Correctness without Serializability: Verifying Transactional Programs under Snapshot Isolation

Ismail Kuru
Koç University, Istanbul
ikuru@ku.edu.tr

Burcu Kulahcioglu Ozkan
Koç University, Istanbul
bkulahcioglu@ku.edu.tr

Suha Orhun Mutluergil
Koç University, Istanbul
bkulahcioglu@ku.edu.tr

Serdar Tasiran
Koç University, Istanbul
stasiran@ku.edu.tr

Tayfun Elmas
University of California, Berkeley
elmas@cs.berkeley.edu

Ernie Cohen
Microsoft
ernie.cohen@acm.org

Abstract

We present a static verification approach for programs running under snapshot isolation (SI) and similar relaxed transactional semantics. In a common pattern in distributed and concurrent programs, transactions each read a large portion of shared data, perform local computation, and then modify a small portion of the shared data. Requiring conflict serializability in this scenario results in serial execution of transactions or worse, and performance suffers. To avoid such performance problems, relaxed conflict detection schemes such as snapshot isolation (SI) are used widely. Under SI, transactions are no longer guaranteed to be serializable, and the simplicity of reasoning sequentially within a transaction is lost. In this paper, we present an approach for statically verifying properties of transactional programs operating under SI. Differently from earlier work, we handle transactional programs even when they are designed not to be serializable.

In our approach, the user first verifies his program in the static verification tool VCC pretending that transactions run sequentially. This task requires the user to provide program annotations such as loop invariants and function pre- and post-conditions. We then apply a source-to-source transformation which augments the program with an encoding of the SI semantics. Verifying the resulting program with transformed user annotations and specifications is equivalent to verifying the original transactional program running under SI – a fact we prove formally. Our encoding preserves the modularity and scalability of VCC's verification approach. We applied our method successfully to benchmark programs from the transactional memory literature. In each benchmark, we were able to verify the encoded program without manually providing any extra annotations beyond those required for verifying the program sequentially. The correctness argument of the sequential versions generalized to SI, and verification times were similar.

1. Introduction

Database and in-memory transactions provide a convenient, composable mechanism for writing concurrent and distributed programs. When an execution platform provides serializable transactions, the code of a transaction can be treated as sequential code, which significantly simplifies writing and verifying applications.

Serializability is typically ensured by enforcing conflict serializability [1] among transactions. The execution platform keeps track of variables read or updated (the "read" and "write sets") by each transaction. A transaction experiences a conflict if, while it is in progress, a variable in its read or write sets is updated by a concurrent transaction. Conflict serializability requires transactions experiencing conflicts to be aborted. This approach results in very poor performance for some common concurrent programming patterns.

For instance, as is the case in a common parallel programming pattern, if transactions read a large portion of the shared data, perform local computation and update only a small portion of the shared data, almost all concurrent transactions conflict. Conflict serializability then amounts to serial execution or even worse, repeated aborts and re-tries. An example of this pattern can be found in the `FindRoute` operation (Figure 1) from the Labyrinth program in the STAMP benchmark suite [2]. Here, transactions compute Manhattan-style paths connecting pairs of points in a three-dimensional grid. The program must maintain the invariant that paths (representing wires) do not overlap. Each transaction running `FindRoute` takes a local snapshot of the grid, computes a path using the local snapshot, and concludes by registering the path `onePath` onto the grid only if it does not overlap paths in the grid at that point in time (not on the possibly stale snapshot). Since all transactions read the entire grid, the updates of each transaction conflict with reads of all other concurrent transactions. Conflict serializability results in unacceptable performance in this example.

```
// Program invariant: \forall int i; 0<=i && i< pathlist->num_paths ==> isValidPath(grid, pathsList->paths[i])

FindRoute(p1, p2) {
 transaction {
 1:    localGridSnapshot = grid; // Local copy made of entire grid
 2:
 3:    // Local, possibly long, computation
 4:    onePath = shortest_path(p1, p2, localGridSnapshot);
 5:    // Desired post-conditions of shortest_path:
 6:    assert(isValidPath(onePath, localGridSnapshot))
 7:    assert(isConnectingPath(onePath, p1, p2);
 8:
 9:    // Register points on onePath as "taken" on grid
10:    // Add onePath to pathsList
11:    gridAddPathIfOK(grid, pathsList, onePath);
12:
13:    // FindRoute must ensure program invariants, and the post-condition
14:    // onePath \in pathsList && IsConnectingPath(onePath, p1, p2)
} }
```

Figure 1. Outline for `FindRoute` code and specification.

To avoid performance problems, in practice, many transactional platforms such as databases or transactional memory implementations carry out more relaxed conflict detection. For instance, in the widely used snapshot isolation (SI) consistency model, all read accesses of a transaction must appear atomic, and all updates must appear atomic, but the entire transaction (i.e., the reads and updates together) need not. This allows a transaction to complete successfully even when global state is updated while the transaction is in progress. In `FindSlot` operating under snapshot isolation, updates to the grid by concurrent transactions do not result in an execution of `FindRoute` to abort unless they overlap the path it computed, `onePath`. Relaxed conflict detection schemes other than SI, such as programmer-defined conflict detection [3], and early release of read set entries [4] have been investigated in the database, software and hardware transactional memory communities. In this paper, we investigate some of these models as well, but in discussions about relaxed conflict detection, we use SI for brevity to refer to relaxed consistency models similar to SI.

Much work on SI has focused on verifying or ensuring that transactions remain conflict-serializable even under SI by, e.g., verifying or ensuring that no "write-skew anomaly" exists. However, as the benchmarks we studied show, it is quite natural to write transactional programs that are correct but contain write-skew anomalies and are not conflict- or view-serializable, . To see that executions of `FindRoute` may not be conflict- or view-serializable, consider the following interleaving of accesses illustrating a scenario where two concurrent executions of `FindRoute` both succeed.

```
    T1                          T2
-----------                 ---------
Take snapshot
Compute path
                            Take snapshot
                            Compute path
Update grid and pathList
                            Update grid and pathList
```

Intuitively, both of these transactions are allowed to succeed because they update disjoint sets of points on the grid. In this execution, `T1` and `T2`'s snapshots of the grid are identical. In a serial execution, for two succeeding transactions, this could never be the case, as the second one to execute would see the updates of the first one. Therefore, these two transactions are not view- or conflict-serializable. In fact, these transactions are not serializable at all, since the path-computation algorithm uses heuristics trying to minimize the number of bends in a path. The path computed by `T2` in the schedule above will be potentially different from paths computed in a serial schedule in which `T1` precedes `T2`. To see why `FindRoute` is correct under SI despite the schedule above, observe that when `FindRoute` attempts to update the grid, it is successful only if `onePath` does not overlap any other paths at commit time. Thus, despite executions not being serializable, `FindRoute` is able to maintain the correctness invariants, i.e., that paths on the grid do not overlap each other.

For such programs, under SI and similar relaxed conflict detection, one obtains performance close to fine-grain locking and other custom synchronization. Unfortunately, the interleaving of accesses allowed by SI means that transactions experience interference and the code for a transaction cannot be treated as if it were sequential. As a result, confidence in the correctness of the application is lost – in papers (e.g., [3]) only informal arguments are made about the correctness of applications under SI. In this paper, we present a method for proving that programs running under SI satisfy their invariants, assertions, transaction pre- and post-conditions, despite the fact that their executions are not serializable.

Static tools targeted at verifying sequential programs [5–7], and the VCC verification tool [8] for verifying concurrent C programs have been quite successful. These tools are (when applicable) thread, function and object-modular, and scale well to large programs. However, they are not aware of transactional and/or SI-like semantics, and cannot be used directly for our purposes. In this paper, we present a verification approach for transactional applications running under SI by transforming the problem to an equivalent one that can be handled by VCC, a verifier for generic C programs.

In order to achieve this transformation, we provide an encoding scheme that, given a transactional program P running under SI, produces a regular C program $\widetilde{P}$ such that verifying properties of $\widetilde{P}$ is equivalent to verifying properties of P running under SI. We prove this latter fact about the correctness of the construction of $\widetilde{P}$ formally.

Our approach for verifying a program P running under SI consists of the following steps:

1. The user starts with P and verifies it assuming transactions are executed sequentially. Desired data structure invariants, assertions, and function pre- and post-conditions are verified in VCC in a modular fashion. The user may need to provide further program annotations such as invariants, assertions, function pre- and post-conditions to facilitate this straightforward sequential verification task.
2. The program P is augmented with a very high-level, abstract implementation of transactional and SI semantics to obtain $\widetilde{P}$.
3. The user then verifies properties on $\widetilde{P}$ using VCC. This last step checks two facts:
   - The annotations on the sequential program P continue to hold in $\widetilde{P}$, equivalently, when P is run under SI, and
   - these annotations are sufficient to verify application correctness for $\widetilde{P}$.

In the benchmarks we studied, we started with a program P interpreted sequentially, and while carrying out verification with VCC, made an effort to provide loose annotations, e.g., made loop invariants and function post-conditions only strong enough to discharge the invariants required for correctness. We found that, after this, it was sufficient to run VCC on $\widetilde{P}$ without providing any extra annotations, i.e., the correctness argument for the sequentially-interpreted program generalized to the concurrent version running under SI. For example in which this may not be the case, users can experiment with different annotations on P. VCC's counterexample viewing facility helps the user determine whether the example does not remain correct under SI, or, otherwise, to arrive at sufficiently relaxed annotations on P so that they continue to hold under SI.

In our source-to-source transformation, we make use of auxiliary variables such version numbers and transaction-local copies of variables. We also establish and make use of ownership and "approval" relationships among "objects" (structs) and in order to represent exactly the set of interleavings that SI allows. While this encoding is straightforward, it is designed with special attention towards preserving the thread, function and object modularity of the verification of the sequential version of the program in VCC. In particular, the encoding avoids inlining code that other, interfering transactions that might be running concurrently. As a result, in the benchmarks we studied, verification times for code interpreted sequentially and code running under SI were close to each other. While we have not yet automated this transformation, it is automatable, and, as our experiments with `!WAR` relaxations show, the transformation scheme is easily adaptable to similar relaxed conflict detections schemes.

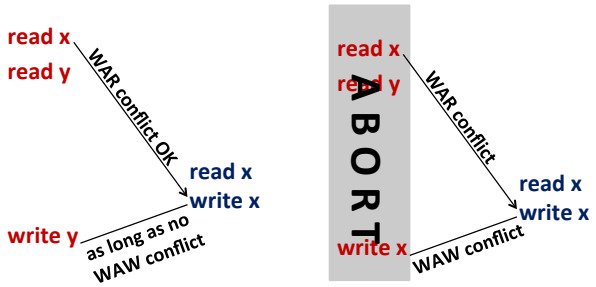

Figure 2. Aborted and successful transactions under `!WAR conflict detection`.

We verified three benchmarks from the STAMP [2] transactional benchmarks suite and a `StringBuffer` pool example that we wrote running under SI and the write-after-read conflict relaxation schemes. Each of these verification tasks took on the order of 30 seconds on a laptop computer. These examples are discussed in detail in the paper, and all materials required to reproduce the verification effort is available at `http://msrc.ku.edu.tr/projects/vcctm`

2. Correctness without Serializability: Case Studies

Our encoding of SI semantics makes it possible to generalize the correctness argument for a program under sequential interpretation to a correctness argument running under SI. In this section, we introduce some of our case studies and the intuition for their correct operation under SI in order to provide concrete motivation for our approach.

2.1 Labyrinth under Snapshot Isolation

As shown in 1, in the Labyrinth benchmark, each concurrent transaction runs an instance of the function `FindRoute` to route a wire "Manhattan-style" in a three-dimensional grid (`globalGrid`) from point `p1` to point `p2`. Wires are represented as paths: lists of points with integer x, y, and z coordinates, where consecutive entries in the list must be adjacent in the grid. The grid is represented as a three-dimensional array, where each entry `[i,j,k]` is either the ID of the path going through the grid point with coordinates `[i,j,k]` if one exists, or `GRID_EMPTY` otherwise. A data structure `pathList` keeps pointers to all paths in an array.

Each execution of `FindRoute(p1,p2)` first takes a snapshot of the grid (line 1) and then performs local computation using this local snapshot to compute a path (`onePath`, line 4) from `p1` to `p2`. Observe that, during this local computation, other executions of `FindRoute` may complete and

modify the grid. In other words, `localGridSnapshot` may be stale snapshot of `grid`. SI guarantees in this example that (i) the read of the entire grid in line 4 is atomic, (ii) that the updates to `onePath` and `globalGrid` in line 11 are atomic.

Formally, a path `path` that is in `pathList` and the grid `grid` must satisfy the following invariant.

```
\bool isValidPath(int ***grid, path_t* path) =
  (\forall int i; 0<= i < path->path_len ==>
     path->ID == grid[path->xs[i]]
                     [path->ys[i]]
                     [path->zs[i]])
   \forall int i; 0<= i < path->path_len-1 ==>
     isAdjacent(path->xs[i], path->xs[i+1],
                path->ys[i], path->ys[i+1],
                path->zs[i], path->zs[i+1]))
   ));
```

This invariant formally specifies two important properties. First, the information contained in the `grid` and each path are consistent. Second, since each grid point is either empty or contains the ID of a single path, no two paths overlap. As shown in Figure 1, `FindRoute` must preserve this invariant for all paths on `pathList` in addition to the post-conditions that `onePath` is a valid path that connects `p1` to `p2` and is in `pathList`.

Correctness argument for sequential `FindRoute`: When `FindRoute` is viewed as if it is running sequentially, with no interference from other transactions, it is straightforward to verify using VCC. The following are the key steps taken:

- We verify that the code for `shortest_path` satisfies the post-conditions in lines 6 and 7.
- Using this fact, we verify that `gridAddPathIfOK`, if and when it terminates, satisfies the program invariant (no two paths overlap and `pathsList` and `grid` are consistent), and the desired post-conditions in 14.

These are straightforward sequential verification tasks within VCC. The only noteworthy point in this verification effort is the explicit abort in the code for `gridAddPathIfOK` which ensures that `gridAddPathIfOK` terminates only if all grid points on `onePath` are free.

Generalizing the correctness argument to SI: We next outline the intuition for why `FindRoute` remains correct under SI. We would like to emphasize that this intuition is provided solely to improve understanding of this example. It is important to note that:

- the verification effort for `FindRoute` running under SI did not require the user to carry out *any additional reasoning at all* beyond that required for verifying `FindRoute` sequentially. The sequential proof annotations remained correct under SI.
- The source-to-source code transformation is automatable and makes no reference to the example-specific correctness argument below.

In a given instance of `FindRoute` if `gridAddPathIfOK` detects that `onePath` overlaps an existing wire, it explicitly aborts the transaction. In fact, in some interpretations of SI this explicit abort is not even necessary, since this instance of `FindRoute` would be aborted by the transaction manager. This is because grid points on `onePath` are both read and written to by `FindRoute` (see 2) and an update by another transaction to such a point between `FindRoute`'s initial read and later update is not allowed for some interpretations of SI.

Intuitively, instances of `FindRoute` that complete do so because they happen to have computed a path `onePath` that not only does not overlap any of the wires in the initial snapshot `localGridSnapshot`, but also does not overlap any of the paths added to the grid since the snapshot was taken. A successfully completing instance of `FindRoute` ensures that the `onePath` computed and registered into the grid would have been a correct solution *even if* `FindRoute` were operating on an up-to-date snapshot, rather than a stale one.

As we will formally argue using our approach later in the paper, `FindRoute` remains correct when run transactionally under SI because,

- As per SI, the updates to `pathList` and `grid` performed by `gridAddPathIfOK` are carried out atomically,
- to verify that an atomic, terminating execution of `gridAddPathIfOK` establishes the desired program invariant and post-condition, it is sufficient to know that the post-conditions established by `shortest_path` in lines 7 and 8 hold at the time `gridAddPathIfOK` starts running, and,
- since the post-conditions of `shortest_path` in lines 7 and 8 are in terms of transaction-local variables, they continue to hold despite interference from other transactions.

While the exact form of the argument differs from benchmark to benchmark and can be somewhat more complicated than above, we have found that the argument has the following pattern:

1. The "read phase" and the "local computation phase" of the transaction establish some conditions in terms of program variables,
2. These post-conditions suffice for the "write phase" to establish the desired invariants and transaction post-condition, and
3. These post-conditions remain preserved even in the presence of interference allowed under SI.

In the example above, it was trivial to argue (iii) since the post-conditions are in terms of transaction-local variables. For other examples, these post-conditions refer to shared state which may be mutated. The transaction remains correct as long as these mutations do not break the properties that the "write phase" depends on. Our tool allows the program-

mer to express these properties and verify that they remain preserved under interference allowed by SI.

2.2 Genome (Linked List) under `!WAR`

Figures 3 shows the pseudocode for a linked list implementation used in the `Genome` benchmark [2]. The code in the figure has been simplified for ease of presentation. The linked list stores a set of keys in ascending order, and the `list_insert` operation adds a new node to the list preserving this ordering. `prev` and `curr` are transaction-local variables used to traverse the list. They both refer to nodes in the shared linked list.

Figure 4 illustrates how concurrent insertions experience write-after-read (`!WAR`) conflicts, and how, intuitively, it would be correct implementation to let an insertion commit even though it experiences a `WAR` conflict. Following []. the body of `list_insert` is marked with the `!WAR` annotation to indicate that write-after-read conflicts should be ignored.

Sequential correctness argument: The loop invariant in `list_insert` states that the node referred to by `prev` is reachable from `head` and `curr` is reachable from `prev` by following `next` pointers. The `node_t` data structure is annotated with the auxiliary (ghost) field `reach` which contains the set of nodes reachable from a node by following `next` pointers. This ghost field is created in order to be able to write and verify an inductive loop invariants for the while loop in lines 5-11 and is needed even for the sequential verification of a linked list in static verification tools such as VCC. Upon loop termination, this loop invariant is sufficient to ensure that lines 15-16) inserts `node` in the right place in the list and preserve sortedness.

Correctness under `!WAR` relaxation: The correctness of `list_insert` under interference from other transactions as permitted by the `!WAR` relaxation is based on the following facts:

```
struct node_t { int key; node_t* next; ghost Set reach;}
1  bool list_insert(list_t *listPtr,
2                   node_t *node) {
3    node_t *prev, *curr = listPtr->head;
4
5     do {
6       prev = curr;
7       curr = curr->next;
8    } while (curr != NULL
9             && key > curr->key);
10   _(invariant loopInv(prev, curr, head, node))
11   // loopInv(prev, curr, head, node) ==
12   //           prevKey < key && prevKey < curKey
13   //           && prev in head->reach
14   //           && curr in prev->reach
15
16    // assume(prev->next == curr);
17   node->next = curr;
18   prev->next = node;
19   return true; // key was not present
20 }
```

Figure 3. The insertion operation of a sorted linked list.

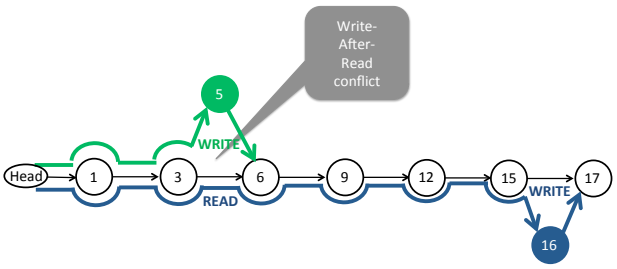

Figure 4. Sorted linked list and a write-after-read conflict.

- The loop invariant remains a correct loop invariant (is preserved) even under the `!WAR` relaxation.
- `loopInv(prev, curr, head, node)` continues to hold until and while lines 17 and 18 are being executed. Even though the `next` fields of list nodes referred to by `prev` and `curr` may change due to interference, `loopInv(prev, curr, head, node)` remains true.
- For a successful execution of `list_insert` after the last iteration, `prev->next` is not written to until line 18, as per the `!WAR` conflict detection scheme. This is because, in this case, `prev->next` is both read and written to by this transaction.
- Starting from a state that satisfies `loopInv(prev, curr, head, node)`, atomic execution of lines 17-18 (guaranteed by the fact that they are both write accesses) ensures the desired post-condition for `list_insert`.

Observe that, if while verifying the sequential interpretation of `list_insert` we had used the more stringent loop invariant `prev->next == curr`, this loop invariant would not be preserved under the `!WAR` relaxation. This is because, concurrently with non-final iterations of the while loop, another transaction running `list_insert` may update the `next` field of the node pointed to by `prev` and insert a new node between `prev` and `next`. The more relaxed invariant in Figure 3 satisfies (i)-(iv) above. Once the user provides this less stringent but still sufficient annotation anticipating interference, our approach not only allows us to statically verify that (i)-(iv) hold. Had the user provided the more stringent invariant, VCC would be unable to verify that `prev->next == curr` is a loop invariant under `!WAR`. This, and VCC's visual debugging (counterexample viewing) facility would then lead the user to provide a less stringent loop invariant.

3. Our Approach

In this section, we formally present our approach to verifying transactional programs running on SI. For simplicity of presentation, we only present our approach for SI. Our approach is applicable to similar relaxed conflict detection schemes such as `!WAR`. We present the formalization for `!WAR`, which is only slightly different from that for SI in the appendix.

We build upon VCC syntax and semantics and give C code with VCC annotations a particular interpretation in order to model database and in-memory transactions. Our model covers equally well both transactions with relaxed consistency semantics being executed by nodes in a distributed program and threads in a shared-memory concurrent program that uses transactional memory with relaxed conflict detection. In the rest of the formalization, for uniformity, we will refer only to threads, with this understanding.

3.1 Preliminaries: Transactional Programs

A *Program Text* consists of a set of valid C functions $\mathcal{F}$. Each C function has the form:

```
func(type_0 arg_0, type_1 arg_1, ..., type_n arg_n)
```

Shared data is represented by aliasing among arguments of functions calls representing different transactions. Unless indicated otherwise in VCC function pre-conditions, when two such functions have arguments that are pointers to the same type, they may potentially alias to the same address. Accesses to such arguments within the function are interpreted as potential shared data accesses. In order to make explicit any potential sharing between transactions, we do not allow any global variables in our VCC programs. All data sharing is modeled via aliasing among input arguments.

A *Program* is a tuple $(Tid, \mathcal{F}, TtoF)$ such that Tid is a set of transactions, $\mathcal{F}$ is a program text and $TtoF : Tid \rightarrow F_{seq}$ is a map which assigns a sequence of functions to each transaction. While referring to an element of the tuple $Element$ representing a program P, we will write it as $Element_P$ (e.g. $GlVar_P$). Transactional programs use C syntax. The code for a transaction has the following structure:

- A call to `beginTrans(`t`)` marks the beginning of a transaction, and a call to `endTrans`(t) marks the end of a transaction. Transaction are not allowed to be nested.
- We make the committing of a transaction syntactically visible by a call to `commitTrans(`t, inv`)`. This call precedes the call to `endTrans`(t). Between `commitTrans(`t, inv`)` and `endTrans`(t), only manipulation of local variables is allowed. The call to `commitTrans(`t, inv`)` allows us to specify invariants that should hold at commit time. When this transaction commits, if inv is not satisfied, a specification violation occurs.

We define states and the transition relation of a program under SI as follows: A *global state* is a tuple $GS = (GlVar, GlMem, TtoLcSts)$ such that

- $GlVar$ is the set of global variables, i.e., shared objects (structs) that multiple transactions hold references to in GS,
- $GlMem : GlVar \rightarrow Val$ maps global variables to their values in the memory, and
- $TtoLcSts : Tid \rightarrow L$ keeps local states of each transaction.

The set of all global states is denoted by G.

A *local state* of a transaction t is a tuple $LS = (LcVar, Stmt, RSet, Wset, ValTrans, LcMem)$ such that

- $LcVar$ is the set of objects local to t,
- $stmt \in Stmt$ is the next statement to be executed by t,
- $RSet \subseteq GlVar$ ($WSet \subseteq GlVar$) is the set of global variables that have been read (written) by t since the beginning of the transaction, until LS was reached,
- $ValTx : Wset \rightarrow Val$ is a map that stores the value of the latest write to each global variable performed by this transaction, and
- $LcMem : LcVar \rightarrow Val$ maps local variables to their values. The domain of this partial map is the local variables of the transaction, i.e. the set of stack variables of the functions in the transaction.

The set of all possible local states is denoted by L. For convenience, we represent $LcSt(t) = \langle LcVar_t, Stmt_t, RSet_t, WSet_t, ValTx_t, LcMem_t \rangle$

In the initial global state, all local states have `beginTrans(t)` as $stmt$, the statement to be executed. A global state s is a final state if and only if for all $t \in Tid$, $TtoLcSts(t)$ is a final local state. A local state s is final if and only if $Stmt = \bullet$, $RSet = WSet = \varnothing$. Err is a global state that corresponds to a specification violation. Err has no outgoing transitions. While referring to an element of a global or a local state s, we will use the notation $Element^s$ (e.g. $GlMem^s$).

The state transition relation for SI (M_{SI}) is denoted by $\triangle_{si} : (s, stmt) \rightarrow s'$ for some global states $s = (GlVar, GlMem, TtoLcSts)$, $s' = (GlVar', GlMem', TtoLcSts') \in G$. For some $t \in Tid$ such that $Stmt_t = stmt$, $(s, stmt) \rightarrow s'$ is defined as follows:

- if $stmt$ is `beginTrans(`t`)`, then $RSet'_t = WSe't_t = \varnothing$.
- if $stmt$ writes val to a local variable a, then $LcMem'_t(a) = val$.
- if $stmt$ assigns a local variable b to a local variable a, then $LcMem'_t(a) = LcMem_t(b)$.
- if $stmt$ creates a fresh global variable a, then $GlVar' = GlVar \cup \{a\}$ and $ValTx'_t(a) = GlMem(a) = \perp$.
- if $stmt$ reads a global variable a, then $RSet'_t = RSet_t \cup \{a\}$ and $ValTx'_t(a) = GlMem'(a)$ if $ValTx'_t(a) == \perp$.
- if $stmt$ writes a local variable b's value to a global variable a, then $WSet'_t = WSet_t \cup \{a\}$ and $ValTx'_t(a) = LcMem_t(b)$.
- `commitTrans(`t, inv`)` statement commits a transaction t and checks if inv is satisfied. If it is, the state is left unchanged, otherwise, execution moves to the global state Err. This statement corresponds to an atomic execution

of a commit operation and an assertion. We find this construct useful when expressing invariants on global state. For all $w \in WSet_t$, $GlMem'(w) = ValTx_t(w)$ is executed atomically..

- if $stmt$ is `endTrans`(t), then $stmt' = \bullet$, $RSet'_t = WSet'_t = \varnothing$ and $ValTx'_t(a) = \perp$ for all variables $a \in WSet_t$.
- if $stmt$ is `assert(`p`)` $s' = Err$ if p is not satisfied in s. p is a boolean formula involving any $a \in LcVar_t$.

Elements of the state other than the $Stmt_t$ remain unchanged if a modification is not specified in the cases above.

An *Action* is a unique execution of a statement by a transaction t in a state s. An *Execution Prefix* of a program P_{SI} is a tuple $E_N = (\vec{s}, \vec{\alpha})$ where $\vec{\alpha}$ is a finite sequence of actions $\alpha_0, \alpha_1, \ldots, \alpha_{N-1}$ and $\vec{s} = s_0, s_1, \ldots, s_N$ is a finite sequence of states such that $(s_i, \alpha_i) \rightarrow s_{i+1}$ for all $i < N$. An execution prefix has the form:

$$s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{N-1}} s_N$$

The formalization so far places no restriction on the interleaving of actions from different transactions. The transaction consistency semantics and conflict detection scheme, such as serial execution of transactions, conflict serializability, and SI specify which interleavings of actions from different transactions are allowed in an execution. We denote the set of all possible prefixes of a program P_{SI} under a semantics M as $Pre(P_{SI}, M)$. A *Program Execution* is an execution prefix $E_F = (\vec{s}, \vec{\alpha})$, such that $\vec{s} = s_0, s_1, \ldots, s_F$ and $\vec{\alpha} = \alpha_0, \ldots, \alpha_{F-1}$, s_0 is an initial state and s_F is a final state of P_{SI} . We denote set of all possible executions of a program P_{SI} under a semantic M as $Execs(P_{SI}, M)$.

3.2 SI and other Relaxed Conflict Detection

We write $Idx_E(\alpha_i)$ to refer to the index i of action α_i in the execution, and $Tr_E(\alpha_i)$ to refer to the transaction performing α_i. For integers i and j such that $i \leqslant j$, we write $[i..j]$ to refer to the set of integers $\{i, i+1, ..., j-1, j\}$. When talking about ordering between actions and states of the same execution, we overload the comparison operators over integers to actions and states, i.e., for $\bullet \in \{<, >, \leqslant, \geqslant\}$, $\alpha_i \bullet_E \alpha_j$ (resp. $s_i \bullet_E s_j$) if $i \bullet j$.

To make precise the sets of executions of a program allowed by different relaxed conflict-detection schemes, we find it useful to define the *protected span* of a shared variable x within a transaction t for a given consistency model M. Intuitively, this span is a set of indices of actions with the property that, according to the consistency model, at none of these indices can an update to x in shared memory take place due to the commit action of a transactions other than t.

Consider an execution E, a transaction t in E and a variable x read in t. The *read span* of x in t is the interval $[i, Idx(commit(t)]$, where α_i is the first action within t that reads x. Similarly, the *write span* of x in t is the interval $[i, Idx(commit(t)]$, where α_j is the first action within t that writes to x. We adopt the convention that the read span of x in t is empty if t does not read x. Similarly for write spans.

In the conflict serializability consistency model, the protected span for a shared variable x within a transaction t is $[i, Idx(commit(t))]$, where i is the index of the first read or write action in t that accesses x. Typical conflict serializability implementations enforce a stricter non-interference condition, where the read and write spans of a variable x within a transaction do not overlap the write span of x in any other transaction.

It is well known that that every conflict-serializable execution is equivalent to a serial one. In most TM implementations [9] the serializability condition above is ensured by maintaining the sets of addresses accessed (the read and write sets) by each transaction and aborting and rolling back transactions that experience disallowed conflicts.

Snapshot Isolation. An execution E is said to obey snapshot isolation iff for all transactions t, (i) all read accesses performed by t are atomic, (ii) all write accesses performed by t are atomic, and (iii) if t both reads and writes to a variable x, the value of x in shared memory is not changed between the first access to x by t and the commit action of t. Requirement (ii) is by satisfied by default for our model of transactions.

Requirements (i) and (iii) are made more precise in the definition below:

Definition 1 (Snapshot isolation)**.** *An execution E obeys snapshot isolation iff there exists an equivalent execution E' such that the following conditions hold:*

- *For all transactions t, the read accesses performed by t are serial in E'.*
- *For all transactions t, if t accesses a variable x more than once, between the first access of t to x in E' and the commit action of t in E', no other transaction that writes to x commits.*

To specify snapshot isolation in terms of spans within an execution, we first define the snapshot read span of a variable x read by a transaction t. Let α_i be the first read action (of any variable) in a transaction t, and let α_j be the last read of a variable x by t. Then, the *snapshot read span* of x in t is the interval $[i, j]$. If x is never read in t, its snapshot read span is the empty interval. The protected span of a variable x in snapshot isolation is defined as follows:

- If x is only read by the transaction, the protected span of x is the snapshot read span of x.
- If x is both read and written to, then the protected span is the interval $[i, j]$ where i is the index of the first access of the transaction to x, and j is the index of the commit action of t.
- If x is only written to, the protected span is defined to be the write span of x.

- Otherwise the protected span is empty.

Snapshot isolation requires that the protected span of each variable x does not contain any commit actions by other threads that write to x. Definition 1 and the definition of snapshot isolation in terms of the snapshot read span are equivalent.

Relaxing Write-After-Read Conflict Detection. We next define the concurrency control semantics, $\mathbf{Rlx}(P)$. This semantics specifies the executions provided by a transactional memory with relaxed detection of conflicts using the `!WAR` annotation as described in [3]. In this semantics, the programmer annotates certain read actions to be *relaxed reads*. The protected span of a variable x in t is still defined as the interval $[i, Idx(commit(t))]$, where α_i is the first regular (not relaxed) read action or write action accessing x as part of t. A relaxed read of x in t is simply required to return the result of the last write to x in shared memory or the last write to x in t, whichever comes later in the execution but before the relaxed read access. Differently from serializable semantics, in read-relaxed semantics, after a relaxed read of x by t but before t commits (or performs a regular read or write to x) other transactions are allowed to commit and update the value of x. In words, after transaction t reads from x, other transactions may write to x arbitrarily many times if it is the case that t itself does not access x again. However, conflicting writes are never allowed between a write access and the corresponding commit action. Therefore, in this consistency model, we can no longer reason about the code for a transaction sequentially.

3.3 Concurrency, VCC and Modular Verification: An Overview

In this section, we briefly, and, due to space constraints, informally, introduce the VCC mechanisms and conventions we make use of in our approach. For an in-depth treatment of VCC, see the manual at

`http://vcc.codeplex.com/documentation`.

Objects and invariants. While the C language does not have objects, VCC allows programmers to think of structs as objects. Each object has a unique owner at any given time. This owner may be another object or thread. The concept of ownership is one mechanism using which access to objects shared between threads is coordinated, and invariants spanning multiple objects are stated and maintained. Objects can be annotated with any number of two-state transition invariants: first-order formulas in terms of any variables in scope in the object that specify the allowed state transitions the object is allowed to make. These invariants may refer to the states of other objects. Objects can be *open* or *closed* at a given time, indicating whether the object is being modified (possibly temporarily violating its invariants) by the owner or not, respectively.

Ghost variables. VCC allows the introduction of ghost variables of all types, including all C types, and more complex ones such as sets or maps. Ghost variables are (auxiliary) history variables, and they do not affect the execution of the program and values of program variables. Maps can be initialized and updated using lambda expressions. Structs can have fields that are ghost variables.

Modular verification. VCC performs modular verification in the following manner. Each function is annotated with pre- and post-conditions. Each loop is annotated with a loop invariant. Every struct may be annotated with two-state transition invariants. Code may also be annotated with assertions in VCC's first-order specification logic, in terms of the program and ghost variables in scope. VCC then verifies the code for one function at a time, using pre-post condition pairs to model function calls, loop invariants to model executions of loops, and "sequential" or "atomic" access, as described below, to model interference from concurrent threads. It generates a first-order verification condition checked by the Z3 theorem prover. Whether all of the annotations are consistent and all assertions, post-conditions, and invariants are satisfied by the code of a function is checked in the same way as typical SMT-based static verifiers, with the exception of how "sequential" and "atomic" accesses to shared objects are modeled. "Sequential access" and "atomic blocks" are two mechanisms using which programmers can describe to VCC how threads within a program coordinate access to shared data.

"Sequential" access and ownership. In "sequential" access, the thread accessing a variable obtains exclusive access to a variable `aVar` by obtaining ownership of `aVar`. This may be accomplished by, for instance, acquiring a lock whose invariant indicates that the lock controls accesses to `aVar`. When the lock is not held by any thread, the lock owns `aVar`. When the lock is acquired by a thread t, the ownership of the object is transferred from the lock to t. The thread then "unwraps" the variable (opens the object it refers to) and manipulates it. While the object is open, its invariants are allowed to be broken temporarily. When the accesses of the thread t are complete, t "wraps" the object and makes it closed. It then transfers its ownership back to the lock by releasing it.

"Atomic" access. One way to coordinate access to shared variables in VCC is to mark them `volatile` and to require that objects they refer to always remain closed. Then, any state transition of the program must adhere to the transition invariants of these objects. Such objects are not ever exclusively owned by any thread, and VCC reasons about a function containing atomic accesses to volatile variables as follows. After each atomic access in the code, VCC only assume that the following are known: (i) the invariants of the atomically-accessed volatile objects hold, and (ii) the states of objects owned by the thread are retained. Atomic accesses

```
StringBuffer* Allocate(StringBuffer* pool){
        StringBuffer* ptr;

        for(int i = 0; i<1000; i++){
            ptr = pool[i];
            if(ptr!=NULL){
                    pool[i] = NULL;
            }
        }
   return ptr;
}
```

Figure 5. StringBuffer pool before code transformation.

are key to how we model the semantics of read accesses according to relaxed consistency. After an atomic read of a reference to object o into thread-local variable l, the verification condition generated by VCC does *not* assume that the state of the object referred to by l does not change. It only assumes that the object referred to by l continues to satisfy the invariants associated with its type. VCC allows the creation of fictitious objects called *claims*. Claims have no state, but only invariants and can be thread-local. One can use a claim to state, prove and retain a formula in terms of variables in scope. VCC verifies in a modular manner that the invariant of a claim holds at the time of its creation, and that it is preserved by atomic accesses to objects by other threads.

3.4 Source-to-source Transformation for Simulating SI

In this section, we present how from a given transactional C program P_{SI} (with VCC annotations) running under SI we produce a program $\widetilde{P_{SI}} = Encode(P_{SI})$ running under ordinary `VCC` semantics making use of constructs presented in the previous section. We formally prove that verifying $\widetilde{P_{SI}}$ under ordinary VCC semantics is equivalent to verifying P_{SI} under transactional SI semantics.

The encoding is obtained via a high-level modeling of the operational semantics of SI using transaction-local and globally-visible shared copies for each object, and VCC statements of the form `assume(`ϕ`)`. A thread in a program can take a state transition by executing `assume(`ϕ`)` only at a state s that satisfies ϕ, in which case, program control moves on to the next statement. Interleavings disallowed by the consistency model M are expressed as a formula ψ in terms of objects' version numbers, and statements of the form `assume` $\neg\psi$ are used in the encoding.

For simplicity, we present the transformation for programs that only use `int` s as primitive types. In the transformation, each shared variable of type `int` is replaced with a variable of type `PInt` as shown below:

```
PInt{
   int inMem;          int inMemVNo;
   int inTM[Trans];    int inTMVNo[Trans];
   Lock lock;
   _(invariant \unchanged(inMemVNo) ==> \unchanged(inMem))
   _(invariant \forall int t;
            \unchanged(inTMVNo[t]) ==> \unchanged(inTM[t]))
};
```

The "wrapper" type `PInt` holds the following information:

```
//Signature of the transformed function
void* Allocate(SBPointer* pool, PTM tm _(ghost \tmbsl claim c))
//Body of transformed function
{
    int i, index;
    void* ptr;
    PTrans ptrans = (PTrans) malloc(sizeof(Trans));
    trans_begin(ptrans, tm _(ghost c));

    for(i=0;i<SIZE; i++)
    _(invariant 0<=i && i< SIZE)
    _(invariant ptrans->tm == tm)
    _(invariant \wrapped(ptrans))
    {
        ptr = tx_relax_read(ptrans, pool[i], tm _(ghost c));
        if(ptr != NULL) break;
    }

    //assume the SB has not been written by another thread
    _(assume pool[i]->version_num == tm->version_numP[pool[i]])
    _(assume (pool[i]->SBPointerLock->locked==0 &&
            pool[i]->SBPointerLock->owning_trans == (PTrans)0 ||
            pool[i]->SBPointer->locked == 1 &&
            pool[i]->SBPointer->owning_trans == ptrans &&
            ptrans->holding[pool[i]]))

    Acquire(pool[i]->SBPointerLock, ptrans _(ghost c))

    //commit to TM
    _(ghost_atomic tm, ptrans, c{
       _(ghost tm->val = (\lambda PInt i;
          ptrans->lockedWritesInteger[i] ? i->data : tm->val[i]))

    _(ghost tm->valP =(\lambda SBPointer pr;
        ptrans->lockedWritesPtr[pr] ? (pr->ptr) :(tm->valP[pr])))

    _(ghost tm->allocated = (\lambda SBPointer pr;
           ptrans->lockedWritesPtr[pr] ? 1 :(tm->allocated[pr])))

    _(ghost tm->version_num =(\lambda PInt pr;
       ptrans->lockedWritesInteger[pr] ?
          pr->version_num : tm->version_num[pr]))

    _(ghost tm->verion_numP = (\lambda SBPointer pr;
            ptrans->lockedWritesPtr[pr]? pr->version_num
            : tm->version_numP[pr]))
    })

    //assert desired invariants and method post-condition
    _(assert tm->allocated[p]==1 && tm->valP[p]==NULL)

    //Pre Release
    unHoldPtr(pool[i],ptrans, tm _(ghost c));
    _(ghost atomic tm, ptrans, c {
            tm->\ owns + pool[i]::g;
            tm->lockedP[pool[i]]=\false;
            tm->holderP[pool[i]]=(PTrans)0;
    })

    //Release
    Release(pool[i]->SBPointerLock, ptrans _(ghost c));
    trans_commit_cleanup(ptrans,tm _(ghost c));
}
```

Figure 6. String Buffer code after transformation.

- a field `inMem` value that corresponds to the value of the variable in shared memory,
- a version number `inMemVNo` that gets incremented atomically each time the `inMem` field is written to,
- a (ghost) field `inTM[Trans]` which is a map from Tid to integers. `inTM[t]` holds the value of the transaction-local copy of the integer

- a (ghost) field `inTMVNo[Trans]` which is a map from Tid to integers. `inTMVNo[t]` is incremented atomically with each update of `inTM[t]`
- a (ghost) field `lock` that is used to convey to VCC when a transaction has exclusive access to the `int` variable

This wrapper type has an important invariant that indicates that a field's value remains unchanged if its version number remains unchanged. This invariant, along with `assume` statements involving version numbers allows us to represent constraints such as the value of a variable remaining unchanged between two accesses within a transaction.

To implement transactional semantics, we use one `Trans` struct per transaction.

```
Trans{
   bool holding[PInt];
   bool readsLockedSet[PInt];
   bool writesLockedSet[PInt];
};
```

Fields of `Trans` are ghost maps that store the variables that have been read and written by a transaction, and variables to which the transaction has exclusive access.

The encoded program makes use of the following VCC statements:

- An assumption statement `assume` ($\tilde{p}$) where $\tilde{p}$ is a boolean formula. After this statement, the execution proceeds considering only the cases that $\tilde{p}$ holds.
- An assertion statement `assert` ($\tilde{p}$) where $\tilde{p}$ is a first order proposition on program variables. This statement aims to prove whether $\tilde{p}$ is satisfied.

$\widetilde{P_{SI}}$, the encoded version of a program P_{SI} is constructed as follows. For each global variable of type `int` in P_{SI}, the encoded program $\widetilde{P_{SI}}$ has a global variable of type `PInt`. For each global `int` variable a in P_{SI}, we denote the corresponding `PInt` variable in $\widetilde{P_{SI}}$ by $\tilde{a}$. When transforming the program syntactically, we use lowercase variables `a` to refer to variables of type `int` in the original program, and uppercase versions (`A`) to refer to the corresponding wrapper variable of type `PInt` in the encoded program.

The code transformation described below describes a bijection $h_{stmt} : Stmt_{P_{SI}} \times Stmt_{\widetilde{P_{SI}}}$ that maps each statement in P_{SI} to its encoded version in $\widetilde{P_{SI}}$. The transformation is described assuming that the code has been decomposed that each statement accesses a global variable at most once, as is typical in transactional applications. The code transformation makes use of a number of C functions whose pre- and post-conditions are presented later in this section.

- Statements of the form `beginTrans(`t`)` remain unchanged in the encoded version.
  h_{stmt}(`beginTrans(`t`)`) = `beginTrans(`t`)`
- Statements that only assign a value val to a local variable or a local variable to a local variable remain unchanged in the encoding.
- Statements that create a new global variable `A` are transformed to `newPInt(A)`.
- Each statement `l = v` by transaction `t` that reads a global variable `v` into local variable l is transformed to an atomically-executed statement that performs the equivalent of the following VCC code atomically.

```
assume( \forall PInt P;
         t->readsLockedSet[P] ==>
              P->inTMVNo[t] == P->inMemeVNo[t]);
l = transRead(t, V);
```

- Each statement `v = l` that writes the value of a local variable `l` to global variable `v` is transformed to the an atomically-executed statements that performs the equivalent of the following VCC code atomically. We make the assumption that a transaction writes a global variable at most once.

```
assume(V->\owner == t || V->\owner == NULL);
acquireLock(V,t);
assume(V->inTMVNo[t] == V->inMemVNo);
//V has not been written to since it was read by t.
transWrite(V, l, t);
```

- Each statement `commitTrans(t, inv)`, is transformed to the following atomically-executed sequence of statements:

```
assume( \forall PInt P;
        t->writesLockedSet[P] ==>
           P->inTMVNo == P->inMemVNo + 1);
commitTrans(t);
assert(INV);
```

- For each statement `endTrans(t)`, we replace the statement with `endAndCleanTrans(t)` in the encoded version.
- Each statement `assert(p)`, where `p` is a boolean expression in terms of local variables, is left as is in the encoded version. Each boolean expression `e` involved in a loop invariant, and function pre- and post-condition is transformed to a boolean expression `E`, where each appearance of a global variable `v` is replaced with a reference to the transaction-local copy `v->inTM[t]`.

The states and transition relation of the encoded program, used in the proof of soundness for our approach, are as follows: A *global state* $(GlMem, TtoLcSts, O)$ extends the global state in the formal model of programs operating under SI by a partial map $O : GlVar \rightarrow GlVar \cup Tid$. O keeps the owner of each global variable (denoted by the ghost `owner` field that each variable automatically gets in VCC) which can be either another global variable or a transaction. Each global variable can be owned by at most one entity at a time. $GlVar_{\widetilde{P_{SI}}}$ is obtained from $GlVar_{P_{SI}}$ by replacing each integer variable by `PInt` variable. $GlVar$(`a->inMem`) represents the value of the `PInt` variable in the memory and $GlVar$($\tilde{a}$ `->inTM(`t`)` represents the transaction-local value of the `PInt` variable in a particular transaction $t \in Tid$. In

the initial global state, the global variables are not owned by any transaction $O(\tilde{a}) = \perp$, and their `inTM` field are not defined

The functions used in the encoded program are listed below together with their preconditions and postconditions:

- `beginTrans(t)` creates a `Trans` structure for thread `t`. This function has no pre-condition and has the post-condition that the read and write sets of `t` and the set of variables `t` has exclusive ownership of are empty, i.e.,

```
\forall PInt P; !t->readsLockSet[P] &&
          !t->writesLockSet[P] && !t->holding[P]
```

- `acquireLock(V, t)` is used to obtain exclusive access to `V` by transaction `t`. This is accomplished by using the fictitious (ghost) lock `V->lock`. Since we are verifying only succeeding executions of transactions (and assuming that aborted transactions have no visible effect), we call `acquireLock` in the encoded program only at a state where it will successfully complete. Thus, this function has the pre-condition that the global variable `V` has no owner or is owned by `t`, and the post-condition that the owner of `V` is the transaction `t`.
- `transRead(V,t)` reads `V` in a transaction `t`. This function does not require `V` to be owned by `t` and has the post-condition that

```
t->readsLockSet[V] == true &&
V->inTM[t] == V->inMEM &&
V->inTMVMo[t] == V->inMEMVNo
```

- `newPInt(V)` is used to create a new PInt variable. This function has the post-condition that `V->owner` is `t`. All version numbers associated with `V` are initialized to 0.
- `transWrite(V, l,t)` writes the value of the local variable `l` to the `inMem` field of `V` and atomically increments `v->inTMVNo[t]`. If `V` has been read previously by `t`, then this function requires that `V`'s version number has not changed since. These are expressed by the pre-condition

```
V->\owner == t && V->inMemVNo == V->inTMVNo[t]
```

and the post-condition

```
t->writesLockSet[V] == true &&
t->inTM[t] == l &&
t->inTMVNo[t] == old(t->inTMVNo[t]) + 1
```

Recall that a transaction can write to a particular global variable only once.

- `commitTrans(t)` commits a transaction by writing the updates performed by the transaction into the memory. Note that a valid execution can have only local statements (that only effect local state) after `commitTrans(`t`)` statement until it ends the transaction. This function is better explained by the following pseudocode

```
_(atomic t {
      \foreach PInt P;
          if (ptrans->writesLockSet[P]) {
               P->inMEM = P->inTM[t];
               P->verNoInMEM = P->verNoInTM[t];
          }
})
```

Since VCC currently does not support loops inside `atomic` statements, the state update corresponding to the loop above is expressed as the function post-condition for `commitTrans`.

- `endAndCleanTrans(t)` ends a transaction `t` by releasing the locks that the transaction holds, cleaning its read and write sets. It has the post-condition that `t` releases ownership of all objects it owns, and the `readLockSet`, `writesLockSet`, and `holding` are all reset to maps corresponding to empty sets.

An example of the encoding is provided in Figures 5 and 6 for a StringBuffer pool example.

The semantics of the encoded program which makes use of these functions is as follows. The *state transition relation* for the transformed program on global states $s, s' \in G$ is denoted by $\triangle_{st} : (s, stmt) \rightarrow s'$. If there exists a $t \in Tid$ such that $TtoStmt(t) = stmt$, then $((GlMem, TtoLcSts, O), stmt) \rightarrow (GlMem', TtoLcSts', O')$ where $TtoLcSts'(t) = (stmt', RSet'_t, WSet'_t, ValTx'_t, LcMem'_t)$ in which $stmt'$ is the next statement to be executed in $\mathcal{F}$:

- If $stmt$ is a function call defined above, then the transition is enabled if s satisfied the function precondition and s' is a state satisfying the postcondition.
- If $stmt$ is a write statement to a local variable, i.e. writing val to a local variable $a \in LcVar_{\widetilde{P_{SI}}}$, then $LcMem'_t = LcMem_t[a \rightarrow val]$ in s'.
- If $stmt$ is `assume(`$\tilde{p}$`)` where $\tilde{p}$ is a boolean formula involving any variables in$GlVar_s$, then the transition is enabled if $\tilde{p}$ holds. The transition only skips to next statement $stmt'$ and no other changes are applied to s'.
- If $stmt$ is `assert(`$\tilde{p}$`)` where $\tilde{p}$ is a boolean formula involving any variables in$GlVar_s$, $s' = Err$ if $\tilde{p}$ is not satisfied. Otherwise, s=s' except for that s' has stmt' as the next statement to be executed.

The following theorem, the proof of which is available at `msrc.ku.edu.tr/projects/vcctm` states the soundness of our verification approach.

Theorem 1 (Soundness)**.** *Let P_{SI} be a transactional program and $\widetilde{P_{SI}}$ be the augmented program obtained from P_{SI} as described above. Then $\widetilde{P_{SI}}$ satisfies its specifications (assertions, invariants, function pre- and post-conditions) if and only if P_{SI} satisfies its specifications.*

It follows from this theorem that users can start with the program P interpreted sequentially, provide the desired

specifications, and additional proof annotations and verify P within VCC. Then, to verify properties of P_{SI}, users can follow the (clearly automatable but not yet automated) source-to-source transformation approach described in this section and obtain $\widetilde{P_{SI}}$. Verifying the transformed specifications with the transformed annotations on $\widetilde{P_{SI}}$ is equivalent to verifying the specifications of P_{SI}, by the soundness theorem.

3.5 The Encoding: Discussion

The source-to-source code transformation described in the previous sections preserves the thread, function, and object structure of the original program. The newly-introduced objects representing transactions are local to each thread or transaction. All additional invariants introduced are per-object. There is no inlining of code from other, possibly interfering transactions, and the size of the transformed code is linear in the size of the original code.

To make modular verification possible in VCC, auxiliary and wrapper objects and their invariants all need to be coordinated carefully using ownership and approval relationships. Since presenting ownership, approval and modular verification in VCC is beyond the scope of this paper, we discuss the rationale for this approach only briefly below.

VCC is the only existing tool for modular static verification of concurrent C programs. By modularity, we mean that a single verification condition generated for each C function, thread and struct/object, and that the effects of other threads and functions are modelled by (i) object invariants, (ii) ownership and approval relationships, and (iii) function pre/post-conditions. The challenge in ensuring modularity is expressing program and proof using (i)-(iii) and ensuring object invariants are "admissible." In return, one gets modular and scalable verification.

The concept of admissibility is key for modularity in VCC and is discussed briefly next: The invariant I_o of object o may refer to o's fields, fields of other objects (e.g. for linked lists, containers), and invariants of other "approver" objects. A program statement s may violate I_o by modifying fields of o or fields of some other object I_o refers to. As a result, s may have to be checked against (potentially) the invariants of all objects. If all object invariants are admissible, one gets more modularity by only checking s against invariants of objects it modifies.

For object o (invariant I_o) we imagine that I_o refers to objects $q_1, q_2, ..., q_n$ and that these objects take transitions consistent with their object invariants $I_{q_1}, ..., I_{q_n}$. I_o is admissible iff it continues to hold after these transitions. To ensure admissibility, one needs to establish ownership and/or approval relationships between o and $q_1, ..., q_n$. This allows I_o and/or I_{q_i} to refer to each other. To make invariants admissible, one needs to carefully orchestrate ownership relationships, often dynamically.

The original program and its correctness argument dictate one set of ownership relationships among program (and therefore, wrapper) objects. Encoding relaxed transactional semantics correctly using objects representing transactions and locks dictates another. Reconciling these conflicting ownership requirements was the key challenge in formulating the encoding. This was accomplished by delineating the "sequentially-accessed," transaction-local fields of the wrapper object as a "group" (nested object) and managing their ownership separately from the "atomically accessed" shared fields of the wrapper object.

In the next section, we report on our experience applying our approach to a number of benchmark programs and the SI and `!WAR` relaxed conflict detection schemes.

4. Experiments

We applied our technique to the Genome, Labyrinth and Self-Organizing Map benchmarks from STAMP [2], a widely-used collection of concurrent benchmark programs containing pre-annotated transactional code blocks, and a String-Buffer pool example. All four of these examples are correct applications but their executions are not conflict serializable. We verified benchmarks under the SI and/or `!WAR` relaxed conflict detection schemes, as explained below. For each benchmark, we wrote partial specifications and statically verified that they hold for transactional code running with the regarding relaxed consistency semantics, starting from a VCC verification of the specifications on a sequential interpretation of the benchmark.

For all benchmarks, the additional user annotations required for sequential verification proved sufficient to verify the benchmarks running under relaxed consistency models as well. Our work makes formal the correctness arguments in the work of Titos et al. [3] about the correctness of the transactions in the benchmarks and provides evidence that the intuitive reasoning about why programs can function correctly under TM relaxations can be expressed and verified systematically and sequentially with moderate effort.

Description and code for the benchmark examples and the results of the code transformation (verified with VCC) are available online at

`http://msrc.ku.edu.tr/projects/vcctm`

Our conclusion from the verification of the encoded programs was that our encoding facilitates modular proofs, and that programmer annotations on encoded program make no reference to auxiliary encoding variables, since they were obtained by transforming the programmer annotations on the sequential program as described in Section 3. In our observation, the key intuition that allowed sequential proofs of transactional programs to generalize to operation under SI was the following: After a read access, even though the read variable may be modified by a concurrent transaction, the variable continues to satisfy all object invariants and other program annotations in the sequential program. This fact is

	Memory Consumption		Time Consumption			
	VCC.exe	Z3.exe	Compiler	Boogie	Verification	Total Time
Seq. Linked List	41.444 K	21.472 K	0.48 s	0.00 s	3.98 s	4.46 s
TM Linked List	58.580 K	84.728 K	1.01 s	0.00 s	23.44	24.45 s
Seq. Labyrinth	39.444 K	3.212 K	0.51 s	0.00 s	0.91 s	1.42 s
TM Labyritnh	64.164 K	75.948 K	1.03 s	0.00 s	17.49 s	18.52 s
Seq. StringBuffer	34.904 K	6.548 K	0.42 s	0.00 s	0.63 s	1.05 s
TM StringBuffer	56.608 K	39.472 K	0.78 s	0.00 s	5.82 s	6.60 s
Seq. SOM	33.068 K	2.268 K	0.31 s	0.00 s	0.56 s	0.87 s
TM SOM	55.028 K	55.356 K	0.87 s	0.00 s	15.88 s	16.75 s

both checked and used by VCC in the proof of the program operating under SI. Put differently, the following abstraction for the interference that a transaction potentially experiences was sufficient for reasoning under SI: The environment of a transaction preserves object invariants and user annotations on the sequential program, but is arbitrary otherwise The key finding from our experiments was that this abstraction is sufficient to argue correctness under relaxed conflict detection.

5. Related work

Relaxed conflict detection.. Relaxed conflict detection has been devised to improve concurrent performance by reducing the number of aborted transactions. Titos et al. [3] introduce and investigate conflict-defined blocks and language construct to realize custom conflict definition. Our work builds on this work, and provides a formal reasoning and verification method for such programs. As we have shown with SI and `!WAR`, we believe that our method can easily be adapted to support other relaxed conflict detection schemes.

Enforcing (conflict) serializability, detecting write-skew anomalies. There is a large body of research on verifying or ensuring conflict or view serializability of transactions even while the transactional platform is carrying out relaxed conflict detection [10–15]. Since runtime and/or static analyses ensure serializability, programmers can reason about transactional code as if it were sequential. For transactional code that is correct but not necessarily conflict or view serializable, as was the case in the examples we studied, verification approaches will signal potential serializability violations while serializability enforcement approaches will result in actual serial execution of transactions. In this work, we enable programmers to verify properties of transactional code on SI even when executions may not be serializable. This allows the user to prove the correctness of and use transactional code that allows more concurrency.

Linearizability:. One way to allow low-level conflicts while preserving application-level guarantees is to use linearizability as the correctness criterion [16]. To prove linearizability of a transactional program P running under SI, one could use the encoded program we construct, $\widetilde{P}$ as the starting point in a linearizability or other abstraction/refinement proof. In this work, we have chosen not to do so for two reasons. First, abstract specifications with respect to which an entire program is linearizable may not exist or may be hard to write. Second, programmers would like to verify partial specifications such as assertions into their program in terms of the concrete program variables in scope. Verifying linearizability does not help the programmer with this task.

Encodings, source-to-source transformations.. As a mechanism for transforming a problem into one for which there exist efficient verification tools, source-to-source code transformations are widely-used in the programming languages and software verification communities. The work along these lines that is closest to ours in spirit involves verifying properties of programs running under weak memory models. Atig et al. [17] propose a method for simulating programs running under total-store order (TSO) semantics with a program running under sequential consistency (SC) semantics. For this purpose, auxiliary variables are introduced to the new program for simulating the store buffers that are part of the TSO operational semantics. Authors prove that the transformed program running under SC correctly models a subset of behaviors of the original program under TSO. Alglave et al. [18] present a sound transformation from programs running under a variety of weak memory models to programs running on the sequential SC memory model. This allows the use of a variety of dynamic and static verification tools for verifying the transformed program. Our work also makes use of a source-to-source translation in order to transform the problem of verifying a transactional program running under SI to a generic C program that can be verified using VCC. Our transformation results in only a linear increase in code size. The distinguishing features of our encoding via source-to-source transformation are as follows.

- We start from code and annotations on a sequential program verified in VCC. In our approach, both the code and the annotations are transformed. In other words, we transform a correctness proof, not only the code that was verified. In the examples we investigated, the transformed annotations provided VCC with enough hints to carry out the verification task under SI.

- In the transformation, the thread, object and procedure structure of the original program is preserved. As a result, while verifying the transactional program under SI, we have the same level of function, object, and thread-modularity that was present in the original VCC verification of the sequential program. No inlining of extra code modeling interference from other transactions is involved.
- While verification of the transactional program under SI may require the user to relax the annotations on the sequential program, the user does not have to provide extra annotations in terms of the extra auxiliary variables in the encoded program.

In this paper, we build on our earlier work [19] where we present a program abstraction that allows us to verify that the abstracted transactional program running under relaxed conflict detection is serializable. While verifying this latter fact, in earlier work, we made explicit use of left- and right-mover actions and commutativity, and the proof was carried out with tool support only for checking the correctness of abstractions and commutativity. In the current work, the entire verification of the transactional program running under SI is carried out within the static verification tool VCC, and the soundness of the verification approach is formally proven (Theorem 1). More importantly, in the current work, we provide an approach to use the correctness proof of the sequential version of the program to derive a correctness proof for the program running under SI.

References

[1] Papadimitriou, C.: The theory of database concurrency control. Computer Science Press (1986)

[2] Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC '08: Proc. of The IEEE International Symposium on Workload Characterization. (September 2008)

[3] Titos, R., Acacio, M.E., Garca, J.M., Harris, T., Cristal, A., Unsal, O., Valero, M.: Hardware transactional memory with software-defined conflicts. In: (To appear) High-Performance and Embedded Architectures and Compilation (HiPEAC'2012). (January 2012)

[4] Skare, T., Kozyrakis, C.: Early release: Friend or foe? In: Workshop on Transactional Memory Workloads. (Jun 2006)

[5] Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: an overview. In: Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. CASSIS'04, Berlin, Heidelberg, Springer-Verlag (2005) 49–69

[6] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI '02, New York, NY, USA, ACM Press (2002) 234–245

[7] Fähndrich, M.: Static verification for code contracts. In: Proceedings of the 17th international conference on Static analysis. SAS'10, Berlin, Heidelberg, Springer-Verlag (2010) 2–5

[8] Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: Vcc: Contract-based modular verification of concurrent c. In: ICSE-Companion 2009. (may 2009) 429 –430

[9] Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory, 2nd edition. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2010)

[10] Dias, R.J., Distefano, D., Seco, J.C., Lourenço, J.: Verification of snapshot isolation in transactional memory java programs. In Noble, J., ed.: ECOOP. Volume 7313 of Lecture Notes in Computer Science., Springer (2012) 640–664

[11] Attiya, H., Ramalingam, G., Rinetzky, N.: Sequential verification of serializability. SIGPLAN Not. **45** (January 2010) 31–42

[12] Alomari, M., Fekete, A., Röhm, U.: A robust technique to ensure serializable executions with snapshot isolation dbms. In: Proceedings of the 2009 IEEE International Conference on Data Engineering. ICDE '09, Washington, DC, USA, IEEE Computer Society (2009) 341–352

[13] Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable isolation for snapshot databases. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. SIGMOD '08, New York, NY, USA, ACM (2008) 729–738

[14] Adya, A.: Weak consistency: a generalized theory and optimistic implementations for distributed transactions. PhD thesis (1999) AAI0800775.

[15] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Transactions on Database Systems (TODS) **30**(2) (2005) 492–528

[16] Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492

[17] Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in tso analysis. In: Computer Aided Verification, Springer (2011) 99–115

[18] Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In Felleisen, M., Gardner, P., eds.: ESOP. Volume 7792 of Lecture Notes in Computer Science., Springer (2013) 512–532

[19] Subasi, O., Elmas, T., Cristal, A., Harris, T., Tasiran, S., Tutos-Gil, R., Unsal, O.: On justifying and verifying relaxed detection of conflicts in concurrent programs. In: WoDet'12

1 Proof for Standard Transactional Semantics

A *Program Text* consists of a set of valid `C` functions $\mathcal{F}$. Each execution of a function in $\mathcal{F}$ belongs to a single transaction. A *Program* is a tuple $(Tid, \mathcal{F}, TtoF)$ such that Tid is a set of transactions, $\mathcal{F}$ is a program text and $TtoF : Tid \rightarrow F_{seq}$ is a map which assigns a sequence of functions to each transaction. While referring to an element of the tuple *Element* representing a program P, we will write it as $Element_P$ (e.g. $GlobalVar_P$).

We define states and the transition relation of a program under standard transactional semantics as follows:

A *Global State* is a tuple $(GlobalVar, GlobalMem, TtoLocalStates)$ such that $GlobalVar$ is the set of objects that can be accessed by multiple transactions in that state, $GlobalMem : GlobalVar \rightarrow Val$ maps global variables to their values in the memory and $TtoLocalStates : Tid \rightarrow L$ keeps local states of each transaction. The set of all Global States is denoted by G.

A *Local State* is a tuple $(LocalVar, Stmt, RSet, Wset, ValTrans, LocalMem)$ such that $LocalVar$ is the set of objects that can only be accessed by the current transaction, $stmt \in Stmt$ represents the current statement to be executed by the transaction, $RSet \subseteq GlobalVar$ ($WSet \subseteq GlobalVar$) is the set of global variables that are read (written) by the transaction so far, $ValTrans : Wset \rightarrow Val$ is a map that keeps latest update on a global variable performed by the transaction (updates on $ValTrans$ will be explained below) and $LocalMem : LocalVar \rightarrow Val$ maps local variables to their values. The domain of this partial map is the local variables of the transaction, i.e. the set of stack variables of the functions in the transaction. The set of all possible local states is denoted by L. For convenience, let us represent

$$LocalState(t) = \langle LocalVar_t, Stmt_t, RSet_t, WSet_t, ValTrans_t, LocalMem_t \rangle$$

Initial global state is the state in which all local states have `beginTrans` as the first statement to be executed. A global state s is a final state if and only if for all $t \in Tid$, $TtoLocalStates(t)$ is a final local state. A local state s is a final if and only if $Stmt = \bullet$, $RSet = WSet = \emptyset$. Let Err represent an erroneous global state that has no outgoing transitions. While referring to a an element of a global or a local state s, we will write it as $Element^s$ (e.g. $GlobalMem^s$).

The *state transition relation* for *standard transaction semantics* (M_{VCC}) is denoted by $\triangle_{st} : (s, stmt) \rightarrow s'$ for some global states $s = (GlobalVar, GlobalMem, TtoLocalStates)$, $s' = (GlobalVar', GlobalMem', TtoLocalStates') \in G$. For some $t \in Tid$ such that $Stmt_t = stmt$, $(s,\ stmt) \rightarrow s'$ is defined as follows:

- if $stmt$ is `beginTrans(`t`)`, then $RSet'_t = WSe't_t = \emptyset$.
- if $stmt$ writes val to a local variable a, then $LocalMem'_t(a) = val$.
- if $stmt$ assigns a local variable b to a local variable a, then $LocalMem'_t(a) = LocalMem_t(b)$.
- if $stmt$ creates a global variable a, then $GlobalVar' = GlobalVar \cup \{a\}$ and $ValTrans'_t(a) = GlobalMem(a) = \perp$.

- if *stmt* reads a global variable a, then $RSet'_t = RSet_t \cup \{a\}$ and $ValTrans'_t(a) = GlobalMem'(a)$ if $ValTrans'_t(a) == \perp$.

- if *stmt* writes a local variable b's value to a global variable a, then $WSet'_t = WSet_t \cup \{a\}$ and $ValTrans'_t(a) = LocalMem_t(b)$.

- `commitTrans(`t, inv`)` statement commits a transaction t iff inv is satisfied which is a proposition obtained by the conjunction of the invariants of the global variables. This can be thought as the atomic execution of a commit operation together with the aasertion of the variable invariants. This ensures the fact that, at the end of every transaction, every invariant on the global variables must be satisfied. For all $w \in WSet_t$, $GlobalMem'(w) = ValTrans_t(w)$ is executed atomically. Then, if inv is satisfied, then execution continues to the next state. Otherwise, the transaction will not be allowed to commit and goes to Err. Note that a program can execute statement only on local variables (that only effect local state) after `commitTrans(`t, inv`)` statement until it ends the transaction.

- if *stmt* is `endTrans`(t), then $stmt' = \bullet$, $RSet'_t = WSet'_t = \emptyset$ and $ValTrans'_t(a) = \perp$ for all variables $a \in WSet_t$.

- if *stmt* is `assert(`p`)` $s' = Err$ if p is not satisfied in s. p is a boolean formula involving any $a \in LocalVar_t$.

Elements of the state other than the $Stmt_t$ remain unchanged if a modification is not specified in the cases above.

An *Action* is a unique execution of a statement by a transaction t in a state s.

An *Execution Prefix* of a program P is a tuple $E_N = (\vec{s}, \vec{\alpha})$ where $\vec{\alpha}$ is a finite sequence of actions $\alpha_0, \alpha_1, \ldots, \alpha_{N-1}$ and $\vec{s} = s_0, s_1, \ldots, s_N$ is a finite sequence of states such that $(s_i, \alpha_i) \rightarrow s_{i+1}$ for all $i < N$. An execution prefix has the form:

$$s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{N-1}} s_N$$

We denote set of all possible prefixes of a program P under a semantic M as $Pre(P, M)$.

A *Program Execution* is an execution prefix $E_F = (\vec{s}, \vec{\alpha})$ be an execution under any semantic, such that $\vec{s} = s_0, s_1, \ldots, s_F$ and $\vec{\alpha} = \alpha_0, \ldots, \alpha_{F-1}$, s_0 is an initial state and s_F is a final state of P .

We denote set of all possible executions of a program P under a semantic M as $Execs(P, M)$.

We transform a program P running under standard transactional semantics into a program $\tilde{P} = Encode(P)$ running under `VCC` semantics. $\tilde{P}$ involves `PInt` typed global variables. This transformation requires to use `PInt` instead of `int` (we give the proof considering only integer variables for simplicity). Moreover, `transRead` method is used to read a global variable and `transWrite` is used to write to a global variable. `PInt` represents a variable of type `int`. Each `PInt` variable has an `InMem` value that keeps the variable's value in global memory

and `InTM` value for each transaction that keeps the value of the variable in the corresponding transaction. `PInt` also keeps the version numbers of `InMem` and `InTM` copies a`->inMemVNo` and a`->inTMVNo[Trans]` that hold the number of updates on a variable. Version number of a variable is incremented when its value is overwritten (i.e. a`->inTMVNo[`t`]` is incremented when a is written in a transaction t). A transaction can write to a `PInt` variable if the transaction owns this variable. In order to be able to move the ownership of a `PInt` variable, each `PInt` is associated to a `Lock` variable.

`Lock` keeps the associated `PInt` variable together with the transaction that owns this `PInt` variable. A `PInt` variable can be owned by only one transaction at a time. Only the transaction owning the lock of $\tilde{a} \in GlobalVar^{\tilde{s}}$ at state s, has the right to write to $\tilde{a}$.

`Trans` represents a thread(transaction) and keeps the read and write sets of the transaction. It also keeps the `PInt` variables it holds (i.e. the `PInt` variables owned by `Trans`).

We define states and transition relation under `VCC` semantics as follows: A *Global State* $(GlobalMem, TtoLocalStates, O)$ extends the global state for standard transactional semantics by a partial map $O : GlobalVar \rightarrow GlobalVar \cup Tid$ under `VCC` semantics. O keeps the owner of each global variable which can be either another global variable or a transaction. Note that each global variable can be owned by at most one entity at a time. In addition, $GlobalVar_{\tilde{P}}$ is formed by replacing each integer variable by `PInt` variable. $GlobalVar($$a$`->inMem`$)$ represents the value of the `PInt` variable in the memory and $GlobalVar(\tilde{a}$ `->inTM(`t`)` represents the value of the `PInt` variable in a particular transaction $t \in Tid$. Since we keep the $inTM$ value of the variable in its structure, we do not use *ValTrans* map in a local state. In the initial global state, the global variables are not owned by any transaction $O(\tilde{a}) = \perp$, and their $inTM$ values are not defined $\tilde{a} \rightarrow inTM(t) = \perp$ for all $\tilde{a} \in GlobalVar_{\tilde{P}}$. While referring to a field $Field$ of a global or a local state s, we will write it as $Field^s$ (e.g. $GlobalMem^s$).

The augmentation of the program makes use of the following VCC statements:

- An assumption statement `assume(`$\tilde{p}$`)` where $\tilde{p}$ is a boolean formula. After this statement, the execution proceeds considering only the cases that $\tilde{p}$ holds.
- An assertion statement `assert(`$\tilde{p}$`)` where $\tilde{p}$ is a first order proposition on program variables. This statement aims to prove whether $\tilde{p}$ is satisfied.

In order to encode standard transactional semantics in `VCC`, we need to define some additional `C` functions. These functions are listed below with their preconditions and postconditions in terms of the program semantics:

- `beginTrans(`t`)` creates a `Trans` structure $t \in Tid$ defined above.

  Precondition: None

Postcondition: $RSet_t = WSet_t = \emptyset$ && $\forall \tilde{a}(\tilde{a} \in GlobalVar_{\tilde{P}} \implies O'(\tilde{a}) = \perp$ && $\tilde{a} \rightarrow inTM(t) = \perp)$.

- `acquireLock(`$\tilde{a}, t$`)` is used to get the lock of $\tilde{a} \in GlobalVar_{\tilde{P}}$ by transaction $t \in Tid$ so that t will be the owner of $\tilde{a}$ and will have the right to write $\tilde{a}$. This function is called after the assume statement which preceeds the call to acquire the lock. The statement `assume`$(O(\tilde{a}) == t \bigvee O(\tilde{a}) == \perp)$ states that the lock of the variable is already acquired by this transaction or it is not acquired by any transactions. Assuming this precondition enables the transaction to acquire the lock.

  Precondition: $O(\tilde{a}) == t \bigvee O(\tilde{a}) == \perp$

  PostCondition: $O(\tilde{a}) = t$

- `transRead(`$\tilde{a}, t$`)` reads $\tilde{a} \rightarrow inMem$ value of $\tilde{a} \in GlobalVar$ in a transaction $t \in Trans$.

  Precondition: $O(\tilde{a}) == t$

  Postcondition: $(GlobalMem(\tilde{a} \rightarrow inTM[t]) == \perp \implies (RSet'_t = RSet_t \cup \{\tilde{a}\}$ && $GlobalMem(\tilde{a} \rightarrow inTM[t]) = GlobalMem(\tilde{a} \rightarrow inMem)$ && $GlobalMem(\tilde{a} \rightarrow inTMVNo[t]) = GlobalMem(\tilde{a} \rightarrow inMemVNo)))$

- `newPInt(`$\tilde{a}$`)` is used to create a new PInt variable.

  Precondition: None

  PostCondition: $GlobalVar_{\tilde{P}} = GlobalVar_{\tilde{P}} \cup \{\tilde{a}\}$ && $O(\tilde{a}) = \perp$ && $GlobalMem(\tilde{a} \rightarrow inMem) = \perp$ && $GlobalMem(\tilde{a} \rightarrow inMemVNo) = 0$ && $\forall t \in Tid(GlobalMem(\tilde{a} \rightarrow inTM[t]) = \perp$ && $GlobalMem(\tilde{a} \rightarrow inTMVNo[t]) = 0)$.

- `transWrite(`$\tilde{a}, localVar, t$`)` writes the value of the local variable $localVar \in LocalVar_P$ to $\tilde{a} \rightarrow inTM(t)$ of $\tilde{a} \in GlobalVar_{\tilde{P}}$ and also increments $\tilde{a} \rightarrow inTMVNo[t]$ in the transaction $t \in Tid$. The method requires that the inMem value of the variable has not changed since it has first read by the transaction. Note that, a transaction can write to a particular global variable only once.

  Precondition: $O(\tilde{a}) == t$ && $GlobalMem_{\tilde{P}}(a \rightarrow inMemVNo) == GlobalMem_{\tilde{P}}(a \rightarrow inTMVNo[t])$

  Postcondition: $WSet'_t = WSet_t \cup \{\tilde{a}\}$ && $GlobalMem'(\tilde{a} \rightarrow inTM[t]) = LocalMem_t(localVar)$ && $GlobalMem'(\tilde{a} \rightarrow inTMVNo[t]) = GlobalMem'(\tilde{a} \rightarrow inTMVNo[t]) + 1$

- `commitTrans(`t`)` commits a transaction by writing the updates performed by the transaction into the memory. Note that a valid execution can have only local statements (that only effect local state) after `commitTrans(`t`)` statement until it ends the transaction.

  Precondition: None

Postcondition: $\forall\tilde{a}(\tilde{a} \in WriteSet_t \implies GlobalMem'(\tilde{a} \rightarrow inMem) = GlobalMem(\tilde{a} \rightarrow inTM[t])$ && $GlobalMem'(\tilde{a} \rightarrow inMemVNo) = GlobalMem'(\tilde{a} \rightarrow inTMVNo[t])$

- `endAndCleanTrans(`t`)` ends a transaction $t \in Tid$ by releasing the locks that the transaction holds, cleaning its read/write sets.

  Precondition: $None$

  PostCondition: $\forall\tilde{a}(\tilde{a} \in GlobalVar_{\tilde{P}}$ && $O(\tilde{a}) == t \implies O'(\tilde{a}) = \perp$) && $\forall\tilde{a}(\tilde{a} \in (WSet_t \cup RSet_t) \implies GlobalMem'(\tilde{a} \rightarrow inTM(t)) = \perp$) && $RSet'_t = WSet'_t = \emptyset$

For `VCC` semantics (together with the functions above), *State transition relation* for the transformed program on *VCC semantics* on global states $s, s' \in G$ is denoted by $\triangle_{st} : (s, stmt) \rightarrow s'$. If there exists a $t \in Tid$ such that $TtoStmt(t) = stmt$, then $((GlobalMem, TtoLocalStates, O), stmt) \rightarrow (GlobalMem', TtoLocalStates', O')$ where $TtoLocalStates'(t) = (stmt', RSet'_t, WSet'_t, ValTrans'_t, LocalMem'_t)$ in which $stmt'$ is the next statement to be executed in $\mathcal{F}$:

- If $stmt$ is a function call defined above, then the transition is enabled if s satisfied the function precondition and s' is a state satisfying the postcondition.
- If $stmt$ is a write statement to a local variable, i.e. writing val to a local variable $a \in LocalVar_{\tilde{P}}$, then $LocalMem'_t = LocalMem_t[a \rightarrow val]$ in s'.
- If $stmt$ is `assume(`$\tilde{p}$`)`where $\tilde{p}$ is a boolean formula involving any variables in$GlobalVar_s$, then the transition is enabled if $\tilde{p}$ holds. The transition only skips to next statement $stmt'$ and no other changes are applied to s'.
- If $stmt$ is `assert(`$\tilde{p}$`)` where $\tilde{p}$ is a boolean formula involving any variables in$GlobalVar_s$, $s' = Err$ if $\tilde{p}$ is not satisfied. Otherwise, s=$s' except for that s'$ has stmt' as the next statement to be executed.

Encoded version of a program P_R under standard transactional model, namely $\widetilde{P_R}$ is obtained by augmenting the program with additional statements or modifications and by exchanging the variables with their wrapper types. The global variables of $\tilde{P}$ is composed of the variables of type `PInt` which is the wrapper type of int. $\tilde{P}$ has $\tilde{a} \in GlobalVar_{\tilde{P}}$ for all $a \in GlobalVar_P$, where $\tilde{a}$ represents the variable in wrapper type of a. The code transformation is defined below together with a bijection $h_s tmt : Stmt_P \times Stmt_{\tilde{P}}$ that maps each statement to its encoded version:

- For each statement `beginTrans(`t`)`, the statement remains same in the encoded version.
  h_{stmt}(`beginTrans(`t`)`) =`beginTrans(`t`)`
- For each statement that assigns a value val to a local variable $a \in LocalVar_P$ (let us call this statement $stmt_{lv}$), the statement remains same in the encoded version.
  $h_{stmt}(stmt_{lv})$ =$stmt_{lv}$

- For each statement that assigns a local variable $b \in LocalVar_P$ to a local variable $a \in LocalVar_P$ (let us call this statement $stmt_{ll}$), the statement remains same in the encoded version.
  $h_{stmt}(stmt_{ll}) = stmt_{ll}$

- For each statement that creates a new global variable (let us call this statement $stmt_{newVar}$), we replace the statement `newPInt(`$\tilde{a}$`)` in the encoded version.
  $h_{stmt}(stmt_{newVar}) =$`newPInt(`$\tilde{a}$`)`

- For each statement that reads from $a \in GlobalVar_{\tilde{P}}$ (let us call this statement $stmt_{readGlobal}$), we replace it with the statements `atomic{assume(`$O[\tilde{a}] = t \bigvee O[\tilde{a}] = \perp$`), acquireLock(`$\tilde{a}, t$`), transRead(`$\tilde{a}, t$`)}`.
  $h_{stmt}(stmt_{readGlobal}) =$`atomic{assume(`$O[\tilde{a}] = t \bigvee O[\tilde{a}] = \perp$`), acquireLock(`$\tilde{a}, t$`), transRead(`$\tilde{a}, t$`)}`

- For each statement that writes the value of a local var $localVar \in LocalVar_P$ to $a \in GlobalVar_{\tilde{P}}$ (let us call this statement $stmt_{writeGlobal}$), we replace it with the statements `atomic{assume(`$O[\tilde{a}] = t \bigvee O[\tilde{a}] = \perp$`), acquireLock(`$\tilde{a}, t$`), assume(`$GlobalMem_{\tilde{P}}(a \rightarrow inMemVNo) == GlobalMem_{\tilde{P}}(a \rightarrow inTMVNo[t])$`), transWrite(`$\tilde{a}, localVar, t$`)}`.
  $h_{stmt}(stmt_{writeGlobal}) =$`atomic{assume(`$O[\tilde{a}] = t \bigvee O[\tilde{a}] = \perp$`), acquireLock(`$\tilde{a}, t$`), assume(`$GlobalMem_{\tilde{P}}(a \rightarrow inMemVNo) == GlobalMem_{\tilde{P}}(a \rightarrow inTMVNo[t])$`),transWrite(`$\tilde{a}, local$

- For each statement `commitTrans(`t, inv`)`, we insert the statement`atomic{commitTrans(`t`), assert(`$\widetilde{inv}$`)}` in the encoded version where $\tilde{inv}$ is transformed version of inv by replacing all $a \in GlobalMem$ in the proposition inv by $\tilde{a} \rightarrow inMem$ in $\widetilde{inv}$.
  $h_{stmt}($`commitTrans(`t, inv`)`$) =$`atomic{commitTrans(`t`), assert(`$\widetilde{inv}$`)}`

- For each statement `endTrans(`t`)`, we replace the statement with `endAndCleanTrans(`t`)` in the encoded version.
  $h_{stmt}($`endTrans(`t`)`$) =$`endAndCleanTrans(`t`)`

- For each statement `assert(`p`)`, we replace the statement `assert(`$\tilde{p}$`)` in the encoded version, where $\tilde{p}$ is transformed version of p by replacing all $a \in LocalVar$ in p by $\dot{a} \in LocalVar$ in $\tilde{p}$.
  $h_{stmt}($`assert(`p`)`$) =$`assert(`$\tilde{p}$`)`

The augmented statements put into atomic blocks are executed atomically. Hence, the augmented statements in an atomic block $stmt_1, stmt_2, ..., stmt_n$ yield a state transition $(s, stmt) \rightarrow s'$ where $stmt =$`atomic{`$stmt_1, stmt_2, ..., stmt_n$`}`. The definition of $h_s stmt$ can be used over action domains as $h : Act_P \rightarrow Act_{\tilde{P}}$.

In order to relate elements of $Pre(P, M_{ST})$ and $Pre(\tilde{P}, M_{VCC})$, we first define a relation between them.

Let $E = (\vec{s}, \vec{\alpha}) \in Pre(P, M_{ST})$ and $E' = (\vec{s}', \vec{\alpha}') \in Pre(\tilde{P}, M_{VCC})$ where $\vec{s} = s_0, \ldots, s_N$, $\vec{s}' = s'_0, \ldots, s'_N$, $\vec{\alpha} = \alpha_1, \ldots, \alpha_{N-1}$ and $\vec{\alpha}' = \alpha_1, \ldots, \alpha_{N-1}$. We define relation $\approx \subseteq Pre(P, M_{ST}) \times Pre(\tilde{P}, M_{VCC})$ such that $E \approx E'$ iff:

– There exist one-to-one functions $f_E^G : GlobalVar^{s_N} \rightarrow GlobalVar^{s'_N}$, $f_E^t : Localvar_t^{s_N} \rightarrow LocalVar_t^{s'_N}$ for all $t \in Tid$ defined recursively as:

  - $f_E^G(a) = f_{E_{N-1}}^G(a)$, for all $a \in GlobalVar^{s_{N-1}}$, $f_E^G(b) = \tilde{b}$ where b is a new global object created by statement $\alpha_{s_{N-1}}$and $\tilde{b}$ is the corresponding object created by $\alpha'_{s_{N-1}}$, if $\alpha_{s_{N-1}}$ is a `newPInt` statement. We denote $f_E^G(a) = \tilde{a}$.
  - $f_E^t(a) = f_{E_{N-1}}^t(a)$, for all $t \in Tid$, $a \in LocalVar_t^{s_{N-1}}$, $f_E^t(b) = \dot{b}$ where b is a new local object created by statement $\alpha_{s_{N-1}}$ and $\dot{b}$ is corresponding object created by $\alpha'_{s_{N-1}}$, if $\alpha_{s_{N-1}}$ is a local variable declaration. We denote $f_E^t(b) = \dot{b}$.
  - Let $E = (s_0, \emptyset).f_E^G$ is any initial one-to-one mapping between global variables in s_o and variables of their wrapped types in s'_0 and and f_E^t's are any initial one-to-one mappings between local variables in s_o and s'_0.

– $GlobalMem^{s_i}(a) = GlobalMem^{s'_i}(\tilde{a} \rightarrow inMEM)$ for all $a \in GlobalVar^{s_i}$, $0 \leqslant i \leqslant N$,

– $LocalMem_t^{s_i}(a) = LocalMem_t^{s'_i}(f_{E_i}^t(a))$ for all $a \in LocalVar_t^{s_i}$,$t \in Tid$, $0 \leqslant i \leqslant N$,

– $ValTrans_t^{s_i}(a) = GlobalMem^{s'_i}(\tilde{a} \rightarrow inTM[t])$ for all $a \in GlobalVar^{s_i}$,$t \in Tid$, $0 \leqslant i \leqslant N$,

– $a \in RSet_t^{s_i}$ iff $\tilde{a} \in RSet_t^{s'_i}$ and $a \in WSet_t^{s_i}$ iff $\tilde{a} \in WSet_t^{s'_i}$ for all $a \in GlobalVar^{s_i}$,$t \in Tid$, $0 \leqslant i \leqslant N$.

Then, we can construct an execution E' in $\tilde{P}$ from an execution E in P by a recursive function $F : Pre(P, M_{ST}) \rightarrow Pre(\tilde{P}, M_{VCC})$ such that:

– $F(s_0) = s'_0$ such that s'_0 is obtained by creating a global variable of wrapped type for each global variable in s_0 and assigning their inMem values same as the initial values of variables in s_0 and inTM values to $\perp$. $O_t = \perp$ for all $t \in Tid$. Rest of the fields are the same as s_0

– Let T be a prefix of E, α be an action in M_{ST} and $\alpha'_0, \ldots \alpha'_k$ be a corresponding sequence of actions in M_{VCC} as explained in the encoding of the program. Then, $F(T \cdot \alpha) = F(T) \cdot \alpha'_0 \cdot \ldots \cdot \alpha'_k$

Then, $E' = F(E)$ becomes the augmented execution of E.

Theorem 1 *Let P be a transactional program and $\tilde{P}$ be the augmented program obtained from P. If $E_N \in Pre(P, M_{ST})$ then there exists an execution prefix $\widetilde{E}_N \in Pre(\tilde{P}, M_{VCC})$ such that $E_N \approx \tilde{E}_N$.*

Proof Proof by structural induction on the length of execution prefix, N. Let $E = (\vec{s}, \vec{\alpha})$ where $\vec{s} = s_0, \ldots, s_N$, $\vec{\alpha} = \alpha_1, \ldots, \alpha_{N-1}$. We will construct $\tilde{E} = (\widetilde{\vec{s}}, \widetilde{\vec{\alpha}})$ where $\widetilde{\vec{s}} = \tilde{s}_0, \ldots, \tilde{s}_N$, $\vec{\alpha} = \tilde{\alpha}_1, \ldots, \tilde{\alpha}_{N-1}$.

As the base step, we build $\tilde{s}_0$ and show that $E_0 \approx \tilde{E}_0$. For all $a \in GlobalVar^{s_0}$ we create $\tilde{a} \in GlobalVar^{\tilde{s}_0}$ of wrapped type, we set $GlobalMem^{\tilde{s}_0}(\tilde{a} \rightarrow inMem) = GlobalMem^{s_0}(a)$ and $GlobalMem^{\tilde{s}_0}(\tilde{a} \rightarrow inTM[t]) = \perp$ for all $t \in Tid$, we also set $f^G_{E_0}(a) = \tilde{a}$, $O^{\tilde{s}_0}(\tilde{a}) = \perp$. Local states of $\tilde{s}_0$ for each $t \in Tid$ are created as: $stmt^{\tilde{s}_0}_t = h_{stmt}(stmt^{s_0}_t)$, for each local variable $b \in LocalVar^{s_0}_t$ we create a local variable $\dot{b} \in LocalVar^{\tilde{s}_0}_t$, set $LocalMem^{\tilde{s}_0}_t(\dot{b}) = LocalMem^{s_0}_t(b)$. Accordingly, $f^t_{E_0}(b) = \dot{b}$. $RSet^{\tilde{s}_0}_t = RSet^{\tilde{s}_0}_t = \emptyset$. One can observe that $E_0 \approx \tilde{E}_0$ with this construction and obviously $\tilde{E}_0$ is a valid execution prefix. Hence, the initial step holds.

As the inductive hypothesis, let $E_i \approx \tilde{E}_i$for some $i < N$. We show $E_{i+1} \approx \tilde{E}_{i+1}$ by considering all possible actions α_i executed by thread t:

- If α_i's statement is `beginTrans(t)`, then $\alpha'_i = h(\alpha_i)$ has `beginTrans(t)` statement. We set $GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTM[t]) = \perp$ for all $\tilde{a} \in GlobalVar^{\tilde{s}_i}$ and $WSet^{\tilde{s}_{i+1}}_t = RSet^{\tilde{s}_{i+1}}_t = \emptyset$. For each $b \in LocalVar^{s_{i+1}}_t$ we create $\dot{b} \in LocalVar^{\tilde{s}_{i+1}}_t$, set $LocalMem^{\tilde{s}_{i+1}}_t(\dot{b}) = LocalMem^{s_{i+1}}_t(b)$. The remaining elements of $\tilde{s}_{i+1}$ are the same as $\tilde{s}_i$.

  $f^t_{\tilde{E}_{i+1}}$ extends $f^t_{\tilde{E}_i}$ such that $f^t_{\tilde{E}_{i+1}}(b) = \dot{b}$ for all $b \in LocalVar^{s_{i+1}}_t$. Since $LocalMem^{\tilde{s}_{i+1}}$ and $LocalMem^{s_{i+1}}$ preserves relation condition, $RSet^{\tilde{s}_{i+1}}_t = WSet^{s_{i+1}}_t = \emptyset = RSet^{\tilde{s}_{i+1}}_t = WSet^{s_{i+1}}_t$, $E_{i+1} \approx \widetilde{E}_{i+1}$ holds.

- If α_i's statement writes a value val to a local variable $b \in LocalVar^{s_i}_t$ then $\alpha'_i = h(\alpha_i)$ has also a statement that writes a value val to a local variable $\dot{b} \in LocalVar^{\tilde{s}_i}_t$. We set $LocalMem^{\tilde{s}_{i+1}}_t(\dot{b}) = val$. The remaining elements of $\tilde{s}_{i+1}$ are the same as $\tilde{s}_i$. Since $LocalMem^{\tilde{s}_{i+1}}$ and $LocalMem^{s_{i+1}}$ preserves relation condition, $E_{i+1} \approx \widetilde{E}_{i+1}$ holds.

- If α_i's statement writes the value of a local variable $c \in LocalVar^{s_i}_t$ to a local variable $b \in LocalVar^{s_i}_t$ then $\alpha'_i = h(\alpha_i)$ has also a statement that writes the value of a local variable $\dot{c} \in LocalVar^{\tilde{s}_i}_t$ to a local variable $\dot{b} \in LocalVar^{\tilde{s}_i}_t$. We set $LocalMem^{\tilde{s}_{i+1}}_t(\dot{b}) = LocalMem^{\tilde{s}_i}_t(\dot{c})$. Induction hypothesis states that $LocalMem^{\tilde{s}_i}_t(\dot{c}) = LocalMem^{s_i}_t(c)$. That yields $LocalMem^{\tilde{s}_{i+1}}_t(\dot{b}) = LocalMem^{s_i+1}_t(b)$. The remaining elements of $\tilde{s}_{i+1}$ are the same as $\tilde{s}_i$. Since $LocalMem^{\tilde{s}_{i+1}}$ and $LocalMem^{s_{i+1}}$ preserves relation condition, $E_{i+1} \approx \widetilde{E}_{i+1}$ holds.

- If α_i's statement creates a new global variable $a \in GlobalVar^{s_{i+1}}$, then $\alpha'_i = h(\alpha_i)$ has a `newPInt(ã)` statement. We set $GlobalVar^{\tilde{s}_{i+1}} = GlobalVar^{\tilde{s}_i} \cup \{\tilde{a}\}$ and $O^{\tilde{s}_{i+1}}(\tilde{a}) = GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inMem) = GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTM[t]) = \perp$ for all $t \in Tid$. The remaining elements of $\tilde{s}_{i+1}$ are the same as $\tilde{s}_i$.

$f^G_{\tilde{E}_{i+1}}$ extends $f^G_{\tilde{E}_i}$ such that $f^G_{\tilde{E}_{i+1}}(a) = \tilde{a}$. Since $GlobalMem^{\tilde{s}_{i+1}}$ and $GlobalMem^{\tilde{s}_i}$ preserves the relation condition. Then, $E_{i+1} \approx \widetilde{E}_{i+1}$ holds.

- If α_i's statement reads $a \in GlobalVar^{s_i}$, then $\alpha'_i = h(\alpha_i)$ involves the statement $\texttt{atomic}\{\texttt{assume}(O(\tilde{a}) = t \vee O(\tilde{a}) = \perp), \texttt{acquireLock}(\tilde{a}, t), \texttt{readTrans}(\tilde{a}, t)\}$. We set $O^{\tilde{s}_{i+1}}(\tilde{a}) = t$, $RSet_t^{\tilde{s}_{i+1}} = RSet_t^{\tilde{s}_i} \cup \{\tilde{a}\}$, $GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTM[t]) = GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inMEM)$ and $GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTMVNo[t]) = GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inMEMVno)$ if $\tilde{a} \notin RSet_t^{\tilde{s}_i}$. $GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTM[t]) = GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inTM[t])$ and $GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTMVNo[t]) = GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inTMVNo[t])$ if $\tilde{a} \in RSet_t^{\tilde{s}_i}$. The remaining elements of $\tilde{s}_{i+1}$ are the same as $\tilde{s}_i$.

  We know by induction hypothesis that if $a \notin RSet_{t' \neq t}^{s_i}$ and $a \notin WSet_{t' \neq t}^{s_i}$, then $a \notin RSet_{t'=t}^{\tilde{s}_i}$ and $a \notin WSet_{t'=t}^{\tilde{s}_i}$. Then, $O^{\tilde{s}_i}(\tilde{a}) \neq t'$ where $t \neq t'$ which shows that proposition $O(\tilde{a}) = t \vee O(\tilde{a}) = \perp$ holds. Therefore, $\texttt{assume}(O(\tilde{a}) = t \vee O(\tilde{a}) = \perp)$ statement satisfies the precondition of $\texttt{acquireLock}(\tilde{a}, t)$ statement. This statement yields $O^{\tilde{s}_{i+1}}(\tilde{a}) = t$ that satisfies the precondition of $\texttt{readTrans}(\tilde{a}, t)$. This statement results in $\tilde{s}_{i+1}$ as described above.

  Since we know that $GlobalMem^{s_i}(a) = GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inMEM)$ and $ValTrans_t^{s_i}(a) = GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inTM[t])$ by induction hypothesis, $ValTrans_t^{s_{i+1}}(a) = GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTM[t])$. Then, $GlobalMem^{\tilde{s}_{i+1}}$ and $GlobalMem^{s_{i+1}}$ preserves relation condition. Moreover, $RSet_t^{\tilde{s}_i+1} = RSet_t^{\tilde{s}_i} \cup \{\tilde{a}\}$ and $RSet_t^{s_i+1} = RSet_t^{s_i} \cup \{a\}$. Thus, relation condition between $RSet_t^{\tilde{s}_i+1}$ and $RSet_t^{s_i+1}$ based on $f^G_{E_{i+1}}$ is preserved. Then, $E_{i+1} \approx \widetilde{E}_{i+1}$ holds.

- If α_i's statement is $\texttt{writeTrans}(a, b, t)$ that writes the value of $b \in LocalVar_t^{s_i}$ to $a \in GlobalVar^{s_i}$, then $\alpha'_i = h(\alpha_i)$ involves the statement $\texttt{atomic}\{\texttt{assume}(O(\tilde{a}) = t \vee O(\tilde{a}) = \perp), \texttt{acquireLock}(\tilde{a}, t), \texttt{assume}(GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inTMVNo[t]) == GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inMEMVno)), \texttt{writeTrans}(\tilde{a}, \dot{b}, t)\}$. We set $O^{\tilde{s}_{i+1}}(\tilde{a}) = t, WSet_t^{\tilde{s}_{i+1}} = WSet_t^{\tilde{s}_i} \cup \{\tilde{a}\}, GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTM[t]) = LocalMem^{\tilde{s}_i}(\dot{b})$ and $GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTMVNo[t]) = GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inTMVNo[t]) + 1$. The remaining elements of $\tilde{s}_{i+1}$ are the same as $\tilde{s}_i$.

  We know by induction hypothesis that if $a \notin RSet_{t' \neq t}^{s_i}$ and $a \notin WSet_{t' \neq t}^{s_i}$, then $a \notin RSet_{t'=t}^{\tilde{s}_i}$ and $a \notin WSet_{t'=t}^{\tilde{s}_i}$. Then, $O^{\tilde{s}_i}(\tilde{a}) \neq t'$ where $t \neq t'$ which shows that proposition $O(\tilde{a}) = t \vee O(\tilde{a}) = \perp$ holds. Therefore, $\texttt{assume}(O(\tilde{a}) = t \vee O(\tilde{a}) = \perp)$ statement satisfies the precondition of $\texttt{acquireLock}(\tilde{a}, t)$ statement. Since an execution can involve at most one $\texttt{writeTrans}(a, b, t)$ for each $a \in GlobalVar^{s_i}$, E_i has no $\texttt{writeTrans}(a, b, t)$ statement. Since $\widetilde{\vec{\alpha}} = h(\vec{a})$, $\tilde{E}_i$ contains no $\texttt{writeTrans}(\tilde{a}, \dot{b}, t)$ statement. Since $GlobalMem^{\tilde{s}_0}(\tilde{a} \rightarrow inTMVNo[t]) = GlobalMem^{\tilde{s}_0}(\tilde{a} \rightarrow inMEMVno) = 0$ and only $\texttt{writeTrans}(\tilde{a}, b, t)$ can increment $GlobalMem^{\tilde{s}_0}(\tilde{a} \rightarrow inTMVNo[t])$, assume$(GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inTMVNo[t]) == GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow$

$inMEMVno$)) holds. The precondition of `writeTrans(`$\tilde{a}, \dot{b}, t$`)` is satisfied and this statement results in $\tilde{s}_{i+1}$ as described above.

Since we know that $LocalMem^{s_i}(b) = LocalMem^{\tilde{s}_i}(\dot{b})$ by induction hypothesis, $ValTrans_t^{s_{i+1}}(a) = GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTM[t])$. Then, $GlobalMem^{\tilde{s}_{i+1}}$ and $GlobalMem^{s_{i+1}}$ preserves relation condition. Moreover, $WSet_t^{\tilde{s}_{i+1}} = WSet_t^{\tilde{s}_i} \cup \{\tilde{a}\}$ and $WSet_t^{s_{i+1}} = WSet_t^{s_i} \cup \{a\}$. Thus, relation condition between $WSet_t^{\tilde{s}_{i+1}}$ and $WSet_t^{s_{i+1}}$ based on $f^G_{E_{i+1}}$ is preserved. Then, $E_{i+1} \approx \widetilde{E}_{i+1}$ holds.

- If α_i's statement is `assert(`p`)` where p is a first order proposition on $b \in LocalVar_t^{s_i}$, then $\alpha'_i = h(\alpha_i)$ has also a statement `assert(`$\tilde{p}$`)` where $\tilde{p}$ is a first order proposition on $\dot{b} \in LocalVar_t^{\tilde{s}_i}$. If s_{i+1} is the erroneous state, then we set $\tilde{s}_{i+1}$ to the erroneous state. Otherwise, $\tilde{s}_{i+1}$ differs from $\tilde{s}_i$ by its $stmt_t$ element. We know by induction hypothesis that $LocalMem^{s_i}(b) = LocalMem^{\tilde{s}_i}(\dot{b})$ for all $b \in LocalVar_t^{s_i}$. Therefore, p is satisfied on s_i iff $\tilde{p}$ is satisfied on $\tilde{s}_i$. Then, $E_{i+1} \approx \widetilde{E}_{i+1}$ holds.

- If α_i's statement is `commit(`t, inv`)` where inv is a first order proposition on $a \in GlobalVar^{s_i}$, then $\alpha'_i = h(\alpha_i)$ has also a statement `atomic{commit(`t`),assert(`$\widetilde{inv}$`)}` where $\widetilde{inv}$ is a first order proposition on $\tilde{a} \in GlobalVar^{\tilde{s}_i}$. We set $GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inMEM) = GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inTM[t])$ and $GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inMemVNo) = GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inTMVNo[t])$for all $\tilde{a} \in WSet_t^{s'_{i+1}}$. The remaining elements of $\tilde{s}_{i+1}$ are the same as $\tilde{s}_i$.

  Induction hypothesis states that $\tilde{a} \in WSet_t^{\tilde{s}_i}$ iff $a \in WSet_t^{s_i}$ and $ValTrans_t^{s_i}(a) = GlobalMem^{\tilde{s}_i}(\tilde{a} \rightarrow inTM[t])$. Therefore, $GlobalMem^{s_{i+1}}(a) = GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inMEM)$ for all $a \in WSet_t^{s_i}$ and the relation condition holds for $GlobalMem$ elements of $\tilde{s}_{i+1}$ and s_{i+1} after `commit(t)`. Due to this fact inv holds after `commit(`t`)` iff $\widetilde{inv}$ holds after `commit(`t`)`. Then, $E_{i+1} \approx \widetilde{E}_{i+1}$ holds.

- If α_i's statement is `endTrans(`t`)`, then $\alpha'_i = h(\alpha_i)$ has a statement `endAndCleanTrans(`t`)`. We set $GlobalMem^{\tilde{s}_{i+1}}(\tilde{a} \rightarrow inTM[t]) = \perp$ for all $\tilde{a} \in GlobalVar^{\tilde{s}_i}$, $O^{\tilde{s}_{i+1}}(\tilde{a}) = \perp$ for all $\tilde{a} \in WSet_t^{\tilde{s}_i} \cup RSet_t^{\tilde{s}_i}$ and $WSet_t^{\tilde{s}_{i+1}} = RSet_t^{\tilde{s}_{i+1}} = \emptyset$. The remaining elements of $\tilde{s}_{i+1}$ are the same as $\tilde{s}_i$.

  Since $RSet_t^{\tilde{s}_{i+1}} = WSet_t^{s_{i+1}} = \emptyset = RSet_t^{\tilde{s}_{i+1}} = WSet_t^{s_{i+1}}$, $E_{i+1} \approx \widetilde{E}_{i+1}$ holds.

□