STO Verification

Jingmei Hu
Harvard University

Xueyuan Han
Harvard University

Hao Bai
Harvard University

Abstract

Software Transactional Object (STO) is a novel, efficient software transactional memory system that leverages data types to avoid false conflicts and reduce bookkeeping effort, significantly improving the performance of existing software transactional memory systems. We formally verify the correctness of STO's commit protocol in Coq, an interactive proof assistant. We use serializability as the correctness condition: we first model the execution of STO transactions as a trace, and then prove that for all STO traces, there exists a serial trace, one that groups all actions of a transaction together and executes one transaction at a time, that produces the same externally-visible effects on the system as its original STO trace, one that have actions of transactions interleave with each other. Although we have made several assumptions and simplifications in our proofs, this work demonstrates the viability to formally machine-prove STO and is the first to do so.

1 Introduction

Transactional memory is an appealing alternative to lock-based synchronization mechanisms. It provides in-memory operations with transaction abstraction analogous to database systems. In order to provide high performance, software transactional memory implementations such as [7, 9, 31] explore different optimization strategies, forcing intricate conflict management schemes. It thus becomes vitally important to verify the correctness of these algorithms.

In this work, we aim to verify in Coq [1] the correctness of a particular software transactional memory system, namely STO [19]. Instead of verifying that its C++ implementation is correct, we verify the algorithm of STO's commit protocol is correct. While opacity [15] is deemed as a better candidate of correctness criterion for transactional memory, for the purpose of this project, we only verify that STO's commit protocol guarantees serializability [11]. We model the execution of STO in Coq with traces, and prove serializability using them. To this end, we prove that all traces generated by the modeling are serializable. In particular, a trace is serializable if there exists an equivalent *serial trace* (section 3) that has the same execution effect as the original trace.

We make the following simplifications in our modeling. First, while the real STO provides many transactional data types that support common data type interfaces (e.g., indexing for arrays, `push_back` for vectors, and insert for red black trees), in our modeling we assume that STO transactions only access one variable with one associated lock, and that this variable can be accessed only by read and write operations. Second, we relax STO's abort conditions by allowing transactions to abort at any point before they decide to commit. This is in contrast to the real STO specification that has strict abort conditions. Third, we do not model STO's bounded spinning in the lock phase of the commit protocol.

We also simplify our proof goal. To prove that the equivalent serial trace of a trace has the same execution effect, normally we need to show that the corresponding read and write operations of the two traces

are the same. In this project, we only show that the equivalent trace preserves all actions and their order in a transaction as the original trace. This property implies that the corresponding read and write operations of the two traces are the same, but imposes less proof burdens.

We introduce a concept of *ghost phase* and an intermediate form of traces called `committed_unconflicted_STO_trace` (`CUST`) to further facilitate our proof strategy.

Ghost phases represent the stage in which a transaction resides during any point of its execution. For example, when a transaction just gets started, it is in a *starting phase*, meaning that it has made no impact on the overall system yet but its existence is known to the system. On the other hand, a transaction in a *commit phase* will globalize its changes to the system by making them externally visible to other transactions. Ghost phases help simplify proof conditions by allowing transactions to only move *forwards* along the phases. That is, once a transaction transitions to a new phase, it cannot return back to its older phases. Although ghost phases are involved in some computation, they do not affect the externally visible state of a machine.

In a nutshell, our proof strategy consists of three phases. In the first phase, we prove that given any STO trace, there exists an equivalent `CUST`. The second phase involves proving that given any `CUST`, there exists an equivalent serial trace. The last phase makes sure that our construction of a serial trace is correct. That is, the resulting serial trace, as we call it, is indeed serial. By transitivity, the first two phases show that there exists a legal transformation that transforms any legal STO trace to an equivalent serial trace, and since the third phase demonstrates that a serial trace generated from such a transformation is indeed serial, we show that a STO trace is serializable.

To the best of our knowledge, our work is the first to verify the correctness of the STO transactional memory using the Coq proof assistant. We make the following contributions in this work: (1) a trace-based approach to modeling the execution of a transactional memory, (2) a machine-checked proof of serializability of a transactional memory. The first contribution also allows us to prove other properties about STO beyond serializability.

The rest of the paper is organized as follows. Section (2) discusses related work. Section (3) introduces our modeling approach. Section (4) details our proof strategy. Section (5) discusses future work, and Section (6) concludes.

2 Background

STO (Software Transactional Objects) is a software transactional memory. STO provides high performance by leveraging semantics of data types. Instead of having a generic commit protocol, STO delegates part of the commit protocol implementation to each specific data type. Being aware of type semantics, each data type can exploit commutativity and other properties to reduce bookkeeping information and false conflicts, thus implementing its part of the commit protocol efficiently. However, we unfortunately do not model all data types in our work. We only model the TBox<int>type and assume all transactions only access one variable of this type.

Serializability is an important property in proving the correctness of transactional memory. It requires that all committed transactions in a schedule issue the same requests and receive the same responses as a serial schedule that consists of precisely the committed transactions in the original schedule. Although serializability is the most commonly used correctness condition for database transactions, it is not sufficient for transactional memory transactions. In particular, it does not specify behavior regarding aborted transactions and aborted transactions can cause irrecoverable errors for transactional memory. Opacity addresses this issue by further requiring that transactions abort as soon as they observe any inconsistent state. While STO does provide opacity, for the sake of simplicity, we only use serializability as the correctness condition.

Our use of the concept of `ghost phase` in the induction type `action phase` (section 3) is inspired by the use of `ghost` in many formal verification literature [18,24,35–37] Unlike most of those systems that use ghost variables/states only for verification, not execution, but similar to Verdi, a framework that formally verifies distributed systems, our system includes ghosts in our execution but does not affect

the externally visible trace.

3 Definitions

Our first step is to prove serializability of a low level implementation of STO that concerns only *one* memory location. We mandate, for ease of constructing proofs, that only two memory-related operations are allowed on the location: *reading* and *writing* a value. We use an inductive type `action` (Appendix A) that describes all possible operations within a transaction. At the same time, we also introduce `action_phase`(Appendix B) that incorporates a *phase variable* of type `nat` to each action, whose main purpose of existence is to simplified later proofs. The latest action of a transaction determines the phase in which the transaction is. However, unlike the common concept of "ghost variables" [24] used in formal verification (section 6), this phase variable is involved in execution, although it does not affect externally visible STO traces. We call this semi-ghost variable *ghost phase*.

Each ghost phase encodes a stage of a transaction in a STO trace. Specifically, when a transaction is in Phase I, it is free to perform reads and writes on the memory location. Once it is ready to commit, it transits to Phase II, during which it also locks its writes if it has performed any writes in the previous phase. Phase III is where a transaction leaves its mark on the trace; it contains a `seq_point` action that records the order of all the transactions in a trace when the trace is serialized. A transaction also validates its reads in this phase after the `seq_point` action. In Phase IV, a transaction uses `commit_txn` action to broadcast its intention to commit, and globalizes its writes using `complete_write_item` action. If every action of the transaction so far proceeds with no conflict with that of other transactions, it declares its completion through the `commit_done_done` action. However, if at any point a conflict arises, the transaction directly transitions to Phase VI and aborts itself via the `abort_txn` action. In addition, if a transaction to be aborted holds a write lock, it must also explicitly release its lock through the `unlock_write_item` action, the only other action in Phase VI. A committed transaction automatically releases its write locks when the commit is completed. We simplify our reasoning and proof burden by allowing a transaction to abort at any time, even when no conflicts have occurred.

In reality, many transactions execute simultaneously, each one having its actions (possibly) interleave with those in other transactions; we define a trace to be a list of actions in all undergoing transactions. When a transaction executes an action, we append the action, along with the ID of the transaction, to a trace. An example trace with four transactions would look like this

$$
\begin{aligned}
[&(4, read\_item1); (3, commit\_done_txn);\\
&(3, commit\_txn); (4, start\_txn)\\
&(3, validate\_read\_item1);\\
&(3, seq\_point); (3, try\_commit\_txn); (3, read\_item1);\\
&(3, start\_txn); (1, abort\_txn); (1, validate\_read\_item1);\\
&(1, try\_commit\_txn); (2, commit\_done\_txn);\\
&(2, complete\_write\_item1);\\
&(2, commit\_txn); (2, validate\_read\_item0);\\
&(2, seq\_point); (2, lock\_write\_item);\\
&(2, try\_commit\_txn); (2, write\_item4);\\
&(1, read\_item0); (2, read\_item0);\\
&(2, start\_txn); (1, start\_txn)]
\end{aligned}
$$

Note that since we *append* actions to the front of the list, the first element in the list is actually the last action of the trace.

When the sequence of actions in a trace follows a set of transactional-memory-specific rules, we call such a trace a *STO trace*. In Coq, we use an inductive type `sto_trace` (Appendix C) to generate legal STO traces, following the rules of software transactional memory STO. We explain what constitutes a legal STO trace by describing the inductive type `sto_trace` in detail in subsection 3.1.

We define a *serial trace* to be a trace that groups together all actions in the transactions that committed successfully. For example, a serial trace of the

above example trace would be

$$[(3, commit\_done_t xn); (3, commit\_txn);$$
$$(3, validate\_read\_item1);$$
$$(3, seq\_point); (3, try\_commit\_txn); (3, read\_item1);$$
$$(3, start\_txn); (2, commit\_done\_txn);$$
$$(2, complete\_write\_item1);$$
$$(2, commit\_txn); (2, validate\_read\_item0);$$
$$(2, seq\_point); (2, lock\_write\_item);$$
$$(2, try\_commit\_txn); (2, write\_item4);$$
$$(2, read\_item0); (2, start\_txn);$$

We can see from the above example that only committed transactions (i.e., transaction 2 and 3) are included in the serial trace. Actions that belong to the aborted transaction (transaction 1) and the unfinished transaction (transaction 4) are all removed from the serial trace.

We introduce an inductive type `committed_unconflicted_sto_trace` Appendix D as an intermediate stage between the original STO trace and its serial trace. `committed_unconflicted_sto_trace` removes aborted and unfinished transactions and contains no read-write conflicts (i.e., it only contains committed transactions). However, it does not require actions in the same transaction to be grouped together (as does a serial trace). We find that having this intermediate stage significantly lessens our proof burden.

We computationally create `serial_traces` from `CUSTs` instead of generating them inductively. To generate a serial trace from a `CUST`, we iteratively swap actions within a trace until no more legal swaps are available. subsection 3.3 describes the mechanism in detail.

3.1 Creating STO Traces

We will explain in this section each action in `sto_trace`. Please refer to Appendix C for the corresponding Coq code.

The `empty_step` is the base case in our inductive type; we construct our STO trace from an empty list.

The `start_txn_step` signifies the inception of a transaction. To create a transaction in a STO trace, one must give a valid `tid` to the transaction. That is, `tid` must be greater than zero and unique. We make sure that the transaction `tid` is unique by checking the phase of a transaction with a given `tid`. We define a function `trace_tid_phase` that returns the ghost phase of a transaction, or 0 if the transaction does not exist. If a `tid` already exists, then its associated transaction must be in a phase larger than I. We find that numerical values of ghost phases makes it easier for us to check the existence of a transaction. Without ghost phases, we have to go through the entire trace to make sure a `tid` is never used.

The `read_item_step` is a `read_item` action. A transaction is only allowed to take this action if it is in Phase I. To be able to read from the memory location, a transaction must ensure that this location is not locked by another transaction ready to commit. We define a function `locked_by` that returns the `tid` of a transaction in a trace that holds a write lock. If no transaction holds a lock, the function will return 0 (recall that all `tid`s must be larger than 0). `read_item` action also records the version number of the memory location. When a transaction is ready to commit, it must validate every version number recorded during its reads against the current version number of the memory location to make sure that the value in that location at the point of the commit is the same value as those at the point of all of its previous reads.

The `write_item_step` is a `write_item` action. A transaction taking this action must be in phase I. We record the value written to the memory location in the trace.

The `try_commit_txn_step` records the time when a transaction is ready to commit; therefore, *after* taking the `try_commit_txn` action, it will transition to Phase II. However, a transaction must be in Phase I before it is ready to commit.

The `lock_write_item_step` is a valid step for read-write transactions. A transaction locks all of its writes (i.e., taking the `lock_write_item` action) as a preparation to commit. No other transaction is able to access the memory location after this action. However, the transaction ready to hold the write locks must make sure that no other transactions have already held the locks. It checks this condition by again using the `lock_by` function. A transaction is only allowed to take this action when it is in Phase

II.

Before a transaction validates its reads, it must takes the `seq_point` action. The location of this *sequence point* in a transaction does not affect the validity of a STO trace. However, this is an important step as it is this action that determines the sequence of all transactions in a trace for creating a *valid* serial trace. Specifically, it is arranged after the `lock_write_item` action but before the `validate_read_item` action (described later). This is because we would like to ensure that in a serial trace that is derived from a valid STO trace, a transaction can only read values that were written by a transaction *before* it in the sequence.0 The `seq_point` action also signifies the transition of a transaction between Phase II and III; therefore, a transaction can take this action only if it is in Phase II.

The `validate_read_item_step` validates the read set of a transaction. Validating reads involves checking the version number associated with the memory location. This version number is determined by the last transaction that completes its writes. We define a function `trace_write_version` to obtain this value.

Once a transaction successfully obtains write locks and validates its read set, the `commit_txn_step` takes place and transitions the transaction from Phase III to Phase IV.

In Phase IV, a read-write transaction can safely globalize its writes as long as it still holds all the write locks. Writing to the memory location also requires the transaction to update the location's version number so a transaction with stale reads will know when it tries to commit that it needs to abort. We use the function `trace_write_version` to obtain the latest version number and increment that value.

The last step for a committed transaction is the `commit_done_step` in Phase IV. A transaction in this step releases its write locks if necessary and finishes.

The `abort_txn_step` occurs as long as a transaction has not committed (i.e., in Phase IV). We allow a transaction to abort at any point before Phase IV for simplicity, even though in practicality, a transaction should not abort when there is no conflict between itself and another transaction. An aborted transaction is considered 'dead'. That is, the traditional roll-back can occur, but it will be considered a new transaction by and of itself.

We also require an explicit step called the `unlock_item_step` to take place after a transaction aborts to release all the locks held by the transaction. Since this action can only occur when a transaction aborts, it must be in Phase VI to take this action. We use `locked_by` function to make sure only read-write transactions take this action since read-only transactions need not do so.

3.2 Creating Committed Unconflicted STO Traces

A committed unconflicted STO trace is a variation of a STO trace with only committed transactions. Since it is a more restricted version of a STO trace and the only requirement is that all transactions must have committed, we can construct a committed unconflicted STO trace on top of a STO trace with one additional condition: all transactions in this trace must be in Phase IV. Using committed unconflicted STO traces, we can discard many irrelevant actions during our proof. For example, since aborted transactions do not affect the final state of the trace, it is safe to remove them and prove properties of the resulting trace. As long as we can prove that no committed transactions are removed or altered during this process, many of those properties can be directly extended to the original STO trace it derives from (section 4).

3.3 Creating Serial Traces

A serial trace contains all committed transactions in a STO trace. Transactions in a serial trace do not interleave; all actions in a transaction are grouped together, and the order of the transactions in a serial trace must lead to the same externally-visible effects as does the original STO trace it derives from. Given a `CUST`, all actions in the `CUST` must also be in the serial trace. Therefore, one only needs to reorder the actions in the trace. Reordering can be performed by swapping adjacent actions multiple times, and stops only when the expected order is achieved. Our algorithm reorders a `CUST` one `tid` at a time, using the position of every transaction's `seq_point` as the

indication of order and grouping all actions belonging to a transaction towards its `seq_point`. We also ensure that adjacent actions in the same transaction are not swapped. After a maximum of `(length t) * (length t)` swaps, where `t` is a `CUST`, we create a serial trace from a `CUST`.

4 Approach

In this section, we discuss how we use our definitions in the previous section to prove correctness of a STO trace (e.g., serializability). We first provide a high level proof strategy (Figure 1), describing capstone theorems we aim to prove in a top-down fashion. We then delve into the details of many proofs, giving lower level explanations.

To prove that a STO trace (say `t`) is serializable, we first prove that the transformation from `t` to a committed unconflicted STO trace (`t'`) is legal, and that the transformation from `t'` to a serial trace (`t''`) is also legal. Given the equivalence of `t` and `t''`, we then only need to show that `t''` is indeed serial.

To prove that `t'` is a legal derivation of `t`, we need to prove three theorems:

- `t'` is also a type `sto_trace`
- `t'` removes all of the non-committed (i.e., aborted and unfinished) transactions in `t`
- `t'` preserves all of the committed transactions in `t`

The first theorem makes sure that the derived trace has the correct type. The second and third theorems ensure that `t'` does not insert random actions in the original trace, and neither does it modify committed transactions.

To prove that `t''` is a legal derivation of `t'`, we need to prove another two theorems:

- `t''` is also a type `CUST`
- `t''` does not reorder actions taken within a transaction in `t'`

The first theorem, combined with the first theorem in the previous list, transitively proves that `t''` is of type `sto_trace`. The second theorem guarantees that for every transaction in `t'`, there exists a corresponding identical transaction in `t''`.

The above five theorems together prove that `t` and `t''` are equivalent in terms of the externally-visible effects on changing the state of a machine. Once we prove in the last theorem that `t''` is indeed serial, we successfully prove that `t` (i.e., a valid `sto_trace`) is serializable.

4.1 Detailed Explanations

Theorem 4.1: A CUST is a STO trace

```
Theorem cust_is_sto_trace t:
committed_unconflicted_sto_trace t
→ sto_trace t.
```

By construction, a `CUST` is a `sto_trace` (Appendix D). Therefore Theorem4.1 can be proved simply by induction on the hypothesis that `t` is a `CUST`.

Theorem 4.2: A CUST removes all uncommitted transactions.

```
Theorem
remove_noncommitted_sto_is_cust t:
sto_trace t →
committed_unconflicted_sto_trace
( remove_noncommitted t (
uncommitted_tids t (tid_list t)
) ).
```

Three functions are involved in Theorem 4.2.

The function `remove_noncommitted` takes two parameters, a trace (i.e., a list of actions) and a list of `tids` of uncommitted transactions, and returns a trace without those transactions.

The function `uncommitted_tids` generates a list of `tids` for `remove_noncommitted` as one of its inputs. It takes a trace and a list of `tids`, and checks whether any of the `tids` in that list corresponds to a uncommitted transaction. That is, if the transaction is not in Phase IV, it includes its `tid` in the returned list.

The function `tid_list` simply returns all `tids` in a trace.

The three functions above are used together to generate a `CUST` from a STO trace, and our goal is to

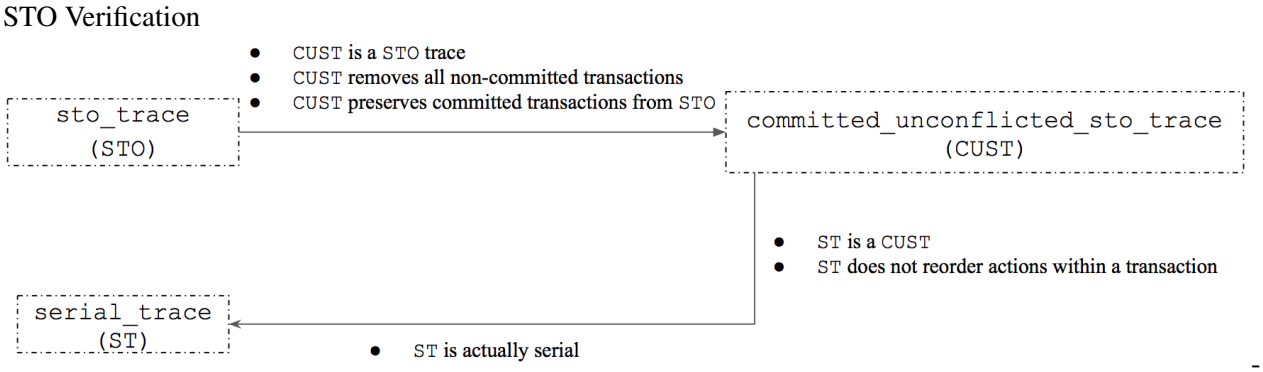

Figure 1: High level proof strategy

prove that the resulting trace is indeed a CUST. Our proof strategy of this theorem contains two phases: 1) We first prove that the trace generated from applying remove_noncommitted function to a STO trace is a STO trace, a weaker argument than the one we have to prove; 2) we then refine our argument by proving two properties of the resulting STO trace: a) All transactions in it are committed transactions, and b) all tids removed from the original STO trace are uncommitted. Following this proof strategy, we successfully prove the veracity of this theorem.

Theorem 4.3: A CUST contains all committed transactions.

```
Theorem
remove_noncommitted_does_not_reorder
t tid:
sto_trace t →
In tid ( uncommitted_tids t
(tid_list t) ) →
filter ( fun pr => fst pr =?  tid
) t =
filter ( fun pr => fst pr =?
tid ) ( remove_noncommitted t (
uncommitted_tids t (tid_list t) )
).
```

Theorem 4.3 proves that a CUST generated from a STO trace not only retains all committed transactions in the STO trace but also ensures that the order of the actions in each transaction is preserved. Combining Theorem 4.1, 4.2, and 4.3, we can prove the equivalence of a STO trace and the CUST that is derived from the STO trace.

We prove this theorem by showing that every time we remove one uncommitted transactions from a STO trace, the rest of the trace is still a valid STO trace, and more importantly, the order of the actions in each transaction remaining in the trace is not changed. We prove this property when we remove each uncommitted transaction until no more transactions can be removed from the trace, which by construction, results in a CUST.

Theorem 4.4: A serial trace is a CUST.

```
Theorem
serial_trace_is_cust t:
committed_unconflicted_sto_trace t
→ committed_unconflicted_sto_trace
( create_serialized_trace t
(seq_list t) ).
```

Theorem 4.4 proves that a serial trace is also a CUST. Combining this theorem with Theorem 4.1, we are able to show that a serial trace is a STO trace. As discussed in section 3, we create a serial trace from a CUST computationally. This is done by calling the function create_serialized_trace. This function takes a trace and an *ordered* list of tids. The order of the tids in the list is determined by the order of the occurrence of seq_point in the committed transactions in the trace. This order is crucial to the correctness of the resulting serial trace. We use the function seq_list to generate this ordered list from the trace. Given this ordered list, create_serialized_trace takes one tid at a time from the list, and calls a function named swaps to group all actions associated with

the tid together by swapping each of them towards the sequence point of the transaction.

To prove that a serial trace generated from applying the create_serialized_trace function to a CUST is also a CUST, we need to prove that legally swapping actions within a CUST does not invalidate its type. This is because ultimately a serial trace is the result of many swaps within the CUST trace. Our mechanism of swapping is legal because each transaction remains valid after being grouped by our swaps, and because we do not reorder the sequence of transactions in a trace.

Note that, although we use the term "serial trace" in this context, we have not yet proved that the trace is indeed serial, and we are not concerned about this property in Theorem 4.4; we have a separate theorem to prove it.

Theorem 4.5: Each transaction in a serial trace has the same order of actions as that in the CUST from which it is derived.

```
Theorem
serial_trace_does_not_reorder t:
committed_unconflicted_sto_trace t
→ forall tid,
filter ( fun pr =>fst pr =?  tid
) t =
filter ( fun pr =>fst pr =?  tid
)
( create_serialized_trace t
(seq_list t) ).
```

In the previous theorem, we prove that the function create_serialized_trace generates a CUST (which should also be a serial trace as we will prove later) from a CUST. However, although we prove in that theorem that each transaction in the resulting trace is valid, we have not yet proved that each transaction is the same in the resulting CUST as the corresponding one in the original CUST. We prove this correctness property in Theorem 4.5.

We construct Theorem 4.5 by checking every transaction in the original CUST and the CUST (or the serial trace) generated from the create_serialized_trace function. The filter function is used to generate a list of actions of one transaction in a trace.

To prove this theorem, we again turn to swaps, just as in Theorem 4.4, and prove that swaps does not modify the order of actions within a transaction. swaps calls a function named swap1 that goes through the entire trace and looks for *one* possible legal swap between an action of a particular tid and some other action of a different tid in the trace.

The conditions of a legal swap1 swap are what guarantee that swaps and thus create_serialized_trace will not reorder actions within a transaction.

Theorem 4.6: A serial trace generated from CUST is indeed serial.

```
Theorem
serial_trace_is_serial t:
committed_unconflicted_sto_trace t
-> is_serial (
create_serialized_trace t
(seq_list t) ).
```

To prove Theorem 4.5, we first provide a definition of *serialization*. We construct this definition as an inductive type (Appendix E). For a CUST to be serial, we only need to make sure that no legitimate swapping exists in the trace. In other words, for all the transactions in the trace, trying to swap any of its actions with other actions in any other transactions results in the same trace as before. This is how we define the inductive type is_serial and how we prove serialization.

5 Discussion and Lessons Learned

Specification Correctness Goes a Long Way

There is one constant theme among all formal verification literature: correctness of the specification of a system cannot be machine-proved. Instead, developers have to manually check its correctness, having confidence in their final specification of the system. Our work is certainly not an exception to this constant theme. Our first version of the specification of STO was unfortunately incorrect, and we did not realize the flaw in the specification until midway into the proofs. Specifically, we made a mistake in identifying the possible positions of the sequence point within a transaction, which is crucial to the correctness of its correspond-

ing serial trace. The consequence of this mistake was rewriting of many definitions and functions and a complete overturn of most of the proofs. Since the specification is the fundamental building block of the formal verification, we learned from our lesson that one cannot spend too much time in checking the correctness of the specification and cannot be too careful in making sure that every aspect of the specification is considered.

Top-Down Approach Promotes Productivity When we first started proving serializability of STO traces, we took a bottom-up approach that causes constant frustration. We defined and proved many helper lemmas, only to find in later development that those lemmas were tangential to our final goal. Although being able to actually prove *something* boosted our confidence in Coq, in reality, we were not making any progress but wasting limited time in blindly proving true, and sometimes trivial statements.

Later in the development, we decided to switch to a top-down approach, starting from the highest-level lemmas and deciding what lower-level lemmas were needed when proving those high-level ones. Starting from the top level specification, which simply states that a STO trace is serializable, we came up with lemmas regarding the order of actions within each transaction, and the sequence of transactions in a serial trace. We then proceeded to even lower level of lemmas concerning swapping actions in a trace. At each level, we *always* completed the proof of each lemma/theorem (i.e., ending the proof with `Qed`, not `Admitted`), although many helper lemmas used in the proof were admitted. Those helper lemmas were usually one level lower in our prove strategy and we would eventually prove those lemmas when we proceed to their level. The benefits of a top-down approach are threefold: 1) We are always able to *complete* proofs of lemmas on a level, which undoubtedly boosts our morale as we achieve a sense of accomplishment and progress; 2) helper lemmas that we need to prove on a lower level are most likely to be useful since we need to use them in higher level lemmas; 3) lemmas that are hard to prove can be constantly refined to smaller lemmas that are easier to prove. Many of our lowest level lemmas can be proved in just a few tactics.

Ghost Provides *Real* Benefits Our use of ghost phases significantly assuaged our proof burden. Conceptually, ghost phases abstract the legal order of actions within a transaction into various stages that are represented by natural numbers. This abstraction gives us two benefits: 1) When dealing with preconditions of an action in a transaction, we no longer need to enumerate a list of actions that must have occurred before it; we only need to make sure that the transaction is in the right phase to take the action; 2) since phases are represented by natural numbers, we are able to use all lemmas regarding natural numbers; reasoning about numbers is intuitive and error-proof than reasoning about each action within a transaction.

Layers of Abstraction Remain Valuable A famous aphorism of David Wheeler goes:

> All problems in computer science can be solved by another level of indirection.

Our addition of a level of indirection, namely the `committed_unconflicted_sto_trace` layer, helps facilitate our reasoning and simplify proof requirements. Without this layer, proving a STO trace is serial requires us to consider multiple aspects of the trace simultaneously. For example, when constructing any lemmas regarding a STO trace, we need to be concerned about aborted and unfinished transactions, in addition to the useful committed transactions. Performing induction on such a general trace can be challenging, if not impossible sometimes. The addition of the `CUST` layer separates our concern of committed transactions in a STO trace, which are what actually constitutes the serial trace, from that of aborted and unfinished transactions, which are eventually removed from the serial trace. Therefore, `CUST` enables us to only reason about committed transactions when we prove serialization of a STO trace.

Decomposition Leads to Simplification A complex algorithm in Coq usually leads to a painful proof when it is used in a lemma. We experienced such pain when we tried to prove properties about a poorly-written swapping function. Our original `swap1` function takes a path along the trace, and swaps actions of a given `tid` with its neighboring action as long as the swap is legal. Therefore, unlike our current `swap1` function, which performs at most one swap, it can potentially swap many times until it

reaches the end of a trace. This fact restricts us from only focusing on two adjacent actions; we cannot assume that the rest of the trace is unchanged. On the other hand, we can call one-swap `swap1` function multiple times and obtain the same result as that of calling multi-swap `swap1` function. Reasoning about one swap at a time localizes our concern and makes the proof manageable; we need not worry about the whole trace but the two adjacent actions being swapped. Our decomposition of the `swap1` function (not to be confused with our `swaps` function that calls `swap1` multiple times) from multiple swaps at a time to one swap at a time enabled us to successfully prove Theorem 4.4 and 4.5.

6 Related Work

Verification of software transactional memories has received much attention in recent years. The works can be divided into two classes of approaches in general: static and dynamic approaches.

6.1 Static Approaches

Research in this direction, including our work, has mostly focused on verification of transactional memories at the algorithm level, instead of the implementation level (e.g. C/C++ implementation). Early works take the model checking approach [4, 10, 12–14, 26]. This line of works aim at automatic verification of transactional memories. In principle, specifications of correctness conditions and transactional memory algorithms (TM algorithms) can be formalized as transition systems, state exploration techniques can then be applied to verify that the TM algorithms satisfy the correctness conditions. However, since TM algorithms generally have an infinite number of states due to unbounded numbers of threads and shared memory locations and unbounded transaction lengths, simple model checking approaches cannot be directly used.

Cohen et al. [4] can verify small instances (i.e. an instantiation of the TM algorithm that has a small number of threads and each transaction contains a small number of operations) of some simple TM algorithms directly using a model checker. But this approach does not apply to larger instances or more complicated TM algorithms. They manually specify the refinement relations between the specifications and the TM algorithms. Then they verify the relations with an explicit state model checker. This approach requires a thorough understanding of both the specification and the TM algorithm, and does not really achieve automation.

Guerraoui et al. [12–14] address the state explosion problem by reducing the verification to a small instance that has only two threads and two locations. And they prove a meta-theorem which states that a TM algorithm is correct if and only if the small instance is correct. The first component can be solved by finite state model checking, but the second component requires manual proof for each individual TM algorithm. This means that the proof is not fully machine checked either.

Emmi et al. [10] attempt to achieve fully automatic parameterized verification by automatically generating and checking parameterized invariants. They generate invariants using a combination of verification by invisible invariants [2, 28] and template-based invariant generation [5, 33]. These techniques exploit structural properties of of TM algorithms. However, their approach does not scale well for instances that have more than two threads and two memory locations.

O'Leary et al. [26] work on verifying an industrial implementation of software transactional memory, namely McRT STM [30]. Unlike previous works that model check transactional memories at the algorithm level, they attempt to model the STM pseudocode as exactly as possible. For example, they model pointer dereferencing and setjmp/longjmp operations. Initially, they use strict serializability as the correctness condition, but the model checker shows that their implementation does not respect real-time order. Eventually, they manage to validate that McRT STM guarantees serializability of purely-transactional programs that have two threads and each transaction has at most three reads or writes.

Overall, while model checking approaches can be used to find bugs, they do not scale well for verification of TM algorithms in the general case. This is not necessarily a problem if it can be shown that a TM algorithm is correct if and only if some smaller instance is correct. Unfortunately, this prop-

erty does not hold for all TM algorithms.

Our work most relates to research that verifies the correctness of TM algorithms via theorem proving [6, 20, 34]. An interactive theorem prover addresses the state explosion problem by allowing human insights to guide the proof, at the cost of significant more human efforts. Our work differs from other works mostly in that we use the Coq proof assistant, and that we are currently using a more specific solution that targets to verify the STO transactional memory.

In a subsequent work, Cohen et al. [6] extend their TM model to handle non-transactional memory accesses. Instead of using a model checker, they opt for an interactive theorem prover, TLPVS [27] (a variant of PVS [29]), to overcome the state explosion problem. They prove that the TCC algorithm [16] augmented with non-transactional memory accesses provides strict serializability.

Lesani et al.'s work [20] resemble the one above in that they both use PVS. They build a framework for verifying TM algorithms in the PVS interactive theorem prover. They use I/O automata [23] to specify both correctness conditions and TM algorithms, and prove that the automaton specifying the TM algorithm implements the automaton specifying the correctness condition by simulation [22]. A common proof strategy in their proof is to introduce intermediate automata that successively have more details about the target TM algorithm. This hierarchical approach not only breaks the proof into manageable pieces, but also allows reuse of the same proof for TM algorithms that share common specifications at some level. This framework provides many theories and lemmas to facilitate verification of new TM algorithms. They propose a new correctness condition for TM algorithms "TMS1" that aims to be as general as possible, and a more restrictive version "TMS2" that is easier to work with [8]. Using their framework, they have proved that TMS2 implements opacity, and opacity implements TMS1. Therefore, to prove that a TM algorithm implements TMS1, it suffices to prove that it implements TMS2. And they have proved that the NOrec algorithm implements TMS2, using the hierarchical approach mentioned above.

Tasiran [34] takes a different approach. He decomposes the verification into two parts. The first part is a manual proof that a TM algorithm provides serializability if certain non-interference properties hold. The second part consists of proving that the Barok STM [17] satisfies these properties. Since the non-interference properties are expressed as assertions in sequential programs, he is able to use the Boogie sequential program verifier to handle all proofs.

6.2 Dynamic Approaches

Due to the limitations and complexity of static verification, researchers also seek runtime verification techniques [3, 21, 25, 32] that verify the correctness of executions of transactional memories.

Manovit et al. [25] use pseudo-random testing to verify and find bugs in transactional memory implementations. They generate random test programs with both transactional and non-transactional operations, run the programs, and feed the traces (including dynamic order of all operations, and load and store values) of programs to an analysis algorithm, which decides if a particular trace can be generated by a legal execution. They show that the general problem of deciding if an trace is legal is NP-complete. However, with extensive optimizations, their analysis algorithm works well in practice. This approach is more suited for offline verification of transactional memory implementations. Similarly, Hu and Hutton [21] implement a simple transactional language similar to STM Haskell. They also implement the semantics, run time and compiler in Haskell, and use randomized testing to verify their compiler is correct.

Chen et al. [3] propose to perform online runtime validation. They construct a constrained graph online that captures the correctness of transactional memories, and check the graph for cycles. A cycle indicates that the correctness guarantee is violated. They use a check-pointing scheme to resume the execution from a previously validated state if a cycle is detected. To make the runtime validation practical, they use several optimization techniques to reduce the number of graph vertices to track and limit the scope of checking.

Singh's work [32] on online runtime verification resembles that of Chen et al. They construct conflict graphs that can encode various correctness

conditions (e.g. serializability, strict serializability, opacity) and check the graph is acyclic. They achieve good performance with a two level verification scheme. The coarse verification uses Bloom filters for fast storage of tracking sets. Since the results of using Bloom filters are inaccurate, they run a precise verification to check the correctness of an execution if the coarse verification reports a cycle.

7 Future Work

Our work is the first step towards proving correctness of a software transactional memory system, STO. However, as we have discussed in section 1, we have made numerous assumptions and simplifications. For example, the most exciting part of STO is its type-aware commit protocol that can effectively reduce false conflicts and bookkeeping effort. However, our work concerns only one memory location and thus does not take data types into consideration. Moreover, a transaction can abort at any time. Therefore, future work involves, for example, proving the correctness of the algorithms of the transactional memory that hosts a specific data type (e.g., a red-black tree or a linked list). We believe one can reuse many of our definitions, lemmas, and theorems to prove such correctness. For instance, one only needs to slightly modify the inductive type `action` to include operations such as checking whether aborting is needed even when there is an invalid read.

8 Conclusion

We have proved correctness (in terms of serializability) of the algorithm of a software transactional memory system, STO. We have made several assumptions and simplifications of STO. Specifically, we are only concerned about one memory location with only one lock. Moreover, we allows a transaction to abort at any time and do not model STO's bounded spinning in the lock phase of the commit protocol. However, to the best of our knowledge, our work is the first that demonstrates the correctness of the STO protocol, and provides a meaningful guidance to those who are interested in formally verify STO.

Acknowledgement

We thank Eddie Kohler for answering numerous questions about the project and assisting us on Coq development. His idea of ghost phases is proved to be extremely useful.

References

[1] Coq. `https://coq.inria.fr`.

[2] ARONS, T., PNUELI, A., RUAH, S., XU, J., AND ZUCK, L. D. Parameterized verification with automatically computed inductive assertions. In *Proceedings of the 13th International Conference on Computer Aided Verification* (London, UK, UK, 2001), CAV '01, Springer-Verlag, pp. 221–234.

[3] CHEN, K., MALIK, S., AND PATRA, P. Runtime validation of transactional memory systems. In *9th International Symposium on Quality Electronic Design (isqed 2008)* (March 2008), pp. 750–756.

[4] COHEN, A., O'LEARY, J. W., PNUELI, A., TUTTLE, M. R., AND ZUCK, L. D. Verifying correctness of transactional memories. In *Formal Methods in Computer Aided Design, 2007. FMCAD '07* (Nov 2007), pp. 37–44.

[5] COHEN, A., PNUELI, A., AND ZUCK, L. D. *Mechanical Verification of Transactional Memories with Non-transactional Memory Accesses*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 121–134.

[6] COHEN, A., PNUELI, A., AND ZUCK, L. D. *Mechanical Verification of Transactional Memories with Non-transactional Memory Accesses*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 121–134.

[7] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking ii. In *International Symposium on Distributed Computing* (2006), Springer, pp. 194–208.

[8] DOHERTY, S., GROVES, L., LUCHANGCO, V., AND MOIR, M. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing 25*, 5 (2013), 769–799.

[9] DRAGOJEVIĆ, A., GUERRAOUI, R., AND KAPALKA, M. Stretching transactional memory. In *ACM sigplan notices* (2009), vol. 44, ACM, pp. 155–165.

[10] EMMI, M., MAJUMDAR, R., AND MANEVICH, R. Parameterized verification of transactional memories. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 134–145.

[11] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.

[12] GUERRAOUI, R., HENZINGER, T. A., JOBSTMANN, B., AND SINGH, V. Model checking transactional memories.

In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 372–382.

[13] GUERRAOUI, R., HENZINGER, T. A., AND SINGH, V. Completeness and nondeterminism in model checking transactional memories. In *Proceedings of the 19th International Conference on Concurrency Theory* (Berlin, Heidelberg, 2008), CONCUR '08, Springer-Verlag, pp. 21–35.

[14] GUERRAOUI, R., HENZINGER, T. A., AND SINGH, V. Software transactional memory on relaxed memory models. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2009), CAV '09, Springer-Verlag, pp. 321–336.

[15] 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* (2008), ACM, pp. 175–184.

[16] HAMMOND, L., WONG, V., CHEN, M., CARLSTROM, B. D., DAVIS, J. D., HERTZBERG, B., PRABHU, M. K., WIJAYA, H., KOZYRAKIS, C., AND OLUKOTUN, K. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2004), ISCA '04, IEEE Computer Society, pp. 102–.

[17] HARRIS, T., PLESKO, M., SHINNAR, A., AND TARDITI, D. Optimizing memory transactions. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 14–25.

[18] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 1–17.

[19] HERMAN, N., INALA, J. P., HUANG, Y., TSAI, L., KOHLER, E., LISKOV, B., AND SHRIRA, L. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 31.

[20] LESANI, M., LUCHANGCO, V., AND MOIR, M. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory* (Berlin, Heidelberg, 2012), CONCUR'12, Springer-Verlag, pp. 516–530.

[21] LIYANG HU, G. H. Towards a verified implementation of software transactional memory. In *Symposium on Trends in Functional Programming* (May 2008).

[22] LYNCH, N., AND VAANDRAGER, F. Forward and backward simulations i.: Untimed systems. *Inf. Comput. 121*, 2 (Sept. 1995), 214–233.

[23] LYNCH, N. A., AND TUTTLE, M. R. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1987), PODC '87, ACM, pp. 137–151.

[24] MAI, H., PEK, E., XUE, H., KING, S. T., AND MADHUSUDAN, P. Verifying security invariants in expressos. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 293–304.

[25] MANOVIT, C., HANGAL, S., CHAFI, H., MCDONALD, A., KOZYRAKIS, C., AND OLUKOTUN, K. Testing implementations of transactional memory. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2006), PACT '06, ACM, pp. 134–143.

[26] O'LEARY, J., SAHA, B., AND TUTTLE, M. R. Model checking transactional memory with spin. In *2009 29th IEEE International Conference on Distributed Computing Systems* (June 2009), pp. 335–342.

[27] PNUELI, A., AND ARONS, T. *tlpvs: A pvs-Based ltl Verification System*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 598–625.

[28] PNUELI, A., RUAH, S., AND ZUCK, L. D. Automatic deductive verification with invisible invariants. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (London, UK, UK, 2001), TACAS 2001, Springer-Verlag, pp. 82–97.

[29] S. OWRE, N. SHANKAR, J. R., AND STRINGER-CALVERT, D. Pvs system guide. Tech. rep., Computer Science Laboratory, SRI International, Menlo Park, CA, 1999.

[30] 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* (New York, NY, USA, 2006), PPoPP '06, ACM, pp. 187–197.

[31] SHAVIT, N., AND TOUITOU, D. Software transactional memory. *Distributed Computing 10*, 2 (1997), 99–116.

[32] SINGH, V. Runtime verification for software transactional memories. In *Proceedings of the First International Conference on Runtime Verification* (Berlin, Heidelberg, 2010), RV'10, Springer-Verlag, pp. 421–435.

[33] SRIVASTAVA, S., AND GULWANI, S. Program verification using templates over predicate abstraction. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 223–234.

[34] TASIRAN, S. A compositional method for verifying software transactional memory implementations. Tech. rep., Koc University and Microsoft Research, 2008.

[35] TASSAROTTI, J., DREYER, D., AND VAFEIADIS, V. Verifying read-copy-update in a logic for weak memory. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 110–120.

[36] TURON, A., VAFEIADIS, V., AND DREYER, D. Gps: Navigating weak memory with ghosts, protocols, and separation. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 691–707.

[37] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 357–368.

A Inductive Type `action`

```
Inductive action:=
|start_txn: action
|read_item: version -> action
|write_item: value -> action
|try_commit_txn: action
|lock_write_item: action
|seq_point: action
|validate_read_item: version ->
   action
|abort_txn: action
|unlock_write_item: action
|commit_txn: action
|complete_write_item: version ->
   action
|commit_done_txn: action.
```

B Definition `action_phase`

```
Definition action_phase (a:action)
    :=
  match a with
  | start_txn => 1
  | read_item _ => 1
  | write_item _ => 1
  | try_commit_txn => 2
  | lock_write_item => 2
  | seq_point => 3
  | validate_read_item _ => 3
  | commit_txn => 4
  | complete_write_item _ => 4
  | commit_done_txn => 4
  | abort_txn => 6
  | unlock_write_item => 6
  end.
```

C Inductive Type `sto_trace`

```
Inductive sto_trace : trace ->
   Prop :=
| empty_step : sto_trace []
| start_txn_step: forall t tid,
    tid > 0
    -> trace_tid_phase tid t = 0
    -> sto_trace t
    -> sto_trace ((tid, start_txn)
       ::t)
| read_item_step: forall t tid,
    trace_tid_phase tid t = 1
    -> locked_by t 0 = 0
    -> sto_trace t
    -> sto_trace ((tid, read_item
       (trace_write_version t)) ::
        t)
| write_item_step: forall t tid
   val,
    trace_tid_phase tid t = 1
    -> sto_trace t
    -> sto_trace ((tid, write_item
        val) :: t)
| try_commit_txn_step: forall t
   tid,
    trace_tid_phase tid t = 1
    -> sto_trace t
    -> sto_trace ((tid,
       try_commit_txn)::t)
| lock_write_item_step: forall t
   tid v,
    trace_tid_phase tid t = 2
    -> In (tid, write_item v) t
    -> locked_by t 0 = 0
    -> sto_trace t
    -> sto_trace ((tid,
       lock_write_item) :: t)
| seq_point_step: forall t tid,
    trace_tid_phase tid t = 2
    -> (forall v, In (tid,
       write_item v) t
                -> In (tid,
                   lock_write_item
                   ) t)
    -> sto_trace t
    -> sto_trace ((tid, seq_point)
        :: t)
| validate_read_item_step: forall
   t tid vers,
```

```
    trace_tid_phase tid t = 3
    -> locked_by t tid = tid
    -> trace_write_version t =
       vers
    -> sto_trace t
    -> sto_trace ((tid,
       validate_read_item vers) ::
        t)
| abort_txn_step: forall t tid,
    trace_tid_phase tid t > 0
    -> trace_tid_phase tid t < 4
    -> sto_trace t
    -> sto_trace ((tid, abort_txn)
        :: t)
| unlock_item_step: forall t tid,
    trace_tid_phase tid t = 6
    -> locked_by t 0 = tid
    -> sto_trace t
    -> sto_trace ((tid,
       unlock_write_item) :: t)
| commit_txn_step: forall t tid,
    trace_tid_phase tid t = 3
    -> (forall vers, In (tid,
       read_item vers) t
                    -> In (tid,
                       validate_read_item
                        vers) t)
    -> (forall vers, In (tid,
       validate_read_item vers) t
                    -> In (tid,
                       read_item
                       vers) t)
    -> sto_trace t
    -> sto_trace ((tid, commit_txn
       ) :: t)
| complete_write_item_step: forall
    t tid val,
    trace_tid_phase tid t = 4
    -> locked_by t 0 = tid
    -> trace_tid_last_write tid t
       = Some val
    -> sto_trace t
    -> sto_trace ((tid,
       complete_write_item (S (
       trace_write_version t))) ::
        t)
| commit_done_step: forall t tid,
    trace_tid_phase tid t = 4
    -> locked_by t 0 <> tid
    -> sto_trace t
    -> sto_trace ((tid,
       commit_done_txn) :: t).
```

D Inductive Type committed_unconflicted_sto_trace

```
Inductive
   committed_unconflicted_sto_trace
    : trace -> Prop :=
| construct_cust: forall tr,
  sto_trace tr
  -> (forall tid, trace_tid_phase
     tid tr > 0 -> trace_tid_phase
      tid tr = 4)
  ->
     committed_unconflicted_sto_trace
      tr.
```

E Inductive Type is_serial

```
Inductive is_serial: trace -> Prop
    :=
  | serial_constructor: forall t,
      committed_unconflicted_sto_trace
          t
      -> (forall tid, In tid (
         seq_list t) -> swap1 tid
         t = t)
      -> is_serial t.
```