ is invariant w.r.t. $I_{\mathit{slidePushed}}$, $I_{\mathit{chronPush}}$, $I_{\mathit{localReorder}}$, and $I_{\subseteq}$.*

Proof. We assume that $\mathsf{cmtpres}$ holds for some $\mathbf{T},G$, that $\mathbf{T},G \xrightarrow{\mathsf{rt}} \mathbf{T}',G'$ and then show that $\mathsf{cmtpres}$ holds for $\mathbf{T}',G'$. We split the proof into two cases, proving each one by induction on $\xrightarrow{\mathsf{rt}}$.

1. *Let $t \equiv \{c,\sigma,L\} \in \mathbf{T}$ be the thread that took a step to $T' \equiv \{c',\sigma',L'\} \in \mathbf{T}'$. (Generally speaking, we reserve the primed notation for components of $\mathbf{T}',G'$, for example when we apply the invariant to T' obtaining variables such as ℓ'_a.) Assume $\mathit{cmtpres}(\{c,\sigma,L\},G)$.*
   Claim: $\mathit{cmtpres}(\{c',\sigma',L'\},G')$.

   Pf. By induction, with the inductive hypothesis

$$
\begin{array}{lr}
\forall \text{‘‘}G.\; G \;\circlearrowleft_L\; \text{‘‘}G. \;\Rightarrow & (0)\\
\forall \{\text{‘}c,\text{‘}\sigma,\text{‘}L\}.\; \{c,\sigma,L\},\text{‘‘}G \;\circlearrowleft_{\mathsf{self}}\; \{\text{‘}c,\text{‘}\sigma,\text{‘}L\},\text{‘}G \;\Rightarrow & (1)\\
\forall G_{\mathsf{post}}.\; \mathsf{cmt}(\text{‘}G,\text{‘}L,G_{\mathsf{post}}) \;\Rightarrow & (2)\\
\forall \sigma',\ell_a.\; (\text{‘}c,\text{‘}\sigma),G_{\mathsf{post}}\cdot\lfloor \text{‘}L\rfloor_{\mathsf{npshd}} \Downarrow \sigma',\ell_a \;\Rightarrow & (3)\\
\;\exists \ell_b.\;\; \mathsf{otx}(\{\text{‘}c,\text{‘}\sigma,\text{‘}L\}),G \smallsetminus L \Downarrow \sigma',\ell_b \;\;\wedge\;\; \ell_a \preccurlyeq \ell_b & (4)
\end{array}
$$

   Note in Line 4 that $\smallsetminus$ does not remove operations from G that have been pld into L.

   Case APP**:** $L' = L\cdot[\langle m,\sigma,\sigma',id\rangle,\mathsf{npshd}\;c]$. From $\xrightarrow{\mathsf{fwd}}$, L allows $\langle m,\sigma,\sigma',id\rangle$ and $(m,c') \in \mathsf{step}(c)$. Also, note that $\mathsf{otx}(\{\text{‘}c,\text{‘}\sigma,\text{‘}L\}) = \mathsf{otx}(\{\text{‘}c',\text{‘}\sigma',\text{‘}L'\})$. We must show that the invariant holds for $\mathbf{T}',G'$ where L' has one more unpushed operation. That is, we must show that

$$
\begin{array}{lr}
\forall \text{‘}G.\; G \;\circlearrowleft_L\; \text{‘‘}G \;\Rightarrow & (0)\\
\forall \{\text{‘}c,\text{‘}\sigma,\text{‘}L\}.\; \{c',\sigma',L'\},\text{‘‘}G \;\circlearrowleft_{\mathsf{self}}\; \{\text{‘}c,\text{‘}\sigma,\text{‘}L\},\text{‘}G \;\Rightarrow & (1)\\
\forall G_{\mathsf{post}}.\; \mathsf{cmt}(\text{‘}G,L',G_{\mathsf{post}}) \;\Rightarrow & (2)\\
\forall \sigma',\ell_a.\; (c',\sigma'),G_{\mathsf{post}}\cdot\lfloor L'\rfloor_{\mathsf{npshd}} \Downarrow \sigma',\ell_a \;\Rightarrow & (3)\\
\;\exists \ell_b.\;\; \mathsf{otx}(\{c',\sigma',L'\}),G \smallsetminus L' \Downarrow \sigma',\ell_b \;\;\wedge\;\; \ell_a \preccurlyeq \ell_b & (4)
\end{array}
$$

Note that $G' = G$. So we can chose the same $\text{‘‘}G$ from the inductive hypothesis. Consequently, Lines 2,3,4 hold when we rewind to every *strictly* previous $\{\text{‘}c, \text{‘}\sigma, \text{‘}L\}$. However, we must show that those lines hold without any rewind:

$$\begin{array}{lr}
\forall G_{\mathsf{post}}.\ \mathsf{cmt}(\text{‘}G, L', G_{\mathsf{post}}) \Rightarrow & (2)\\
\forall \sigma', \ell_a.\ (c', \sigma'), G_{\mathsf{post}} \cdot \lfloor L' \rfloor_{\mathsf{npshd}} \Downarrow \sigma', \ell_a \Rightarrow & (3)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{c', \sigma', L'\}), G \smallsetminus L' \Downarrow \sigma', \ell_b \quad \wedge \quad \ell_a \preccurlyeq \ell_b & (4)
\end{array}$$

G_{post} is the same as in the inductive hypothesis because the new operation is yet unpushed. What remains is to show that

$$\begin{array}{lr}
\forall \sigma', \ell_a.\ (c, \sigma), G_{\mathsf{post}} \cdot \lfloor L \rfloor_{\mathsf{npshd}} \Downarrow \sigma', \ell_a \Rightarrow & (i)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{c, \sigma, L\}), G \smallsetminus L \Downarrow \sigma', \ell_b \quad \wedge \quad \ell_a \preccurlyeq \ell_b & (ii)
\end{array}$$

implies

$$\begin{array}{lr}
\forall \sigma', \ell_a.\ (c', \sigma'), G_{\mathsf{post}} \cdot \lfloor L \rfloor_{\mathsf{npshd}} \cdot [\langle m, \sigma, \sigma', id \rangle] \Downarrow \sigma', \ell_a \Rightarrow & (iii)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{c, \sigma, L\}), G \smallsetminus (L \cdot [\langle m, \sigma, \sigma', id \rangle]) \Downarrow \sigma', \ell_b \quad \wedge \quad \ell_a \preccurlyeq \ell_b & (iv)
\end{array}$$

where $\mathsf{cmt}(\text{‘}G, L', G_{\mathsf{post}})$. Above Line (ii) is equivalent to (iv) because id is not in G. We instantiate $\Downarrow$ in Line (i) for the case where $(m, c') \in \mathsf{step}(c)$, which gives us precisely Line (iii).

Case PUSH**:** In this case, we let $L = L_1 \cdot [\langle m, \sigma, \sigma', id \rangle, \mathsf{npshd}\ c_\alpha] \cdot L_2$. The post-log L' is identical, except the npshd flag is replaced with pshd. To prove that the invariant still holds, we must again consider all rewindings of the logs. We chose the same $G \circlearrowleft_L \text{‘‘}G$ as in the inductive hypothesis. If the local log is rewound to L_1 (or some prefix thereof), then we have passed the operation in question, and Lines 2–4 of the invariant hold directly from the inductive hypothesis. What remains is to prove that the invariant holds after rewinding to some $\hat{L} \equiv L_1 \cdot [\langle m, \sigma, \sigma', id \rangle, \mathsf{npshd}\ c_\alpha] \cdot \hat{L}_2$ where $\hat{L}_2$ is a prefix of L_2. We must now show that

$$\begin{array}{lr}
\forall G'_{\mathsf{post}}.\ \mathsf{cmt}(\text{‘}G, L_1 \cdot [\langle m, \sigma, \sigma', id \rangle, \mathsf{pshd}\ c_\alpha] \cdot \hat{L}_2, G'_{\mathsf{post}}) \Rightarrow & (2)\\
\forall \sigma_{post}, \ell_a.\ (\text{‘}c, \text{‘}\sigma), G'_{\mathsf{post}} \cdot \lfloor \hat{L} \rfloor_{\mathsf{npshd}} \Downarrow \sigma_{post}, \ell_a \Rightarrow & (3)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{\text{‘}c, \text{‘}\sigma, \text{‘}L\}), G \smallsetminus \hat{L} \Downarrow \sigma_{post}, \ell_b \quad \wedge \quad \ell_a \preccurlyeq \ell_b & (4)
\end{array}$$

Let G_{post} be from the inductive hypothesis. Note that $G'_{\mathsf{post}} = G_{\mathsf{post}} \cdot [\langle m, \sigma, \sigma', id \rangle, \mathsf{gCmt}]$. Distributing the $\lfloor\ \rfloor_{\mathsf{npshd}}$ over append, we can drop the op in Line 3, as well as replacing G'_{post}. What remains is to show:

$$\begin{array}{lr}
\forall \sigma_{post}, \ell_a.\ (\text{‘}c, \text{‘}\sigma), G_{\mathsf{post}} \cdot [\langle m, \sigma, \sigma', id \rangle] \cdot \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{npshd}} \Downarrow \sigma_{post}, \ell_a \Rightarrow & (3)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{\text{‘}c, \text{‘}\sigma, \text{‘}L\}), G \smallsetminus \hat{L} \Downarrow \sigma_{post}, \ell_b \quad \wedge \quad \ell_a \preccurlyeq \ell_b & (4)
\end{array}$$

We now perform the following rewrites of the log in Line 3, with each successive log, having precongruence with the previous:

$$\begin{array}{lrcll}
 & G_{\mathsf{post}} & \cdot & [\langle m, \sigma, \sigma', id \rangle] \cdot \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{npshd}} & \\
\preccurlyeq & (G_{\mathsf{post}} \smallsetminus \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}) \cdot (G_{\mathsf{post}} \cap \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}) & \cdot & [\langle m, \sigma, \sigma', id \rangle] \cdot \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{npshd}} & \textit{Lm 5.9}\\
\preccurlyeq & (G_{\mathsf{post}} \smallsetminus \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}) \cdot (G_{\mathsf{post}} \cap \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}) & \cdot & \lfloor L_1 \rfloor_{\mathsf{npshd}} \cdot [\langle m, \sigma, \sigma', id \rangle] \cdot \lfloor \hat{L}_2 \rfloor_{\mathsf{npshd}} & \textsc{push}\ (i)\\
\preccurlyeq & (G_{\mathsf{post}} \smallsetminus \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}) \cdot \lfloor L_1 \cdot L_2 \rfloor_{\mathsf{pshd}} & \cdot & \lfloor L_1 \rfloor_{\mathsf{npshd}} \cdot [\langle m, \sigma, \sigma', id \rangle] \cdot \lfloor \hat{L}_2 \rfloor_{\mathsf{npshd}} & \textit{Lm 5.11}\\
\preccurlyeq & (G_{\mathsf{post}} \smallsetminus \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}) & \cdot & \lfloor L_1 \cdot [\langle m, \sigma, \sigma', id \rangle, __] \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}^{\mathsf{npshd}} & \textit{Lm 5.13}
\end{array}$$

Finally, we use Lemma 5.4 to substitute this rewritten log into place on the left side of $\Downarrow$ on Line 3, and we have proved that the invariant still holds.

Cases UNPUSH**:** In this case, we let

$$\begin{array}{rcl}
L & = & L_1 \cdot [\langle m, \sigma, \sigma', id \rangle, \mathsf{pshd}\ c_\alpha] \cdot L_2\\
G & = & G_1 \cdot [\langle m, \sigma, \sigma', id \rangle, \mathsf{gUCmt}] \cdot G_2
\end{array}$$

The post-log L' is identical to L, except the pshd flag is replaced with npshd. $G' = G_1 \cdot G_2$. To prove that the invariant still holds, we must again consider all rewindings of the logs. From the inductive hypothesis we have

$$G_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{gUCmt}] \cdot G_2 \;\circlearrowleft_L\; ``G_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{gUCmt}] \cdot ``G_2.$$

If the local log is rewound to L_1 (or some prefix thereof), then we have passed the operation in question, and Lines 2–4 of the invariant hold directly from the inductive hypothesis. What remains is to prove that the invariant holds after local rewinding to some

$$\hat{L} \equiv L_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{pshd}\ c_\alpha] \cdot \hat{L}_2 \text{ and } `G_1 \cdot `G_2$$

where $\hat{L}_2$ is a prefix of L_2. We must now show that

$$\begin{array}{lr}
\forall G'_{\mathsf{post}}.\ \mathsf{cmt}(`G_1 \cdot `G_2, L_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{npshd}\ c_\alpha] \cdot \hat{L}_2, G'_{\mathsf{post}}) \Rightarrow & (2)\\
\forall \sigma_{post}, \ell_a.\ (`c, `\sigma), G'_{\mathsf{post}} \cdot \lfloor L_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{npshd}\ c_\alpha] \cdot \hat{L}_2 \rfloor_{\mathsf{npshd}} \Downarrow \sigma_{post}, \ell_a \Rightarrow & (3)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{`c, `\sigma, `L\}), G \smallsetminus \hat{L} \Downarrow \sigma_{post}, \ell_b \quad \wedge \quad \ell_a \preccurlyeq \ell_b & (4)
\end{array}$$

Here, G'_{post} is identical to G_{post}, except that $\langle m, \sigma, \sigma', id\rangle$ has been dropped. We now perform the following rewrites of the log in Line 3, with each successive log having precongruence with its previous:

$$\begin{array}{lrcll}
 & G'_{\mathsf{post}} & \cdot & \lfloor L_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{npshd}\ c_\alpha] \cdot \hat{L}_2 \rfloor_{\mathsf{npshd}} & \\
\preccurlyeq & (G'_{\mathsf{post}} \smallsetminus \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}) \cdot (G'_{\mathsf{post}} \cap \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}) & \cdot & \lfloor L_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{npshd}\ c_\alpha] \cdot \hat{L}_2 \rfloor_{\mathsf{npshd}} & \textit{Lm 5.9}\\
\preccurlyeq & (G'_{\mathsf{post}} \smallsetminus \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}) \cdot \lfloor L_1 \cdot L_2 \rfloor_{\mathsf{pshd}} & \cdot & \lfloor L_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{npshd}\ c_\alpha] \cdot \hat{L}_2 \rfloor_{\mathsf{npshd}} & \textit{Lm 5.11}\\
\preccurlyeq & (G'_{\mathsf{post}} \smallsetminus \lfloor L_1 \cdot \hat{L}_2 \rfloor_{\mathsf{pshd}}) & \cdot & \lfloor L_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{npshd}\ c_\alpha] \cdot \hat{L}_2 \rfloor^{\mathsf{npshd}}_{\mathsf{pshd}} & \textit{Lm 5.13}
\end{array}$$

Finally, we use Lemma 5.4 to substitute this rewritten log in to place on the left side of $\Downarrow$ on Line 3, and prove that the invariant still holds.

Case CMT**:** We again pick the same $``G$ from the inductive hypothesis. Now, since the transaction has performed a CMT, $L' = []$ and $`G = ``G$. So we need only show that

$$\begin{array}{lr}
\forall G'_{\mathsf{post}}.\ \mathsf{cmt}(`G, [], G_{\mathsf{post}}) \Rightarrow & (2)\\
\forall \sigma_{post}, \ell_a.\ (c', \sigma'), G'_{\mathsf{post}} \Downarrow \sigma_{post}, \ell_a \Rightarrow & (3)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{c', \sigma', []\}), G \Downarrow \sigma_{post}, \ell_b \quad \wedge \quad \ell_a \preccurlyeq \ell_b & (4)
\end{array}$$

Now, $G = G_{\mathsf{post}}$ and $\mathsf{otx}(\{c', \sigma', []\}) = (c', \sigma')$. Thus, Lines 3 and 4 are identical, and we can use ℓ_a as a witness for ℓ_b.

Case PULL**:** After PULL, $L' = L \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{pld}]$. For all rewinds of L' that produce prefixes of L, the invariant holds due to the inductive hypothesis. So we need only show that it holds for the unrewound L'. However, in that case this pulled operation is filtered out by the $\lfloor\ \rfloor_{\mathsf{npshd}}$ in Line 3 of the invariant, so it holds from the inductive hypothesis.

Case UNPULL**:** Let $L = L_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{pld}] \cdot L_2$. Due to the UNPULL rule, $L = L_1 \cdot L_2 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{pld}]$ is $\mathsf{allowed}$. This case then follows from the fact that our inductive hypothesis is closed under rewind, Lemma 5.14, and that Lines 2,3,4 of the inductive hypothesis don't depend on PULLed operations.

Case UNAPP**:** This case follows from the fact that our inductive hypothesis is closed under rewind.

Cases NONDETL, NONDETR, LOOP, SEMI, SEMISKIP**:** Details omitted because they are uninteresting.

2. *Let* $\{c, \sigma, L\} \in \mathbf{T}$ *be the thread that took a step and let* $\{\hat{c}, \hat{\sigma}, \hat{L}\} \in \mathbf{T}$ *be some other thread. Assume* $\mathit{cmtpres}(\{c, \sigma, L\}, G)$ *and* $\mathit{cmtpres}(\{\hat{c}, \hat{\sigma}, \hat{L}\}, G)$.
Claim: $\mathit{cmtpres}(\{\hat{c}, \hat{\sigma}, \hat{L}\}, G')$.

Pf. We use the notations T, T', and $\hat{T}$. Proof by induction on the step taken by $\{c, \sigma, L\}$, with inductive hypothesis

$$\begin{array}{lr}
\forall ‘‘G.\ G\ \circlearrowleft_{L}\ ‘‘G \Rightarrow & (0)\\
\forall \{‘c, ‘\sigma, ‘L\}\ ‘G.\ \{c, \sigma, L\}, ‘‘G\ \circlearrowleft_{\mathsf{self}}\ \{‘c, ‘\sigma, ‘L\}, ‘G\ \Rightarrow & (1)\\
\forall G_{\mathsf{post}}.\ \mathsf{cmt}(G, ‘L, G_{\mathsf{post}})\ \Rightarrow & (2)\\
\forall \sigma_{post}, \ell_a.\ (‘c, ‘\sigma), G_{\mathsf{post}} \cdot \lfloor ‘L \rfloor_{\mathsf{npshd}}\ \Downarrow\ \sigma_{post}, \ell_a\ \Rightarrow & (3)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{‘c, ‘\sigma, ‘L\}), G \smallsetminus ‘L \Downarrow \sigma_{post}, \ell_b\ \ \wedge\ \ \ell_a \preccurlyeq \ell_b & (4)
\end{array}$$

and

$$\begin{array}{lr}
\forall ‘‘G.\ G\ \circlearrowleft_{\hat{L}}\ ‘‘G \Rightarrow & (5)\\
\forall \{‘c, ‘\sigma, ‘L\}\ ‘G.\ \{\hat{c}, \hat{\sigma}, \hat{L}\}, ‘‘G\ \circlearrowleft_{\mathsf{self}}\ \{‘c, ‘\sigma, ‘L\}, ‘G\ \Rightarrow & (6)\\
\forall G_{\mathsf{post}}.\ \mathsf{cmt}(‘G, ‘L, G_{\mathsf{post}})\ \Rightarrow & (7)\\
\forall \sigma_{post}, \ell_a.\ (‘c, ‘\sigma), G_{\mathsf{post}} \cdot \lfloor ‘L \rfloor_{\mathsf{npshd}}\ \Downarrow\ \sigma_{post}, \ell_a\ \Rightarrow & (8)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{‘c, ‘\sigma, ‘L\}), G \smallsetminus ‘L \Downarrow \sigma_{post}, \ell_b\ \ \wedge\ \ \ell_a \preccurlyeq \ell_b & (9)
\end{array}$$

Cases APP, UNAPP, PULL, UNPULL**:** In all of these cases, $G' = G$, so $\mathsf{cmtpres}(\{\hat{c}, \hat{\sigma}, \hat{L}\}, G')$ is trivial.

Case PUSH**:** $G' = G \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{gUCmt}]$. We must show that Lines 5–9 still hold for G'. In this case, G'_{post} is similar to what it was in the inductive hypothesis, but with the additional uncommitted operation $\langle m, \sigma, \sigma', id\rangle$ that was appended by T. So we must show that:

$$\begin{array}{lr}
\forall ‘‘G.\ G'\ \circlearrowleft_{\hat{L}}\ ‘‘G \Rightarrow & (5)\\
\forall \{‘c, ‘\sigma, ‘L\}\ ‘G.\ \{\hat{c}, \hat{\sigma}, \hat{L}\}, ‘‘G\ \circlearrowleft_{\mathsf{self}}\ \{‘c, ‘\sigma, ‘L\}, ‘G\ \Rightarrow & (6)\\
\forall G'_{\mathsf{post}}.\ \mathsf{cmt}(G \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{gUCmt}], ‘L, G'_{\mathsf{post}})\ \Rightarrow & (7)\\
\forall \sigma_{post}, \ell_a.\ (‘c, ‘\sigma), G'_{\mathsf{post}} \cdot \lfloor ‘L \rfloor_{\mathsf{npshd}}\ \Downarrow\ \sigma_{post}, \ell_a\ \Rightarrow & (8)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{‘c, ‘\sigma, ‘L\}), (G \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{gUCmt}]) \smallsetminus ‘L \Downarrow \sigma_{post}, \ell_b\ \ \wedge\ \ \ell_a \preccurlyeq \ell_b & (9)
\end{array}$$

In the case where $\circlearrowleft_{\hat{L}}$ removes the PUSHed op, Lines 5–9 follow directly from the inductive hypothesis. Otherwise, there is now a new operation $\langle m, \sigma, \sigma', id\rangle$ that is not in the local log $\hat{L}$. We now perform the following rewrites of the log in Line 8, with each successive log, having precongruence with the previous:

$$\begin{array}{llcll}
 & G'_{\mathsf{post}} & \cdot & [\langle m, \sigma, \sigma', id\rangle] \cdot \lfloor ‘L \rfloor_{\mathsf{npshd}} & \\
\preccurlyeq & (G'_{\mathsf{post}} \smallsetminus \lfloor ‘L \rfloor_{\mathsf{pshd}}) \cdot (G'_{\mathsf{post}} \cap \lfloor ‘L \rfloor_{\mathsf{pshd}}) & \cdot & [\langle m, \sigma, \sigma', id\rangle] \cdot \lfloor ‘L \rfloor_{\mathsf{npshd}} & \textit{Lm 5.9}\\
\preccurlyeq & (G'_{\mathsf{post}} \smallsetminus \lfloor ‘L \rfloor_{\mathsf{pshd}}) \cdot [\langle m, \sigma, \sigma', id\rangle] \cdot (G'_{\mathsf{post}} \cap \lfloor ‘L \rfloor_{\mathsf{pshd}}) & \cdot & \lfloor ‘L \rfloor_{\mathsf{npshd}} & \text{PUSH } \textit{(ii)}\\
\preccurlyeq & ((G'_{\mathsf{post}} \cdot [\langle m, \sigma, \sigma', id\rangle]) \smallsetminus \lfloor ‘L \rfloor_{\mathsf{pshd}}) \cdot (G'_{\mathsf{post}} \cap \lfloor ‘L \rfloor_{\mathsf{pshd}}) & \cdot & \lfloor ‘L \rfloor_{\mathsf{npshd}} & \textit{via disjointness}\\
\preccurlyeq & ((G'_{\mathsf{post}} \cdot [\langle m, \sigma, \sigma', id\rangle]) \smallsetminus \lfloor ‘L \rfloor_{\mathsf{pshd}}) \cdot \lfloor ‘L \rfloor_{\mathsf{pshd}} & \cdot & \lfloor ‘L \rfloor_{\mathsf{npshd}} & \textit{Lm 5.11}\\
\preccurlyeq & ((G'_{\mathsf{post}} \cdot [\langle m, \sigma, \sigma', id\rangle]) \smallsetminus \lfloor ‘L \rfloor_{\mathsf{pshd}}) & \cdot & ‘L & \textit{Lm 5.13}
\end{array}$$

Finally, we use Lemma 5.4 to substitute this rewritten log into place on the left side of $\Downarrow$ on Line 8, and we have proved that the invariant still holds.

Case UNPUSH**:** Let $G = G_1 \cdot [\langle m, \sigma, \sigma', id\rangle, \mathsf{gUCmt}] \cdot G_2$ and $G' = G_1 \cdot G_2$. We must show that:

$$\begin{array}{lr}
\forall ‘‘G.\ G'\ \circlearrowleft_{\hat{L}}\ ‘‘G \Rightarrow & (5)\\
\forall \{‘c, ‘\sigma, ‘L\}\ ‘G.\ \{\hat{c}, \hat{\sigma}, \hat{L}\}, ‘‘G\ \circlearrowleft_{\mathsf{self}}\ \{‘c, ‘\sigma, ‘L\}, ‘G\ \Rightarrow & (6)\\
\forall G_{\mathsf{post}}.\ \mathsf{cmt}(G_1 \cdot G_2, ‘L, G_{\mathsf{post}})\ \Rightarrow & (7)\\
\forall \sigma_{post}, \ell_a.\ (‘c, ‘\sigma), G_{\mathsf{post}} \cdot \lfloor ‘L \rfloor_{\mathsf{npshd}}\ \Downarrow\ \sigma_{post}, \ell_a\ \Rightarrow & (8)\\
\quad \exists \ell_b.\ \ \mathsf{otx}(\{‘c, ‘\sigma, ‘L\}), G_1 \cdot G_2) \smallsetminus ‘L \Downarrow \sigma_{post}, \ell_b\ \ \wedge\ \ \ell_a \preccurlyeq \ell_b & (9)
\end{array}$$

This holds based on the fact that for all $\{c, \sigma, L\}, op, G_1, G_2$ that

$$op \notin L \wedge \mathsf{cmtpres}(\{c, \sigma, L\}, G_1 \cdot [op, \mathsf{gUCmt}] \cdot G_2)\ \ \Rightarrow\ \ \mathsf{cmtpres}(\{c, \sigma, L\}, G_1 \cdot G_2)$$

which we prove by induction.

Case CMT**:** In this case, $\mathsf{cmt}(G, L, G')$, so all of the operations in G belonging to T are now marked as gCmt. We must show that:

$$
\begin{aligned}
&\forall ``G.\ G' \ \circlearrowleft_{\hat{L}}\ ``G \ \Rightarrow && (5)\\
&\forall \{`c, `\sigma, `L\}\ `G.\ \{\hat{c}, \hat{\sigma}, \hat{L}\} \ \circlearrowleft_{\mathsf{self}}\ \{`c, `\sigma, `L\} \ \Rightarrow && (6)\\
&\forall G'_{\mathsf{post}}.\ \mathsf{cmt}(G', `L, G_{\mathsf{post}}) \ \Rightarrow && (7)\\
&\forall \sigma_{post}, \ell_a.\ (`c, `\sigma), G'_{\mathsf{post}} \cdot \lfloor `L \rfloor_{\mathsf{npshd}} \Downarrow \sigma_{post}, \ell_a \ \Rightarrow && (8)\\
&\ \exists \ell_b.\ \ \mathsf{otx}(\{`c, `\sigma, `L\}), G' \smallsetminus `L \Downarrow \sigma_{post}, \ell_b \ \ \wedge \ \ \ell_a \preccurlyeq \ell_b && (9)
\end{aligned}
$$

This follows easily from the fact that $\lfloor G \rfloor_{\mathsf{gCmt}} \subseteq \lfloor G' \rfloor_{\mathsf{gCmt}}$

Cases NONDETL, NONDETR, LOOP, SEMI, SEMISKIP**:** Details omitted because they are uninteresting.

□

5.5 Main Theorem

We now prove the main theorem via simulation between the PUSH/PULL machine and an atomic machine. The theorem depends strongly on the $\mathsf{cmtpres}$ invariant.

Theorem 5.17 (Serializability)**.** PUSH/PULL *is serializable.*

Proof. Via a simulation relation between $\rightarrow^*$ and $\xrightarrow{a}^*$. The simulation relation is defined as follows:

$$\mathbf{T}, G \sim \mathbf{A}, \ell \ \ \equiv \ \ (\mathsf{map\ rewind}\ \mathbf{T}) = \mathbf{A} \ \wedge \ \lfloor G \rfloor_{\mathsf{gCmt}} \preccurlyeq \ell$$

where rewind T rolls a transaction back to its original code (details pertaining to semi-colon treatment in rewind omitted for simplicity). We define $\mathbf{T} \sim \mathbf{A}$ and $G \sim \ell$ with the appropriate conjunct from above.

To prove simulation, we show that for every $\mathbf{T}, G \rightarrow^* \mathbf{T}', G'$ such that $\mathbf{T}, G \sim \mathbf{A}, \ell$, there exists some $\mathbf{A}', \ell'$ such that $\mathbf{A}, \ell \xrightarrow{a}^* \mathbf{A}', \ell'$ and that $\mathbf{T}', G' \sim \mathbf{A}', \ell'$. The reflexive, transitive and thread-end rules are straight-forward. What remains is aligning AMACH_ONE and MACH_ONE. In this case, we prove a helper lemma that shows that the simulation relation holds after each single step $\rightarrow$. That is, for every $\mathbf{T}, G \rightarrow \mathbf{T}', G'$ such that $\mathbf{T}, G \sim \mathbf{A}, \ell$, there exists some $\mathbf{A}', \ell'$ such that $\mathbf{A}, \ell \xrightarrow{a}^* \mathbf{A}', \ell'$ and that $\mathbf{T}', G' \sim \mathbf{A}', \ell'$.

So we must consider each $\rightarrow$ step from Figures 5 and 6 and show that an appropriate $\mathbf{A}', \ell'$ can be found. In each case, the inductive hypothesis gives us that the simulation relation **rewind**s all uncommitted transactions in $\mathbf{T}$ to obtain $\mathbf{A}$ and **drop**s all uncommitted operations from G to obtain ℓ. Moreover, we rely on all of the invariants holding for $\mathbf{T}, G$ as well as $\mathbf{T}', G'$ (most significantly, the $\mathsf{cmtpres}$ invariant).

Case APP**:** This step is straight-forward. We let $\mathbf{A}' = \mathbf{A}$ and $\ell' = \ell$. The inductive hypothesis tells us that $\mathbf{T}, G \sim \mathbf{A}, \ell$ and, after an APP step, $\mathbf{T}'$ is very similar to $\mathbf{T}$, except with one more operation $\langle m, \sigma_1, \sigma_2, id, \mathsf{npshd}\ c \rangle$ in the local log of a single thread.

Case UNAPP**:** Similar to APP.

Case PUSH**:** In this case a new operation has been PUSHed into the shared log. Since the new operation has not been committed, it will be filtered out by rewind so, again, we can let $\mathbf{A}' = \mathbf{A}$ and $\ell' = \ell$.

Case UNPUSH**:** In this case, we first use a simple invariant that the operation that is UNPUSHed was not yet committed. Then, we like the PUSH case, since the new operation has not been committed, it will be filtered out by rewind so, again, we can let $\mathbf{A}' = \mathbf{A}$ and $\ell' = \ell$.

Cases PULL, UNPULL**:** Let $\mathbf{A}' = \mathbf{A}$ and $\ell' = \ell$. In both PULL and UNPULL, the shared log is unchanged, so **drop**ping leads to the same ℓ. Moreover, $\sim$ is independent of the operations PULLed into the local logs in $\mathbf{T}$.

Cases NONDETL, NONDETR, LOOP**:** Let $\mathbf{A}' = \mathbf{A}$ and $\ell' = \ell$. These cases are straight-forward, and simply involve rewinding (with an additional lemma that threads not executing transactions have empty local logs).

Cases SEMI, SEMISKIP**:** (Details omitted because they are uninteresting.)

Case CMT**:** This is the only case in which we map $\mathbf{T}', G'$ to new $\mathbf{A}'$ and ℓ'. We must show that such an $\mathbf{A}', \ell'$ exists such that $\mathbf{T}', G' \sim \mathbf{A}', \ell'$. Let $\mathbf{A}'$ be constructed from each of the transactions in $\mathbf{A}$, except the committing one. The committing one is defined by rewinding the transaction T entirely, and then atomically running the transaction via AM_RUNTX. $(\mathsf{map\ rewind}\ \mathbf{T}') = \mathbf{A}'$ holds by construction.

Consider the committing transaction $T \equiv \{\mathtt{tx}\ c_1, \sigma_1, L_1\}$. The CMT rule gives us that $\mathsf{cmt}(G, L_1, G')$, where G' is the same as G except that the flag gUCmt has been swapped to gCmt for all operations that have been pushed in L_1. $\mathbf{T}'$ is defined by the CMT rule, where the committing transaction has been replaced with $\{\mathtt{skip}, \sigma_1, []\}$.

What remains is to *show the existence an* ℓ' *such that* $\lfloor G' \rfloor_{\mathsf{gCmt}} \preccurlyeq \ell'$, as follows:

- We start by applying the $\mathsf{cmtpres}$ invariant to the configuration $\mathbf{T}, G$.
- We pick the “G that has dropped all uncommitted operations due to other transactions (excluding T):
$$``G \equiv \mathsf{filter}\ (\lambda[\langle m, \sigma, \sigma', id\rangle, g].g = \mathsf{gCmt})\ G'$$
- Now, we do a 0-step local log rollback, keeping L_1 untouched and ‘G = “G.
- The remainder of the $\mathsf{cmtpres}$ invariant tells us that
$$\begin{array}{l}\forall G_{\mathsf{post}}.\ \mathsf{cmt}(`G, L_1, G_{\mathsf{post}}) \Rightarrow \\ \forall \sigma', H.\ (c, \sigma), G_{\mathsf{post}} \cdot \lfloor L_1 \rfloor_{\mathsf{npshd}} \Downarrow \sigma', H \Rightarrow \\ \ \exists \hat{\ell}'.\ \ \mathsf{otx}(\{c, \sigma, L_1\}), `G \setminus L_1 \Downarrow \sigma', \hat{\ell}' \quad \wedge \quad H \preccurlyeq \hat{\ell}'\end{array}$$
- Note that $G_{\mathsf{post}} = \lfloor G' \rfloor_{\mathsf{gCmt}}$ due to a commutativity: G_{post} is obtained from G by dropping all operations due to other uncommitted transactions and then marking T operations as committed. Meanwhile, $\lfloor G' \rfloor_{\mathsf{gCmt}}$ is obtained from G by committing T and then dropping uncommitted operations.
- Note also that $\lfloor L_1 \rfloor_{\mathsf{npshd}} = \varnothing$ because of CMT criterion (ii) and that and $\mathsf{fin}(c)$ due to CMT criterion (i). By the BSFIN rule, $H = G_{\mathsf{post}}$. This leaves us with
$$\exists \hat{\ell}'.\ \ \mathsf{otx}(\{c, \sigma, L_1\}), \lfloor G \rfloor_{\mathsf{gCmt}} \Downarrow \sigma', \hat{\ell}' \quad \wedge \quad H \preccurlyeq \hat{\ell}'$$
- By BSSTEP, we know that $\hat{\ell}' = \lfloor G \rfloor_{\mathsf{gCmt}} \cdot \ell_{new}$. We use this as our successor atomic log in the simulation relation and the right conjunct gives us that $H \preccurlyeq \lfloor G \rfloor_{\mathsf{gCmt}} \cdot \ell_{new}$.
- Note that H contains only committed operations, so $H = \lfloor H \rfloor_{\mathsf{gCmt}}$. Thus, we can conclude that $\lfloor H \rfloor_{\mathsf{gCmt}} \preccurlyeq \lfloor G \rfloor_{\mathsf{gCmt}} \cdot \ell_{new}$ (*i.e.* $\lfloor H \rfloor_{\mathsf{gCmt}} \preccurlyeq \hat{\ell}'$),
- The inductive hypothesis gives us that $\lfloor G \rfloor_{\mathsf{gCmt}} \preccurlyeq \ell$, so $\lfloor H \rfloor_{\mathsf{gCmt}} \preccurlyeq \ell \cdot \ell_{new}$ and thus we have the witness to the step of the atomic machine so the simulation relation still holds. □

6 Evaluation

We applied our model to reason about a wide variety of transactional memory implementations in the literature. In each case, we recast the implementation strategy in terms of the PUSH/PULL model and then show that the implementations satisfy the conditions of each rule in PUSH/PULL.

6.1 Opacity

For general Push/Pull transactions, opacity [10] does not necessarily hold: transactions may view the uncommitted effects of other concurrent transactions. However, there are several ways that we can characterize opacity as a fragment of Push/Pull transactions. For example, if transactions do not perform PULL operations during execution then they are *opaque*.

However, we can take things a step further. An active transaction T may PULL an operation m' that is due to an uncommitted transaction T' provided that T will never execute a method m that does not commute with m. This suggests an interesting way of ensuring opacity while PULLing uncommitted effects by examining (statically or dynamically) the set of all reachable operations that a transaction may perform.

6.2 Optimistic Models

STMs such as TL2 [6], TinySTM [8], Intel STM [31] are optimistic (or mostly-optimistic) and do not share their effects until they commit. Transactions begin by PULLing all operations (there are never uncommitted operations) by simply viewing the shared state. As they continue to execute, they APP locally and do not PUSH until an uninterleaved moment when they check the second PUSH condition on all of their effects (which is approximated via read/write sets) and, if it holds, PUSH everything and CMT. Effects are pushed in order so the first PUSH condition is trivial. If a transaction discovers a conflict, it can simply perform UNAPP repeatedly and needn't UNPUSH.

Transactions that use checkpoints [19] and (closed) nested transactions [27] do not share their effects until commit time. They are similar to the above optimistic models, except that placemarkers are set so that, if an abort is detected, UNAPP only needs to be performed for some operations.

6.3 Pessimistic Models

Matveev and Shavit [25] describe how pessimistic transactions can be implemented by delaying write operations until the commit phase. In this way, write transactions appear to occur instantaneously at the commit point: all write operations are PUSHed just before CMT, with no interleaved transactions. Consequently, read operations perform PULL only on committed effects.

Transactional boosting [11] is also a pessimistic model. It's Push/Pull representation is straightforward.

6.4 Mixed Models

For the irrevocable transactions of Welc *et al.* [34], there is at most one pessimistic ("irrevocable") transaction and many optimistic transactions. The pessimistic transaction PUSHes its effects instantaneously after APP.

6.5 Reading Uncommitted Effects

As discussed in Section 4, the early release mechanism [14] and dependent transactions [30] can be modeled with Push/Pull. In early release, an executing transaction T communicates with T' to determine whether the transactions conflict. This is modeled as T' performing a PUSH(*op*) and T checking whether it is able to PULL(*op*). A *dependent transaction* T will PULL the effects of another transaction T'. This comes with the stipulation that T does not commit until T' has committed. If T' aborts, then T must abort. However, note that T must only move backwards (via $\xrightarrow{\mathsf{back}}$) insofar as to detangle from T'.

7 Boosting/HTM interaction

The Push/Pull model is expressive, permitting transactions to announce their effects in orders different from the way they are done locally (see the PUSH rule). Moreover, transactions can undo their effects in different orders from the order they were announced in (see the UNPUSH rule). This may seem a fairly obscure behavior which, to our knowledge, has not been realized in any transactional implementations. Nonetheless, in this section we demonstrate a simple example where these complex behaviors seem natural.

Consider the following example transaction that accesses a boosted [11] version of a `ConcurrentSkipList` and a boosted version of a `ConcurrentHashTable`, as well as integer variables `size`, `x`, and `y` that are controlled via a hardware transactional memory [17]:

```
BoostedConcurrentSkipList skiplist;
BoostedConcurrentHashTable hashT;
HTM int size;
HTM int x, y;

atomic {
  skiplist.insert(foo);
  size++;

  hashT.map(foo => bar);
  if (*)
    x ++;
  else
    y ++;
}
```

Let us say that execution proceeds, modifying the `skiplist`, incrementing `size`, updating the `hashT`, and the following the `if` branch. At this underlined point when `x` is about to be incremented, let us say that the hardware transactional memory detects a conflict with a concurrent access to `x`.

The PUSH/PULL model shows that the implementation can rewind (UNPUSH) the effects of the HTM, but leave the effects of the boosted objects (which are expensive to replay) in the shared view. So the HTM can discard the effects to `x` and `size` with UNPUSHP, perform a partial rewind via UNAPP, then execute:

```
if (*)
  x ++;
else
  y ++;
}
```

In terms of the PUSH/PULL model, the transaction has performed the rules given in Figure 7. This figure decomposes the elaborate behavior into the simple PUSH/PULL rules. We can then construct a correctness argument for the example from the criteria of each rule, and the hard work of the simulation proof is done for us.

8 Conclusions and Future Work

We have described an expressive model of transactions and shown that it is capable of serving as proof of serializability for a wide variety of transactional memory algorithms. We work with pure logs and develop a model in which transactions pass around their effects by PUSHing to or PULLing from a shared log. The model gives rise to simple proof rules that allow us to more easily construct proofs for a wide range of transactional behaviors—optimism, pessimism, opacity, dependency, etc.—all within a unified treatment.

One potential avenue for future work is to consider weaker notions than serializability [9, 3].

Transaction begins.	PULL(all `skiplist` operations)
	APP(`skiplist.insert(foo)`),
	PUSH(`skiplist.insert(foo)`),
	APP(`size++`),
	PULL(all `hashT` operations)
	APP(`hashT.map(foo=>bar)`),
	PUSH(`hashT.map(foo=>bar)`),
	APP(`x++`),
Push HTM ops:	PUSH(`size++`),
	PUSH(`x++`),
HTM signals abort.	UNPUSH(`x++`),
	UNPUSH(`size++`),
Rewind some code:	UNAPP(`x++`),
March forward again:	APP(`y++`),
Uninterleaved commit:	PUSH(`size++`),
	PUSH(`y++`),
	CMT

Figure 7: Decomposing behavior in terms of PUSH/PULL rules.

References

[1] ABADI, M., BIRRELL, A., HARRIS, T., AND ISARD, M. Semantics of transactional memory and automatic mutual exclusion. In *The 35th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL'08)* (2008), G. C. Necula and P. Wadler, Eds., ACM, pp. 63–74.

[2] BALDASSIN, A., AND BURCKHARDT, S. Lightweight software transactions for games. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)* (2009).

[3] BURCKHARDT, S., AND LEIJEN, D. Semantics of concurrent revisions. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings* (2011), G. Barthe, Ed., vol. 6602 of *Lecture Notes in Computer Science*, Springer, pp. 116–135.

[4] 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, Tucson, AZ, USA, June 7-13, 2008* (2008), R. Gupta and S. P. Amarasinghe, Eds., ACM, pp. 304–315.

[5] COHEN, A., PNUELI, A., AND ZUCK, L. D. Mechanical verification of transactional memories with non-transactional memory accesses. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings* (2008), A. Gupta and S. Malik, Eds., vol. 5123, Springer, pp. 121–134.

[6] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)* (September 2006).

[7] 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).

[8] 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* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 237–246.

[9] FELBER, P., GRAMOLI, V., AND GUERRAOUI, R. Elastic transactions. In *DISC'09* (Berlin, Heidelberg, 2009), pp. 93–107.

[10] 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)* (New York, NY, USA, 2008), ACM, pp. 175–184.

[11] 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).

[12] Herlihy, M., and Koskinen, E. Composable transactional objects: A position paper. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems (ESOP'14)* (2014), vol. 8410, pp. 1–7.

[13] Herlihy, M., Luchangco, V., Moir, M., and III, W. N. S. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003* (2003), E. Borowsky and S. Rajsbaum, Eds., ACM, pp. 92–101.

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

[15] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993* (1993), A. J. Smith, Ed., ACM, pp. 289–300.

[16] IBM. Ibm transactional memory compiler `https://www.ibm.com/developerworks/mydeveloperworks/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/ibm_s_alphaworks_software_transactional_memory_compiler?lang=en`.

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

[18] Kim, D., and Rinard, M. C. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (2011), ACM, pp. 528–541.

[19] Koskinen, E., and Herlihy, M. Checkpoints and continuations instead of nested transactions. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures* (2008), F. Meyer auf der Heide and N. Shavit, Eds., ACM, pp. 160–168.

[20] 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* (2010), M. V. Hermenegildo and J. Palsberg, Eds., ACM, pp. 19–30.

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

[22] Lesani, M., Luchangco, V., and Moir, M. A framework for formally verifying software transactional memory algorithms. In *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings* (2012), M. Koutny and I. Ulidowski, Eds., vol. 7454, Springer, pp. 516–530.

[23] Lesani, M., and Palsberg, J. Communicating memory transactions. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011* (2011), C. Cascaval and P.-C. Yew, Eds., ACM, pp. 157–168.

[24] Lipton, R. J. Reduction: a method of proving properties of parallel programs. *Commun. ACM 18*, 12 (1975), 717–721.

[25] Matveev, A., and Shavit, N. Towards a fully pessimistic stm model. In *Proc. Workshop on transactional memory (TRANSACT12)* (2012).

[26] Moore, K. F., and Grossman, D. High-level small-step operational semantics for transactions. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'08)* (New York, NY, USA, 2008), ACM, pp. 51–62.

[27] 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 Oper. Syst. Rev. 40*, 5 (2006), 359–370.

[28] 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)* (New York, NY, USA, 2007), ACM Press, pp. 68–78.

[29] Papadimitriou, C. The serializability of concurrent database updates. *Journal of the ACM (JACM) 26*, 4 (1979), 631–653.

[30] Ramadan, H. E., Roy, I., Herlihy, M., and Witchel, E. Committing conflicting transactions in an stm. In *PPOPP* (2009), pp. 163–172.

[31] 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 eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'06)* (New York, NY, USA, 2006), ACM, pp. 187–197.

[32] Spear, M. F., Marathe, V. J., Dalessandro, L., and Scott, M. L. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC'07)* (New York, NY, USA, 2007), ACM, pp. 338–339.

[33] Weihl, W. E. Data-dependent concurrency control and recovery (extended abstract). In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC'83)* (New York, NY, USA, 1983), ACM Press, pp. 63–75.

[34] Welc, A., Saha, B., and Adl-Tabatabai, A.-R. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2008), ACM, pp. 285–296.