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. 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.

We present 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.

1. Introduction

Transactions provide a convenient, composable mechanism for writing concurrent and distributed programs. They are used to write shared memory programs using Transactional Memory, programs that access a single, central databases, and, more recently but increasingly widely, to write programs that access geo-replicated databases. In each of these domains, the transactional platform can provide a strict guarantee. For TM and database transactions, this is atomicity and serializability of transactions, while, for geo-replicated data types and databases, this is strong conistency. Again, in each of these domains, transactional platforms provide less strict guarantees. For TM and database transactions, the reason is performance and avoidance of frequent transaction aborts. Examples from TM and database domains include snapshot isolation (SI) and its variants (the default consistency mode provided by popular databases), early release of readset entries, and programmer-defined conflict detection. For geo-replicated databases, the choice to avoid strong consistency is motivated by providing availability to the datatype, even when the replicas are disconnected ("partition"ed). This is especially the case when each mobile device has its own replica of (a subset of) shared objects, a programming model used more and more widely in computing platforms consisting of cloud devices and the cloud. For geo-replicated databases, programming models that relax the strong consistency guarantee are variants of eventual consistency, with additional guarantees relating different objects and the order in which different updates are propagated to different replicas.

When the transactional execution platform provides strong consistency and serializable transactions, the code of a transaction can be treated as sequential code. In this case, the code for transactional applications can be treated as sequential. This significantly simplifies writing and verifying applications. However, as argued above, for performance and availability reasons, it is increasingly the case that transactional programs run on a platform that implements a weaker consistency model. One way to retrieve the simplicity of sequential programming on relaxed transactional platforms is to deploy analyses or implement additional analyses in order to ensure that the particular program's transactions run in a serializable manner (e.g. by preventing or avoiding write-skew anomalies) or under strong consistency although the underlying platform only provides relaxed consistency. This approach can be useful some of the time, but, for many examples, especially for geo-replicated databases, it may defeat the purpose. Much of the time, it is the application author's intent to implement a transactional program that is correct, i.e. satisfies assertions and invariants and other de-

sirable application-level properties, while being cognizant that the transactional platform provides a guarantee weaker than serializability or strong consistency.

Typically, the way relaxed consistency exhibits itself in transactional code is in the form of "stale reads"' – data read by the transaction may not be the most recent version later during the transaction, or even at the time of the read access, in the case of geo-replicated databases. Of course, some guarantees about the set of reads (e.g. causality, limits on how stale, not seeing partial results of other transactions, cross-object consistency) are provided by the weak transactional semantics.

The intuition behind writing such programs is ensuring that

- the transaction computes a correct result despite stale reads, or - optimistically computing a possibly-correct result, verifying at commit time that the result is indeed correct (does not violate application-level properties) and aborting the transaction otherwise.

Unfortunately, there are currently no tools for verifying properties (assertions and invariants) of transactional programs whether the transactional program provides serializability and strong consistency or not. Static tools for code verification targeted at sequential programs [**? ? ?**], and the VCC verification tool [**?**] 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. For transactional programs running with the serializability or strong consistency guarantee, transactional code can be treated as sequential and these existing tools can be used. But for the increasingly more common transactional platforms with SI or eventual consistency, these tools cannot be used as they are not aware of transactions or relaxed consistency semantics. In this paper, we present a static code verification technique for transactional applications running under weak consistency semantics such as SI or eventual consistency. The goal of our technique is to provide a verification environment exactly like that of VCC but for programs running on relaxed tranasactional platforms. The verification approach provides scalability and modularity, as VCC does, but requires programmer annotations for procedure pre- and post-conditions and loops in the same way all existing modular static code verification tools do.

In our approach, we take a transactional program and the semantics of the transactional platform that provides relaxed consistency. We produce an augmented C program with VCC annotations. The program our approach outputs is the same (has the same structure, etc.) as the input program, but includes an encoding of the relaxed transactional semantics and exactly the executions and interleavings involved there through the use of auxiliary variables in VCC.

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. This encoding can be viewed as a very high-level implementation of the transactional platform, where guarantees provided by the transactional platform are modeled using "atomic" code blocks and "assume" statements. The transformation 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.

To illustrate the applicability of our technique, we verified programs that were written to be correct even under relaxed transactional consistency. These programs were three benchmarks from the STAMP [**?**] transactional benchmarks suite and a `StringBuffer` pool example. We verified using our approach and tool that these examples satisfy application-level invariants and assertions despite relaxed transactional semantics. The user annotations required in each case reflected the correcness intuition in the relaxed transactional case, were relaxed versions of the annotations that would have been required had the program been running sequentially, and did not refer to the auxiliary variables involved in our encoding. In other words, the user was able to simply express the correctness intuition without referring to the mechanisms implementing the transactional semantics. Verification times for programs running under relaxed semantics were comparable with verification times for programs running sequentially.

Although we recognize the distinctions between these models, in discussions about relaxed conflict detection in this paper, for brevity's sake, we use SI to refer to relaxed consistency models similar to SI and "serializability" to stand for the class of transactional platforms that provide atomic, strongly consistent, serializable transactions.

2. Motivating Example

In this section, we present at a high level one of the four benchmark programs we applied our method to. We do this for two purposes: First, this example is typical of the design and correctness intuition for programs that satisfy desired assertions and invariants while operating under SI. Second, on this example, we provide intuition fot how this example running under SI is verified in our approach.

This example has the property that it is correct despite its executions not being serializable, therefore, enforcing serializability (as is typically accomplished by enforcing conflict serializability [**?**]) would be an unnecessary restriction that hurts performance. The example follows a common parallel programming pattern, the transactions in this example read

```
// 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 of 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,
14     // and the post-condition
15:    //   onePath \in pathsList &&
16:    //        IsConnectingPath(onePath, p1, p2)
} }
```

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

a large portion of the shared data, perform local computation and update only a small portion of the shared data. Under conflict-serializability all concurrent transactions conflict and transactions can only run serially, one at a time.

As shown in 1, in the Labyrinth benchmark from the STAMP benchmark suite, 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 the unique ID of the path (wire). 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, but does *not* guarantee that the entire transaction is atomic.

Desired properties for this program are that (i) the `grid` is filled correctly by the information, and that (ii) no two paths overlap. The latter of these is implicitly ensured because each grid point contains a single wire ID number. The former is formally expressed below

```
isValidPath(int ***grid, path_t* p) =
  (\forall int i; 0<= i < path->path_len ==>
     p->ID == grid[p->x[i]][p->y[i]][p->z[i]])
   \forall int i; 0<= i < path->path_len-1 ==>
     isAdjacent(p->x[i], p->y[i], p->z[i],
             p->x[i+1], p->y[i+1], p->z[i+1])
```

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`.

Static Verification of 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.

Verifying `FindrRoute` Under SI: We next outline the intuition for why `FindRoute` remains correct under SI.

In a given instance of `FindRoute`, if `gridAddPathIfOK` detects that `onePath` overlaps an existing wire, it explicitly aborts the transaction. Intuitively, instances of `FindRoute` that complete do so because they 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.

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.

In our approach, the code in Figure 1 is augmented with auxiliary variables using which the semantics of SI is encoded. Below, we list a few highlights of this encoding:

- We associate with each variable (e.g. each grid cell and path element) a version number.
- Using "assume" statements, we encode the fact that the grid snapshot in line 1 is taken atomically. This snapshot is accomplished by traversing the grid and building a transaction-local snapshot of it. To indicate that while reading grid cell `grid[i1][j1][k1]` a grid cell `grid[i2][j2][k2]` read earlier has not been modified by a concurrent transaction, we write
  `assume grid[i1][j1][k1].version localGridSnapshot[i1][j1][k1].version`

- Using fictitious locks and the "atomic" statement in VCC, we apply the updates carried out by the tranasaction indivisibly.

The encoded file preserves the structure of the original, and does not inline code from other possibly interfering transactions, as is the case with some techniques on static verification of concurrent programs.

Apart from the restrictions expressed by the encoding as explained above, the resulting program is an annotated, concurrent VCC program. By verifying the invariants and assertions in the resulting program, the programmer is ensured that the original program running under SI satisfies the invariants and assertions.

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 correctness argument for the programs we target 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 conditions remain true even after the global state is changed by other concurrent transactions committing
3. These conditions suffice for the "write phase" to establish the desired invariants and transaction post-condition, and

3. Our Approach

In this section, we formally present our approach to verifying transactional programs running on SI. We first present our formal model and a formalization of relaxed transactional semantics. We then give an overview of VCC and explain how our source-to-source code translation works.

3.1 Preliminiaries: Transactional Programs

In this section, we define some formal definitions that will later be used to describe our program transformation, relaxed transactional semantics.

Program texts we consider are sets of valid C functions. Functions may contain a list of arguments. These function arguments are interpreted by our method as shared objects. If the user wants a function argument to be interpreted as thread local, she should annotate it as thread local in the function precondition. If there exists two pointer arguments of the same type in a function, they may be interpreted as aliases to same address. To prevent this, difference between them should be stated in the function precondition. All data sharing is modelled via aliasing among input arguments.

A *program* is a function which maps a given transaction to a sequence of functions from program text. In our approach, transactions have the following structure:

- A call to `beginTrans` function initiates the transaction and returns a unique transaction id t
- Transactions may read a shared variable to a local copy if it is allowed by consistency semantics. However, write operations on shared variables are performed on the local copy until commit of the transaction. When a transaction calls `commitTrans(`t`)` function, all objects in the write set of t are updated atomically by their values in the local copy, if it is allowed by consistency semantics. If it violates consistency semantics, transaction aborts without updating any shared variable. Every transaction calls `commitTrans` function once and only once and after calling this function, the transaction is assumed not to perform any reads or writes on the shared variables.
- The transaction t terminates by calling `endTrans(`t`)` function.

The transactional programs we considered contain both local and shared variables. A program *state* is a function like heap which assigns values to both local and shared variables.

Program *executions* are sequences of *actions* which are program statements executed by a transaction. Each action takes a program state and changes it into another state depending on the semantic interpretation of the statement it executes. An execution shows the history of a particular run of the program including interleavings of the transactions. We do not provide any formal details about executions since they are the similar to history or schedule definitions of the many studies in the literature.

We say that a transaction is successful for an execution, if transaction successfully commits (or does not abort) in this execution. For this study, we consider set of executions of which all transactions are successful. Since aborted transactions do not have any visible effect on the program state projected on to shared variables, there is nothing to verify about these transactions. Thus, we may neglect them.

3.2 SI and other Relaxed Conflict Detection

Since an execution is a sequence of actions, we can enumerate its actions i.e., $E = e_1, e_2, e_3, ...$ Then, we can define an interval $[i, j]_E$ of an execution $E = e_1, e_2, e_3, ...$ as the sequence $e_i, e_{i+1}, ..., e_j$, where i and j are non-negative integers and $i < j$.

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 M, at none of these indices a transaction other than t can update the value of x.

Consider a variable x read by transaction t in execution E. Then, *read span* of x in t on E is the interval $[i, j]_E$, where e_i is the first action within t that reads x and e_j is the commit action of t. Similarly, the *write span* of x in t on E is the interval $[i, j]_E$, where e_i is the first action within t that writes to x and e_j is the commit action of t. We adopt the

convention that the read span of x in t is empty if t does not read x. Similarly for write spans.

Snapshot Isolation. A successful 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 a transaction t both reads and writes to a variable x, then x can not be modified by another transaction between first access to x by t and commit of transaction t. These requirements can be expressed by precisely defining protected span for SI.

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 in execution E. Let e_i be the first read action (of any variable) in a transaction t, and let e_j be the last read of a variable x by t. Then, the *snapshot read span* of x in t on execution E is the interval $[i, j]_E$. 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 **??** and the definition of snapshot isolation in terms of the snapshot read span are equivalent.

3.3 Concurrency, VCC and Modular Verification

In this section, we briefly, and, due to space constraints, informally, introduce the VCC mechanisms and conventions we make use of in our approach.

VCC allows programmers to think C structs as objects and other base C types (int, char, double etc.,) as primitive types. VCC allows programmer to create `ghost` objects or declare `ghost` structs which can not modify the concrete program state but can be used for verification tasks. `ghost` structs can be C structs defined in the program or special types provided by VCC.

Each object has a unique owner at any given time. 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.

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.

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. In "sequential" access, the thread accessing a variable obtains exclusive access to a variable `aVar` by obtaining ownership of `aVar`. Another way to coordinate access to shared variables in VCC is to mark them `volatile` and to require that any state transition of the program must adhere to the transition invariants of these objects.

3.4 Source-to-source Transformation for Simulating SI

In this section, we present a source-to-source transformation. The input to the transformation is C program P_{SI}. P_{SI} contains program text and user annotations that are the program specifications. These specifications are:

- struct invariants for user defined data types,
- desired function pre- and post-conditions which are boolean expressions about inputs and outputs of the functions
- assertions which are boolean expressions over transaction local or shared variables that all the program states at the point of executing this statement must satisfy.

User is not supposed to provide annotations about ownership of the shared objects. Ownership mechanism for shared objects is established by transformation. However, user can provide ghost data types and objects. Then, she can write annotations about ownership mechanisms of these objects.

The output is a program $\tilde{P_{SI}} = Encode(P_{SI})$ that will be verified with VCC. It runs under ordinary C semantics and contains the kinds of VCC annotations described in Section **??**. We formally prove that verifying $\tilde{P_{SI}}$ under ordinary VCC semantics is equivalent to verifying P_{SI} under transactional SI semantics.

The encoding is obtained via a high-level modelling of the operational semantics of SI. The model uses 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.

Transforming data types: Original transactional program could use primitive C types (int,char etc.) and user defined structs. During transformation, we wrap primitive C types into new structs so that we can use ownership and synchronization mechanisms on them and augment user defined structs by adding new fields to them for the same reason. We explain data type transformation by giving example for both primitive C types and user defined structs.

First consider primitive C type `int`. After transformation, we define a new struct `Int` as follows:

```
Int{
   int value;          int vNo;
   SpanFormer* spanFormer;
   _(invariant \unchanged(vNo) ==> \unchanged(value))
   _(invariant \unchanged(vNo) || vNo == \old(vNo)+1)
   _(invariant spanFormer ==> spanFormer->protectedObj == this)
};

SpanFormer{
   object token; object protectedObj;
   _(invariant mine(token) || mine(object))
}
```

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

- a field `value` that corresponds to the value of the variable in shared memory,
- a version number `vNo` that gets incremented atomically each time the `value` field is written to,
- a (ghost) field `spanFormer` that is used for coordinating access to the object. It serves two purposes. First, it is used to prevent two concurrent transactions from simultaneously writing to this `Int`. Second, when a transaction is in the point of committing, the `spanFormer` is used to prevent other transactions from reading and writing to the `Int` object. Initially, `spanFormer` "owns" its `protectedObj`. When a transaction t wants to write on an `Int` object x, it first takes ownership of its `spanFormer`'s `token`. Then, no other transaction can get the ownership of x and write to it due to invariant of `spanFormer` but could read it. When t commits, it exchanges `token` with x and could modify it.

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.

Now we present transformation of a user defined struct `Node`, which represents a node in a sorted linked list. In the original program, this struct is defined as follows:

```
Node{
   int data;  Node* next;

   _(invariant next ==> next->data > data)
};
```

Then, the transformed node struct is:

```
Node{
  //Transformed user input:
  Int data;  Node* next;

  _(invariant   next ==> next->data.val > data.val)
  //Augmentation:
  int vNo; SpanFormer* spanFormer;

  _(invariant \mine(data) && !data.spanFormer)
  _(invariant \unchanged(vNo) ==> \unchanged(data.value) && \unchanged(
  _(invariant \unchanged(vNo) || vNo == \old(vNo)+1)
  _(invariant spanFormer ==> spanFormer->protectedObj == this)
};
```

For this struct transformation, we emphasize the following points:

- Original type of the `data` field is transformed to `Int` and user invariant on this data changed accordingly. This is done for transformation of user defined structs with primitive fields. If a struct fields is another user defined struct field, this change is not performed.
- Ownership of the `data` field is given to `Node` struct. However, this is not true for `next` field. Since node pointed by `next` is a distinct object, its synchronization and ownership should be managed by itself (i.e., by its `vNo` and `spanFormer` fields). However, `data` must be under control of the node. Therefore, we give ownership of `data` to node and set its `spanFormer` to null. For all fields with a pointer type, we apply transformation as in `next`. Otherwise, transformation applied is like `data`.
- Since `data` is managed by node, its `vNo` field must be synchronized with node. When a transaction commits by changing `data`, it should increment version number of both `data` and node.

To implement transactional semantics, we define one more struct called `Trans` and provide an instance of this struct per transaction.

```
Trans{
   bool readSetInt[PInt];
   bool writeSetInt[PInt];
   PInt localIntCopy[PInt];
   ...
};
```

In the definition above `PInt` stands for `struct Int*`. Fields of `Trans` are ghost maps. `readSetInt` and `writeSetInt` are maps that store `Int` objects read and written to by this transaction, respectively. `localIntCopy` keeps the transaction local values of `Int` s. If an `Int` object x is neither read nor written to by this transaction then `localIntCopy[x]` is null.

If there are struct declarations in the original program, `Trans` contains three maps for each of these structs used in the same reasoning with `Int`.

Transformation of Functions 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, only the executions that satisfy $\tilde{p}$ can proceed.

- An assertion statement `assert` ($\tilde{p}$) where $\tilde{p}$ is a first order proposition on program variables. $\tilde{p}$ must be satisfiable for all executions that reach to this statement.

In the following, we describe how we transform functions. 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. Due to space restrictions, we only provide highlights of the transformation rules:

- Statements of the form `beginTrans`(t) remain unchanged in the transformed version. (see pre- post-conditions of this function below)
- Statements that only assign a value val to a local variable or a local variable to a local variable remain unchanged in the transformation.
- Statements that create a new shared variable `A` of type `Int` are transformed to `newPInt(A)`. This is similar for creating new shared variable of other types.
- 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;
         trans->readSet[P] ==>
              trans->localIntCopy[P]->vNo == P->vNo);
l = transReadInt(trans, V);
```

Consider `grid` example. It takes snapshot of the grid like in the following pseudo code:

```
foreach(i,j,k in grid's range){
   localGrid[i][j][k] = grid[i][j][k];
}
```

When we transform this code, we obtain something as follows:

```
foreach(i,j,k in grid's range){
   _(atomic){
      //assume previously read grid points are the same
      _(assume \forall PInt P; trans->readSet[P] ==>
       trasn->localIntCopy[P]->vNo == P->vNo)
      localGrid[i][j][k] = transReadInt(trans,grid[i][j][k]);

   }
}
```

One can observe that after execution of above code, grid has been read atomically since it is assumed that no one has changed the grid.

- Each statement `V = l` that writes the value of a local variable `l` to a shared variable `V` is transformed to the an atomically-executed statements that performs the equivalent of the following VCC code atomically.

```
_(assume (trans->readSet[V] ==>
           trans->localIntCopy[V]->vNo == V->vNo) &&
         (V->owner == t ||
            V->owner == V->spanFormer)
  )
beginWriting(V,t);
transWriteInt(V, l, trans);
```

At some point in the function `addGridPathIfOK` of grid example, original code looks like the following:

```
foreach(i,j,k in grid's range){
    grid[i][j][k] = localGrid[i][j][k];
}
```

This code is transformed like following pseudo code:

```
foreach(i,j,k in grid's range){
  _(assume readSet[grid[i][j][k]] ==> \unchanged(grid[i][j][k]) &&
        grid[i][j][k]->owner is not any other transaction)
  beginWriting(grid[i][j][k],t);
  _(atomic){
      transWriteInt(grid[i][j][k],localGrid[i][j][k],trans);

  }
}
```

This code achieves that grid points that are on the `onePath` has not changed because:

  - They stay the same until `beginWriting` by assumption,
  - Their ownership can be taken inside `beginWriting` before any other object can modify them inside due to previous assumption. Since `beginWriting` gives ownership of tokens to the transaction (see its post condition below), no other transaction can modify them from this point on (see invariant of `SpanFormer`).

- Each statement `commitTrans(t, inv)`, is transformed to a sequence of statements. In this sequence, first we get ownership of each object we want to commit such that no other transaction can read or write to them any more. Then, we modify them and make them readable and writeable by other transactions. Since effects of failing transactions are not visible, we only model succeeding executions of a transaction. This being the case, when t is about the commit, the objects in t's modify must have stayed the same since first access of t. This is achieved by the code we transformed while writing to the global data. Therefore, commit is transformed as:

```
commitTrans(trans,t);
assert(inv);
```

At the point of commit we are sure that the objects t wants to modify has not changed since first time it has been read or written to because:

  - if first access is a read, transformation of the write statement above ensures that it has not changed since the read and it can not change after the write due to the fact that t holds token.
  - if first access is a write, t owns the token after write and it can not change until commit.

In the grid example, transaction commits after execution of `addGridPathIfOK` function. `trans->writeSet` contains all the grid points on `onePath`. After call to

commitTrans, ownership of all these points are obtained by the transactiont. Due to the properties of encoding explained above, points on the onePath are the same since the snapshot was taken. Thus, they are not a part of another path. Hence, program invariant and assertions stated in the commented lines are satisfied.

- For each statement endTrans(t), we replace the statement with endAndCleanTrans(t) in the encoded version.
- Each statement assert(p) changed to assert($\tilde{p}$) is obtained by accessing sub fields of a primitive type as obtained in the invariant of Node struct.

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

- Trans beginTrans(t) creates a Trans object trans for thread t. This function has no pre-condition and has the post-condition:

```
\forall PInt P; !trans->readSet[P] &&
          !trans->writeSet[P] &&
          trans->localIntCopy[P] == null
```

- beginWriting(V, t) is called when t wants to write V. It passes V->spanFormer->token to t.Since we are verifying only successful executions of transactions (and assuming that aborted transactions have no visible effect), we call beginWriting in the encoded program only at a state where it will successfully complete. This function has the post-condition that the owner of V->spanFormer->token is the current transaction t.
- PInt transReadInt(V,trans) reads V in a transaction that owns trans. If V has not been read or written to before, it creates a new PInt V', assigns its fields equal to V and adds V to readSet of trans. Otherwise, it returns trans->localIntCopy[V]. This function does not require V to be owned by transaction calling this function and has the post-condition that

```
t->readSet[V] == true &&
trans->localIntCopy[V] == result &&
result->vNo == V->vNo && result->value == V->value
```

- 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.
- transWriteInt(V, l,trans,t) writes the value of the local variable l to local copy of the shared variable V. If this variable has not been read or written to by this transaction before, it creates the local copy $lclV$ of which fields are equal to V and sets trans->localIntCopy[V] to $lclV$. If V has been read previously by t, then this function requires that V's version number has not changed since. Moreover, current transaction should have the ownership of V->spanFormer->token. These are expressed by the pre-condition

```
trans->readSet[V] ==> trans->localIntCopy[V]->vNo == V->vNo &&
V->spanFormer->token->owner == t
```

and the post-condition is:

```
trans->localIntCopy[V] &&
trans->writeSet[V] &&
trans->localIntCopy[V]->value == l &&
unchanged(trans->localIntCopy[V]->vNo )
```

- commitTrans(trans,t) has two phases:
  - For all $x \in$trans->writeSet, get ownership of x by exchanging it with x->spanFormer->token's ownership (call beginCommit(x,t, x->spanFormer->token)).
  - For all $x \in$trans->writeSet, modify x s.t. x->value == trans->localIntCopy[x]->value and increment x->vNo by one.

  commitTrans has a precondition that transaction t has the ownership of the tokens of the spanFormers of the objects it wants to modify:

```
\forall PInt P; trans->writeSet[P] ==> P->spanFormer->token->owner
```

  Its post-condition ensures that values of the objects has changed as desired:

```
\forall PInt P;
   trans->writeSet[P] ==>
      P->value == trans->localIntCopy[P]->value &&
      P->vNo == trans->localIntCopy[P]->vNo+1
```

- beginCommit(x,token,t) function passes ownership of x to transaction t so that it can modify it. It has precondition

```
x->spanFormer->token == token &&
token->owner == t
```

  It ensures:

```
x->owner == t
```

- endAndCleanTrans(trans,t) ends a transaction t by releasing the ownership of the objects it holds, cleaning its read and write sets. It has the following post-condition:

```
\forall PInt P; !trans->writeSet[P] &&
!trans->readSet[P] &&
trans->localIntCopy[P] == null &&
P->owner != t
```

The following theorem, the proof of which is available at states the soundness of our verification approach.

Theorem 1 (Soundness)**.** *Let P_{SI} be a transactional program and $\tilde{P_{SI}}$ be the augmented program obtained from P_{SI} as described above. Then $\tilde{P_{SI}}$ 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, provide the desired specifications, and additional proof annotations. 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.

The source-to-source code transformation 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.

3.5 Verifying Transformed Program With VCC

In this part, we explain how verification of the transformed program is performed on the grid example. For the grid, user provides the program invariant both as the precondition and postcondition of `findRoute` and specifications between lines 13-16 as postcondition for the original program.

Generally, program pre and postconditions are not enough for verification and the user may need extra ghost variables or annotations. Especially for the loops or other code blocks enclosed with curly parentheses, user should provide conditions about user defined shared or local objects that are satisfied throughout the code block and helps verification of the post conditions. Since `findRoute` does not contain such code blocks. Hence, no extra annotation is needed.

Moreover, user may need to provide extra annotations although the function does not contain any such code blocks. These annotations reflect the correctness intuition of the program. To our experience with SI, user should provide a condition that holds right after end of the read phase (after snapshot has been taken) such that this condition is preserved although other transactions interfere and modify data. In the grid example, assertions on lines 6,7 reflect the correctness intuition. `onePath` is a valid and connecting path for `localGrid` and `grid` when the snapshot was taken. It continues to hold during execution although other transactions interfere and modify `grid`. This information is enough for VCC to verify post conditions of `findRoute`: Since `onePath` is a valid and connecting path on the `localGrid` and points on the `onePath` stays the same in `grid`, `onePath` becomes a valid and connecting path after call to `addGridPathIfOK`.

Note that assertions added for verification on lines 6,7 do not include variables, fields or calls to functions introduced by the transformation. Therefore, user does not need any knowledge about transformation and these extra program parts. This is the case we encountered during the verification of examples. Correctness intuition based on local and shared user variables are enough for verification.

If initial correctness intuition is not enough for verification for function post-conditions, user may come up with tighter and stricter annotations for verification of assertions or program post-conditions until the function is verified.

4. Experiments

We applied our technique to the Genome, Labyrinth and Self-Organizing Map benchmarks from STAMP [**?**], a widely-used collection of concurrent benchmark programs containing pre-annotated transactional code blocks, and a StringBuffer pool example. All four of these examples are correct applications but their executions are not conflict serializable. The STAMP examples had been implemented in a way that is correct under SI and using programmer-defined conflict detection previously [**?**]. We made precise and formally verified the correctness arguments for these implementations and for the `StringBuffer` example.

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.

I filled the benchmarks

- **Labyrinth:** We explained and analysed this benchmark in the motivation section. We also explain verification effort in section 3.5.
- **Genome:** In this benchmark, concurrent transactions runs method `addNode` under `!WAR` semantics to add a new node to a shared linked list of which node values are in ascending order. Linked list struct has two invariants: (i) its nodes are in ascending order and (ii) linked list is not circular. `addNode` method has the post-condition that added node is reachable from the head of the linked list.

  `addNode` contains a loop in which two pointers (`prev` and `curr`) iterate over linked list until correct place for inserting new node is found. In order to satisfy post condition that new node is reachable from head node of the linked list we introduced loop invariant `prev` is reachable from the head of linked list and `prev->next == curr`. Although this statement is true for sequential execution of `addNode`, it is too strict for `!WAR` semantics since other transactions can add nodes between `prev` and `curr` during the execution of loop body. Therefore, introduced a relaxed invariant by changing second part of the condition as `curr` is reachable from the `prev`. This invariant was enough for verification of `addNode` post conditions.
- **SOM:** In this benchmark, concurrent transactions run learning phase of the machine learning algorithm SOM. SOM contains a shared grid of which nodes are n-dimensional vectors. The learning function `solve` takes an n-dimensional vector v and grid as input, calculates euclidean distance of v to each grid nodes, picks the clos-

	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

est one v' and moves nodes in a neighbourhood of v' closer to v. //I do not know what to write for what we verified

- **StringBuffer** In this example, `stringBuffer` is a shared array of which elements are pointers to objects. Concurrent transactions run `Allocate` function which iterates over `stringBuffer`, sets first non-null pointer to null and returns the element or `Free` function which iterates over `stringBuffer` and sets first null pointer it sees to the element it wants to free, under `!WAR` semantics.

  We verified for the `Allocate` function that after transaction commits successfully, index of the `stringBuffer` shown by the iterator is null and for the `Free` function that after transaction commits successfully, index of the `stringBuffer` shown by the iterator points to the object given in the input.

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.

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. [**?**] 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 [**? ? ? ? ? ?**]. 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 [**?**]. To prove linearizability of a transactional program P running under SI, one could use the encoded program we construct, $\tilde{P}$ 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. [**?**] 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. [**?**] 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. First, on the transformation, the thread, object and procedure structure of the original program is preserved. 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. We also have the practically important advantage that while verifying his code under SI, 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 [**?**] 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).