A Transactional Correctness Tool for Abstract Data Types

CHRISTINA PETERSON and DAMIAN DECHEV, University of Central Florida

Transactional memory simplifies multiprocessor programming by providing the guarantee that a sequential block of code in the form of a transaction will exhibit atomicity and isolation. Transactional data structures offer the same guarantee to concurrent data structures by enabling the atomic execution of a composition of operations. The concurrency control of transactional memory systems preserves atomicity and isolation by detecting read/write conflicts among multiple concurrent transactions. State-of-the-art transactional data structures improve on this concurrency control protocol by providing explicit transaction-level synchronization for only non-commutative operations. Since read/write conflicts are handled by thread-level concurrency control, the correctness of transactional data structures cannot be evaluated according to the read/write histories. This presents a challenge for existing correctness verification techniques for transactional memory, because correctness is determined according to the transitions taken by the transactions in the presence of read/write conflicts.

In this article, we present Transactional Correctness tool for Abstract Data Types (TxC-ADT), the first tool that can check the correctness of transactional data structures. TxC-ADT elevates the standard definitions of transactional correctness to be in terms of an abstract data type, an essential aspect for checking correctness of transactions that synchronize only for high-level semantic conflicts. To accommodate a diverse assortment of transactional correctness conditions, we present a technique for defining correctness as a happens-before relation. Defining a correctness condition in this manner enables an automated approach in which correctness is evaluated by generating and analyzing a transactional happens-before graph during model checking. A transactional happens-before graph is maintained on a per-thread basis, making our approach applicable to transactional correctness conditions that do not enforce a total order on a transactional execution. We demonstrate the practical applications of TxC-ADT by checking Lock Free Transactional Transformation and Transactional Data Structure Libraries for serializability, strict serializability, opacity, and causal consistency.

CCS Concepts: • **Computing methodologies** → **Concurrent programming languages**; **Concurrent algorithms**;

Additional Key Words and Pharses: Concurrency, correctness verification, transactional data structure

ACM Reference format:
Christina Peterson and Damian Dechev. 2017. A Transactional Correctness Tool for Abstract Data Types. *ACM Trans. Archit. Code Optim.* 14, 4, Article 37 (November 2017), 24 pages.
https://doi.org/10.1145/3148964

1 INTRODUCTION

Advancements in multiprocessor programming remains at the forefront of research and development to leverage the processing power of multi-core systems. The synchronization of shared

This work was funded by the National Science Foundation (NSF) under Grant Numbers NSF OAC 1440530, NSF CCF 1717515, and NSF OAC 1740095.
Authors' addresses: C. Peterson and D. Dechev, University of Central Florida, 4000 Central Florida Blvd. Orlando, FL 32816 USA; emails: clp8199@knights.ucf.edu, dechev@cs.ucf.edu.

https://doi.org/10.1145/3148964

memory accesses such that safety and liveness properties are preserved is an inherent challenge associated with multiprocessor algorithms. Safety ensures that an algorithm is correct with respect to a defined correctness condition while liveness ensures that the program threads terminate according to a defined progress guarantee.

The difficulty of designing multiprocessor programs inspired transactional memory, a programming paradigm introduced by Herlihy et al. [19], that would allow designers to write sequential blocks of code that could be executed concurrently while preserving the safety and liveness properties expected from multiprocessor programs. In addition to safety and liveness properties of traditional multiprocessor programs, transactional memory is expected to exhibit for each transaction (1) atomicity and (2) isolation. Atomicity guarantees that the effects of a transaction are either entirely committed to memory or are entirely aborted. Isolation guarantees that the operations of each transaction appear to take effect in an uninterrupted sequential order. These properties have broad applicability, since they enable the atomic execution of a composition of operations.

A concurrent data structure is considered *transactional* if it supports executing operations atomically and in isolation. Transactional data structures bring the benefits of transactional memory to concurrent data structures by enabling the execution of a composition of operations. The concurrency control employed by transactional memory systems [8, 9, 24] preserves atomicity and isolation by detecting accesses to the same memory location among multiple concurrent transactions. Conflicts on memory accesses are resolved by allowing one transaction to commit while the others are aborted or delayed, referred to as *transactional synchronization*. This concurrency control protocol suffers a performance loss for transactional data structures when recurring conflicts arise on frequently accessed memory locations such as the head of a linked list.

Commutative operations are operations that when executed in opposite order will yield the same abstract state of the data structure. *Non-commutative operations* are operations that when executed in opposite order will yield a different abstract state of the data structure. State-of-the-art transactional data structures [18, 34, 38] improve the concurrency control of transactional memory systems by leveraging semantic knowledge of the data structure to provide explicit transactional synchronization for only non-commutative operations. This high-level semantic conflict detection allows commutative operations to proceed concurrently by utilizing atomic read, atomic write, and atomic read-modify-write (RMW) operations for the thread-level synchronization of low-level read/write conflicts [18]. The exploitation of data structure semantics substantially improves performance but comes with a subtle catch. The correctness of transactional data structures cannot be judged according to the histories of low-level reads and writes. Since transactional data structures do not synchronize transactions for read/write conflicts, the read/write histories do not exhibit the isolation property expected from transactional memory systems. This presents a challenge for verification techniques [2, 6, 11, 12, 14] that evaluate the correctness of transactional memory systems based on low-level reads and writes.

In this article, we present Transactional Correctness tool for Abstract Data Types (TxC-ADT), the first tool that can check the correctness of transactional data structures. As introduced in Reference [33], TxC-ADT allows the standard definitions of transactional correctness to be recast in terms of an abstract data type. Correctness is evaluated based on the abstract data type history rather than the read/write history. TxC-ADT accommodates a variety of widely accepted transactional correctness conditions, including serializability [29], strict serializability [29], opacity [17], and causal consistency [21].

We address several challenges to unify a diverse collection of correctness conditions applicable to transactional data structures. First, the concurrent histories must be in terms of an abstract data type. We address this challenge by providing the user with lightweight annotations that identify the invocation and response of an abstract data type. We use the model checker CDSChecker [27]

to iterate through all possible interleavings of the transactional application and generate the concurrent histories based on the defined abstract data type.

Second, the legal sequential histories that define the allowable histories vary for each correctness condition. We address this challenge by defining correctness through a happens-before relation on transactions using a custom specification language. TxC-ADT automatically constructs a transactional happens-before graph that represents the allowable ordering of the transactions based on the happens-before relation. The legal sequential histories that represent correct behavior for the concurrent history are automatically extracted from the graph through a recursive topological sort algorithm. The advantage of deriving the legal sequential histories from a transactional happens-before graph is that the atomicity and isolation properties are preserved in the legal sequential histories. We optimize the recursive topological sort by pruning a reordering of transactions that are commutative from the search space. The exploration of a reordering of commutative transactions is redundant, because the transactions executed in either order will yield the same abstract state.

Third, transactional correctness conditions do not necessarily enforce a total order on a transactional execution. Causal consistency is one such example in which transactions may be perceived in a different order by each thread. We address this challenge by allowing the happens-before relation to be defined on a per-thread basis, in which a transactional happens-before graph will be constructed for each individual thread. The generated legal sequential histories will therefore reflect the observed history for each individual thread.

TxC-ADT checks the correctness of a transactional data structure by automatically generating all possible concurrent histories from a transactional program and verifying that each concurrent history is equivalent to a legal sequential history in terms of an abstract data type according to the defined correctness condition. The ability to check correctness in terms of an abstract data type is essential as transactional data structures become mainstream in database [1, 26, 35] and data analysis [36, 37] applications that require atomicity and isolation for a composition of operations. TxC-ADT will impact multiprocessor designers that are seeking to deliver high-performance transactional capabilities that maintain the correctness properties expected from a transactional program while benefiting from a high-level semantic conflict detection protocol. We demonstrate the practical applications of TxC-ADT by checking the correctness of the transactional data structures presented by Zhang et al. [38] and Spiegelman et al. [34].

This article makes the following contributions:

(1) We present the first tool that can check the correctness of transactional data structures. We evaluate correctness based on an abstract data type, making our approach applicable to transactional data structures that use a high-level semantic conflict detection. Existing correctness verification tools for transactional memory systems evaluate correctness based on the low-level read/write histories, making these techniques impractical for state-of-the-art transactional data structures.
(2) We present a technique for representing a transactional correctness condition as a happens-before relation. The main advantage of this technique is that it enables a diverse assortment of correctness conditions to be checked automatically by generating and analyzing a transactional happens-before graph during model checking. Furthermore, this technique enables TxC-ADT to be adaptable to other transactional correctness conditions that may become prevalent in the advancement of transactional data structures.
(3) We present an optimization to the recursive topological sort of the transactional happens-before graph that prunes a reordering of transactions that are commutative from the search space. We accomplish this by allowing the user to specify the conditions for which two operations commute.

(4) We present a strategy for checking the correctness of a transactional data structure when the designed correctness condition does not enforce a total order on a history. Serializability, strict serializability, and opacity require a total order on the history such that all threads observe the transactions in the same order. However, causal consistency requires only a partial order on a history, allowing threads to observe transactions in a different order. To the best of our knowledge, this is the first verification strategy capable of checking a transactional memory system for causal consistency.
(5) We present two case studies demonstrating the practical application of TxC-ADT to check the correctness of state-of-the-art transactional data structures, including Lock-Free Transactional Transformation [38] and Transactional Data Structure Libraries [34].

2 RELATED WORK

A significant amount of research focuses on the correctness verification of transactional memory. Several approaches [11, 12, 22] propose automatic techniques to verify correctness of transactional memory systems. Flanagan et al. [12] present the dynamic analysis tool Velodrome that performs atomicity verification that is both sound and complete. Velodrome analyzes operation dependencies within atomic blocks and infers the transactional happens-before relations of an observed execution trace. Serializability of the execution trace is determined by verifying that the transactional happens-before graph is acyclic. Emmi et al. [11] present an automatic verification method to check that transactional memories meet the correctness property strict serializability. Their technique parameterizes a transactional memory implementation according to the number of threads n and number of shared locations k by constructing a family of simulation relations that demonstrates for all $n > 0$ and $k > 0$, the transactional memory implementation refines the strict serializability specification. Litz et al. [22] present a tool that automatically corrects snapshot isolation (SI) anomalies in transactional memory programs. The tool promotes dangerous read operations in the conflict detection phase of the SI transactional memory implementation and forces one of the affected transactions to abort. The authors reduce the problem of choosing the read operation to be promoted to a graph coverage problem for a dependency graph focusing on read operations. Since these techniques verify correctness based on the low-level read/write histories of the transactions, they are not directly applicable to transactional data structures that utilize high-level semantic conflict detection.

Model checking is a well-known technique for checking correctness properties of concurrent programs. The model checker CHESS [25] enables the systematic and deterministic testing of concurrent programs. Binary instrumentation is provided between the test program and the concurrency API to explore the possible thread schedules. CDSChecker [27] enables the exploration of thread schedules that use the relaxed semantics of the C/C++ memory model, which utilizes a variation of the dynamic partial order reduction [13] technique to minimize the exploration of redundant thread schedules. Line-Up [5], a tool that automatically checks deterministic linearizability, uses CHESS [25] to produce all sequential histories of a finite test and checks that all concurrent histories are consistent with the sequential histories. While Line-Up is designed for checking correctness of non-blocking data structures, the general approach of comparing concurrent histories with sequential histories to evaluate correctness is utilized by TxC-ADT.

Approaches including those in References [2, 14–16, 28] propose techniques based on model checking to verify correctness of transactional memory systems. Guerraoui et al. [14] present a technique for verifying software transactional memory (STM) safety properties using model checking. Their technique leverages the structural symmetries of STM algorithms to reduce the verification problem of an unbounded STM state space to a finite-state verification problem that requires a small number of threads and shared variables. O'Leary et al. [28] verify the

correctness of Intel's McRT STM [31] using the model checker Spin [20]. Baek at al. [2] present *ChkTM*, a model checking environment that can verify the correctness of transactional memory systems. ChkTM checks serializability and strong isolation of a transactional memory system by performing a coarse-grained state space exploration that records the transactional reads and writes when only a single processor is active at a time and comparing the result to a fine-grained state space exploration that records the memory accesses for all possible interleavings. These approaches verify correctness at the granularity of low-level reads and writes, so the correctness checking algorithm of these approaches needs to be modified to account for a concurrent history in terms of an abstract data type to be relevant for the high-level semantic conflict detection of transactional data structures.

Many approaches [3, 4, 6, 7, 10, 23, 32] propose a formal logic to verify correctness of transactional memory systems. Blundell et al. [4] demonstrate that a direct conversion of lock-based critical sections into transactions can cause deadlock even if the lock-based program is correct. The observations of Blundell et al. [4] highlights safety violations that may be introduced in transactional programs but does not provide a methodology for detecting the resulting faulty behavior. Cohen et al. [6] present an abstract model for specifying transactional memory semantics, a proof rule for verifying that the transactional memory implementation satisfies the specification, and a technique for verifying serializability and strict serializability for a transactional sequence. Since conflicts considered in the abstract model are defined at the read/write level, the approach is limited to transactional memory systems that synchronize at low-level reads and writes. Manovit et al. [23] present a framework of formal axioms for specifying legal operations of a transactional memory system. The dynamic sequence of program instructions called in the test are converted to a sequence of nodes in a graph, where an edge in the graph represents constraints on the memory order. The analysis algorithm constructs the graph based on the Total Store Order (TSO) memory model ordering requirements and checks for cycles to determine order violations. The graph construction is based on TSO ordering requirements, so the framework cannot be directly used to verify transactional correctness conditions that utilize high-level semantic conflict detection.

Bieniusa et al. [3] provide a formalization of a semantics of transactional memory that can prove properties of a transactional memory system. The semantics are based on low-level reads/writes and does not account for high-level semantic conflict detection. Doherty et al. [10] present Transactional Memory Specification 1 (TMS1), a correctness specification of a transactional memory runtime library comprising transactional features in programming languages such as C or C++. TMS1 is specified using an I/O automaton, enabling formal and machine-checked correctness proofs of transactional memory implementations. The advantage of TxC-ADT over this verification technique is that TxC-ADT is capable of automatically checking a correctness condition specification while Doherty et al. [10]'s approach requires that the correctness proofs be constructed manually using formal logic. Schmidt-Schauß et al. [32] present the specification calculus STM-Haskell with Futures (SHF) and a concurrent implementation of SHF, referred to as CSHF. The CSHF specification is proved correct by showing that it is semantically equivalent to the big-step reduction defined for SHF. To extend the approach to be applicable to transactional data structures, updates are necessary for the SHF and CSHF calculus syntax and reduction rules to account for a user-specified abstract data type and the transaction log maintained in CSHF to abort transactions for access conflicts on the abstract data type.

3 METHODOLOGY

State-of-the-art transactional data structures [18, 34, 38] deliver improved performance over data structures built using traditional transactional memory systems by performing transactional synchronization only for high-level semantic conflicts. To achieve these performance benefits,

```
1  template<typename T>
2  T Method(T x){
3   begin(&Method, x);
4   //Method body
5   end(&Method, y);
6   return y;
7  }
8
9  void thread_body()
10 {
11   int Input1, Input2;
12   txn_begin();
13   Method(Input1);
14   Method(Input2);
15   txn_end();
16 }
```

Fig. 1. Abstract data type annotation example.

transactional data structures must abandon the isolation property at the granularity of low-level reads and writes. This presents a challenge for previous correction verification techniques [2, 6, 11, 12, 14] for transactional memory systems, because the transitions incorporated in the correctness proofs exhibit actions taken by the transactions in the presence of read/write conflicts.

3.1 Transactional Correctness for Abstract Data Types

We first provide definitions relevant to a transactional execution. An *event* is (1) an operation associated with changes in the status of a transaction, including transaction-begin, commit, or abort, or (2) an operation associated with a method invocation or method response. A *history* is a finite series of instantaneous events [18]. A *sequential history* is a history such that the events of a transaction run in isolation from the events of other transactions [18]. A *concurrent history* is a history in which the finite series of events are ordered according to a thread schedule.

The correctness of transactional data structures is evaluated by elevating the standard definitions of transactional correctness to be properties on an abstract data type [33]. We provide the user with lightweight annotations to indicate the invocation and response of a method invoked on an abstract data type and use the model checker CDSChecker [27] to generate the concurrent histories based on the defined abstract data type. An example of the annotation usage is shown in Figure 1. The annotations are displayed in C-like syntax, because it demonstrates how to specify an abstract data type using TxC-ADT. The invocation of a method is specified by passing a function pointer of the method and associated input to the `begin` function on line 3. The response of a method is specified by passing a function pointer of the method and associated output to the `end` function on line 5. A transactional region is specified using the `txn_begin` function on line 12 to indicate the beginning of a transaction and the `txn_end` function on line 15 to indicate the end of a transaction.

Our approach for specifying an abstract data type can be applied to reads and writes for the verification of legacy transactional memory systems. The read/write operations need to be enclosed between the `begin` function on line 3 and `end` function on line 5 with the appropriate parameters passed to each function. This can be elegantly handled using macros for the read and write operations. TxC-ADT will then evaluate correctness based on the read/write histories of the transactions.

During model checking, the information extracted from each annotation is stored in an action object, shown in Algorithm 1. The *ConcurrentHistory* type on line 1.36 is a list of action objects that represents a single generated concurrent history. To facilitate the correctness checking algorithm presented in Algorithm 4, we assemble a transaction descriptor for each transaction in a concurrent history. An *active* status indicates that a transaction is live and has not yet committed or aborted. A *committed* status indicates that a transaction has completed and its effects are committed to memory. An *aborted* status indicates that a transaction has completed and its effects are rolled back. A transaction descriptor contains the transaction status (active, committed, or aborted), a sequence number for the beginning and ending of a transaction, and a list of the methods invoked

on the abstract data type with a corresponding function pointer and associated input and observed output values, as shown in Algorithm. 1. We now provide definitions that are fundamental for the correctness checking strategy used by TxC-ADT:

Definition 3.1. The *happens-before* relation, denoted $<_H$, is a partial order defined over the set of transactions in a history h such that for any two transactions T_1 and T_2, if $T_1 <_H T_2$, then the commit or abort event of transaction T_1 precedes the commit or abort event of transaction T_2 in history h.

Definition 3.2. The *transactional happens-before graph* is a directed graph such that for any two transactions T_1 and T_2 in history h, if an edge exists from T_1 to T_2, then $T_1 <_H T_2$.

The *txn_map* on line 1.34 maps each transaction to a unique identification number to maintain the transactional happens-before graph as a two-dimensional list of transaction ids, as shown on line 1.35. The *LegalHistory* type on line 1.37 is a list of transaction identification numbers that corresponds to a legal ordering of the transactions according to the correctness condition. Correctness is evaluated by comparing the abstract data type methods list output values to the legal sequential histories, discussed in Section 3.2.

If the transaction status is aborted, then a list of inverse operations is maintained on line 1.33 to undo the effects of the operations to the abstract data type. This undo log is necessary to verify correctness conditions, such as opacity, that require aborted transactions to observe a consistent state of the system. When correctness is judged on aborted transactions, the generated legal sequential history must include the observed output from the aborted transaction. However, the inverse operations must be called immediately after invoking all operations for the aborted transaction so its effects do not propagate throughout the remaining generated legal sequential history.

ALGORITHM 1: Type Definitions

```
1  typedef int TxnId;
2  enum TxStatus
3      Active;
4      Commited;
5      Aborted;
6  enum ActionType
7      Invocation;
8      Response;
9      Correctness_Condition;
10     Txn_Begin;
11     Txn_End;
12     Commit;
13     Abort;
14 struct MethodDesc
15     int id;
16     void *(func_ptr)(uint64_t);
17     uint64_t input;
18     uint64_t observedOutput;
19 struct ActionObject
20     int sequence_number;
21     ActionType type;
22     int tid;
23     TxStatus status;
24     TxnId txn_id;
25     void *(method)(uint64_t);
26     uint64_t input;
27     uint64_t observedOutput;
28 struct TxnDesc
29     TxStatus status;
30     int begin;
31     int end;
32     List<MethodDesc> method_list;
33     List<MethodDesc> method_list_inv;
34 Map<TxnId, TxnDesc> txn_map;
35 typedef List<List<TxnId>> Graph;
36 typedef List<ActionObject> ConcurrentHistory;
37 typedef List<TxnId> LegalHistory;
```

```
1    int main() {
2      correctness_condition(...);
3      //Spawn threads for unit test
4    }
```

Fig. 2. Correctness condition declaration.

ALGORITHM 2: Recursive Topological Sort

```
1 Function RecTopologicalSort(Graph g)
2     list L ;                                  // Empty list that contains sorted transactions
3     list N ;                                  // List of all transactions with no incoming edges
4     LegalHistory S[] ;
5     foreach n ∈ N do
6         PrunedRecTopologicalSort(S[], n, L, N, g) ;
7     return S[] ;                              // Return all legal sequential histories
```

3.2 A Unification of Transactional Correctness Conditions

To check that the generated concurrent histories are correct, each concurrent history must be equivalent to a legal sequential history based on a transactional correctness condition. As transactional data structures become widespread, the diverse assortment of transactional correctness conditions will be potential candidates for delivering a design that provides the safety expected from multiprocessor algorithms. For this reason, we designed TxC-ADT to accommodate well-known transactional correctness conditions including serializability, strict serializability, opacity, and causal consistency.

TxC-ADT unifies the transactional correctness conditions by observing that the ordering constraints on the transactions according to the correctness conditions can be represented by a transactional happens-before graph. The idea of a transactional happens-before graph was used in Velodrome [12]. However, Velodrome's graph is constructed by automatically inferring the happens-before relationship between transactions from the low-level read/write orderings, which is not applicable to transactional data structures that use a high-level semantic conflict detection.

Our strategy is to define for each correctness condition a happens-before relation on transactions using a custom specification language. The definition for the correctness condition is placed in the main method using the `correctness_condition` function as shown on line 2 of Figure 2. The specifications for serializability, strict serializability, opacity, and causal consistency are presented in Section 3.3. A unit test for the transactional data structure must declare the main entry point as `user_main(int, char**)` instead of `main(int, char**)` and use CDSChecker's threads library and the C++ Atomic Operations Library for atomic operations. Once the happens-before relation is defined, TxC-ADT automatically constructs a transactional happens-before graph during model checking. A topological sort of the happens-before graph will yield a possible legal sequential history for the transactional execution. We can derive all possible legal sequential histories for the transactional execution by applying a recursive topological sort to the transactional happens-before graph.

Algorithm 2 presents the recursive topological sort function. The worst case time complexity of a recursive topological sort is $O(n!)$ due to the consideration of all possible orderings of n transactions. We observe that this time complexity can be reduced by pruning the recursive topological sort to not explore ordering variations for commutative transactions.

ALGORITHM 3: Pruned Recursive Topological Sort

```
1  Function PrunedRecTopologicalSort(LegalHistory S[], TxnId n, list L, list N, Graph g)
2      if (L.size() != 0) && (commutes_matrix[n][L.back()] == true) && (reorder_matrix[n][L.back()] ==
         true) && (n < L.back()) then
3          return ;          // Prune redundant recursive call for commutative transactions
4      Graph g' = g;
5      L.push_back(n) ;                                  // Add n to list of sorted transactions
6      N.remove(n);
7      foreach m ∈ g'[n] do
8          g'[n].remove(m);
9          if m.incoming_edges() == 0 then
10             N.push_back(m) ;     // Add m to list of transactions with no incoming edges
11     foreach n' ∈ N do
12         PrunedRecTopologicalSort(S[], n', L, N, g') ;
13     if N.size() == 0 then
14         S[].push_back(L);
15     return;
```

Algorithm 3 presents the pruned recursive topological sort function called within Algorithm 2. The pruned recursive topological sort function is passed a list of legal sequential histories $S[]$, a transaction id n to select as the next ordered transaction, a list L that contains the sorted transactions, a list N of transactions with no incoming edges, and a graph g. The *commutes_matrix* is a Boolean two-dimensional matrix where position (i, j) is true if transaction i and transaction j commute and false otherwise. The *reorder_matrix* is a Boolean two-dimensional matrix where position (i, j) is true if transaction i and transaction j have no ordering constraints and false if transaction i and transaction j are ordered by the happens-before relation. Both matrices are constructed based on the correctness condition specification, described in Section 3.3. If transaction n commutes with the last transaction in list L and these transactions can be reordered, then the orderings in which $n < L.back()$ will not be explored. Alternatively, we could have also chosen to not explore the orderings in which $L.back() < n$. If pruning is not possible, then all edges m outgoing from transaction n are removed from an updated graph g' on line 3.8 and the pruned recursive topological sort function is called on the updated list N of transactions with no incoming edges.

The algorithm for checking a correctness condition is presented in Algorithm 4. The `IsHistoryCorrect` function generates the transactional happens-before graph on line 4.2 from the *ConcurrentHistory* object in conjunction with the correctness condition specification. A recursive topological sort on the graph, shown on line 4.3, computes all possible orderings of the transactions. For each possible transaction ordering, the concurrent output and sequential output are generated from the transaction descriptor *TxnDesc* detailed in Algorithm 1. For each method in a transaction's *method_list*, the observed output is amended to the concurrent history on line 4.9, and the method's function pointer is invoked on line 4.10 and amended to the sequential output on line 4.11. If a transaction does not commit, then the function pointers of the inverse methods in *method_list_inv* are invoked to undo the effects of the transaction in the remainder of the legal sequential history. The order of the inverse methods in *method_list_inv* is the reverse order of the corresponding methods in *method_list*. This is essential to restore the correct abstract state for non-commutative operations [18].

ALGORITHM 4: Correctness Checking Algorithm for Concurrent History

```
1  Function IsHistoryCorrect(ConcurrentHistory* history)
2      Graph g ← GenerateGraph(history) ;   // Generate transactional happens-before graph
3      LegalHistory S[] ← RecTopologicalSort(g) ;      // Generate set of legal sequential
        histories
4      foreach s ∈ S[] do
5          list concurrent_output;
6          list sequential_output;
7          foreach txn ∈ s do
8              foreach m ∈ txn.method_list do
9                  concurrent_output.push_back(m.observedOutput) ;      // m's observed output
10                 uint64_t temp = (*m.func_ptr)(m.input) ;            // Invoke m sequentially
11                 sequential_output.push_back(temp) ;                 // m's sequential output
12             if txn.status!=COMMITTED then
13                 foreach m_inv ∈ txn.method_list_inv do
14                     (*m_inv.func_ptr)(m_inv.input) ;            // Invoke m_inv sequentially
15         if concurrent_output==sequential_output then
16             return true ;       // Concurrent output is equivalent to a legal sequential
                history
17         else
               // Document counterexample
       // Report all documented counterexamples
18     return false ;  // Concurrent output is not equivalent to a legal sequential history
```

The concurrent history output is compared with the legal sequential history output on line 4.15. If this comparison is true, then the individual concurrent history is correct and `IsHistoryCorrect` returns true. Otherwise, the counterexample is documented and the for-loop on line 4.4 continues to iterate through the possible legal sequential histories. If the concurrent history output is not equivalent to any legal sequential history output, then the concurrent history is not correct. `IsHistoryCorrect` returns false and the counterexamples collected are reported to the user at the end of model checking.

The derivation of the legal sequential histories from a transactional happens-before graph preserves two critical properties expected from a transactional execution: atomicity and isolation. The transactions that appear in a legal sequential history are executed entirely (allowing for the verification of atomicity) and in a one-at-a-time sequential order (allowing for the verification of isolation). Moreover, since all transactional correctness conditions preserve atomicity and isolation, our approach may be extended to other correctness conditions that may be adopted in the advancement of transactional data structures.

The comparison between legal sequential history and concurrent history is performed by comparing the observed effects of each individual method in the concurrent history to the observed effects of the corresponding method in the legal sequential history. The comparison is performed in this manner rather than directly comparing the concurrent history output in the order of observation with the output of each legal sequential history, because the order in which the method response occurs in the concurrent history may appear to violate isolation. Since commutative operations do not require transactional synchronization, the concurrent history may reflect a method

ALGORITHM 5: Correctness Checking Algorithm for Implementation

```
1 Function IsUnitTestCorrect(UnitTest m)
2     ConcurrentHistory H[] ← GenerateConcurrentHistories(m) ;    // Generate concurrent
        histories from unit test
3     bool outcome = true;
4     foreach h ∈ H[] do
5         if IsHistoryCorrect(h) == false then
6             outcome = false;
7     return outcome ;    // At least one concurrent history is not equivalent to a legal
        sequential history
```

response ordering in which the effects of a transaction are interleaved with the effects of another transaction. This is acceptable behavior for transactional data structures, because the interleaved effects of commutative operations result in the same abstract state.

The algorithm for checking the correctness of a unit test is presented in Algorithm 5. The `IsUnitTestCorrect` function accepts a unit test as a parameter and generates all concurrent histories of the unit test from a model checker on line 5.2. The foreach-statement on line 5.4 iterates through all concurrent histories and checks if each concurrent history is correct. If all concurrent histories are correct, then the unit test meets the transactional correctness condition; otherwise, the unit test does not meet the transactional correctness condition. Since correctness is judged only on the generated concurrent histories, the outcome of the `IsUnitTestCorrect` function is relevant only to the unit test. To check that the implementation is correct, the unit test must be written to include all methods and a minimal set of inputs such that all behaviors of the transactional data structure are explored.

3.2.1 Correctness of TxC-ADT Technique. We provide a formal definition of commutativity between transactions as follows.

Definition 3.3. Two transactions T_1 and T_2 *commute* if for all histories h if $h \cdot T_1$ and $h \cdot T_2$ are both legal, then $h \cdot T_1 \cdot T_2$ and $h \cdot T_2 \cdot T_1$ are both legal and define the same abstract state.

Theorem 3.4. *Let transaction i and transaction j be commutative and allowed to be reordered according to the transactional happens-before graph. Let h be all possible histories generated by a topological sort of the transactional happens-before graph. Algorithm 3 will explore $h \cdot i \cdot j$ and terminate the* `PrunedRecTopologicalSort` *call for $h \cdot j \cdot i$.*

Proof. Let *commutes_matrix* be a Boolean two-dimensional matrix where position (i, j) is true if transaction i and transaction j commute and false otherwise. Let *reorder_matrix* be a Boolean two-dimensional matrix where position (i, j) is true if transaction i and transaction j have no ordering constraints and false if transaction i and transaction j are ordered by the transactional happens-before graph. Let L be a partial list of transactions that are sorted according to the transactional happens-before graph. Let N be a list of all transactions with no incoming edges. Let $n \in N$ be a transaction that is under consideration for being amended to the end of L. By Definition 3.3, if $L.back()$ and n commute, then these transactions executed in either order will yield the same abstract state. If $L.back()$ and n are allowed to be reordered according to the transactional happens-before graph, then it is only necessary to explore the ordering $L.back() \cdot n$ or $n \cdot L.back()$. Arbitrarily choose to explore the ordering in which transaction id n is greater than transaction id $L.back()$, denoted $n > L.back()$. Given that L is not empty, $commutes\_matrix[n][L.back()]$ is true,

and $reorder\_matrix[n][L.back()]$ is true, then the orderings such that $n < L.back()$ do not need to be explored. Let transaction i have a smaller id than transaction j. Since the if-statement on line 3.2 will resolve to true if $n < L.back()$, $L.back()$ and n commute, and $L.back()$ and n can be reordered according to the transactional happens-before graph, the history $h \cdot i \cdot j$ will be explored and the `PrunedRecTopologicalSort` call will terminate for $h \cdot j \cdot i$. □

Theorem 3.5. `RecTopologicalSort` *returns a set S of all legal sequential histories defined by the transactional happens-before graph.*

Proof. Let N be a list of all transactions with no incoming edges in the transactional happens-before graph. Any selection of $n \in N$ will yield a valid topological sort of the transactional happens-before graph. The foreach-statement on line 2.5 calls `PrunedRecTopologicalSort` with $n \in N$ as a parameter. Since L is initially empty, n is amended to the back of L and removed from N. All transactions m with an edge from n to m are removed from the transactional happens-before graph on line 3.8. If m has no incoming edges, then it is amended to N on line 3.10. Any selection of $n' \in N$ will yield a valid topological sort of the happens-before graph. The foreach-statement on line 3.11 calls `PrunedRecTopologicalSort` with $n' \in N$ as a parameter. Since L is not empty, n' may possibly be pruned from the recursive topological sort on line 3.2. By Theorem 3.14, `PrunedRecTopologicalSort` will only explore one ordering of commutative transactions that are allowed to be reordered according to the transactional happens-before graph. Given that n' is not pruned from the recursive toplological sort, n' is amended to the back of L and removed from N. The recursive calls terminates when N is empty and amends the topological sort L to S on line 3.4. Since all possible orderings are considered for exploration and the pruned orderings will produce the same abstract state as another topological sort $L \in S$ by Theorem 3.4 and Definition 3.3, set S will contain all legal sequential histories defined by the transactional happens-before graph. □

Theorem 3.6 (Soundness). *Let h be a concurrent history of implementation X. If h does not satisfy the specified correctness condition, then* `IsHistoryCorrect(h)` *(Algorithm 4) returns false.*

Proof. By Theorem 3.5, `RecTopologicalSort` returns a set S of all legal sequential histories defined by the transactional happens-before graph. The foreach-statement on line 4.4 iterates through every legal sequential history in S. The foreach-statements on line 4.7 and line 4.8 iterate through each method called by each transaction in legal sequential history $s \in S$. For each method the observed output from the concurrent history is amended to the list *concurrent_output* on line 4.9. The method's function pointer is invoked on line 4.10 to generate the sequential output, which is amended to the list *sequential_output* on line 4.11. Aborted transactions are accounted for by invoking the inverse method's function pointer on line 4.14. If the if-statement on line 4.15 evaluates to true, then concurrent history h is equivalent to a legal sequential history generated from the transactional happens-before graph. Since the transactional happens-before graph represents the allowable orderings of the transactions according to the specified correctness condition, concurrent history h satisfies the specified correctness condition and `IsHistoryCorrect`(h) returns true. If the end of the for-loop on line 4.4 is reached, then `IsHistoryCorrect`(h) returns false. In this case, since concurrent history h is not equivalent to any legal sequential history generated from the transactional happens-before graph, h does not satisfy the specified correctness condition. Therefore, the theorem holds. □

Theorem 3.7 (Completeness). *Let H be the set of concurrent histories generated from a unit test m of implementation X. If for any arbitrary h ∈ H* `IsHistoryCorrect(h)` *(Algorithm 4) returns false, then implementation X does not satisfy the specified correctness condition and* `IsUnitTestCorrect` *(Algorithm 5) returns false.*

```
1   <program> ::= <function>
2   <function> ::= <function> <stmt> | /* NULL */
3   <stmt> ::= ';'
4           | <expr> ';'
5           | <variable> '=' <expr> ';'
6           | <expr> 'happens_before' <expr> ';' //Places a happens-before relation between two transactions
7           | 'insert_txn' <expr> ';' //Inserts transaction in the happens-before graph
8           | <expr> 'happens_before_partial' '[' <expr> ']' <expr> ';' //Places a per-thread happens-before
                ↪ relation between two transactions
9           | 'insert_txn_partial' '[' <expr> ']' <expr> ';' //Inserts transaction in a per-thread happens-
                ↪ before graph
10          | <expr> 'commutes_with' <expr> ',' '(' <expr> ')' ';'
11          | 'if' '(' <expr> ')' <stmt>
12          | 'if' '(' <expr> ')' <stmt> 'else' <stmt>
13          | 'for' '(' <stmt> <expr> ';' <stmt_partial> ')' <stmt>
14          | '{' <stmt_list> '}'
15  <stmt_list> ::= <stmt> | <stmt_list> <stmt>
16  <stmt_partial> :: = <variable> '=' <expr> | <variable> '++' | '++' <variable>
17                  | <variable> '--' | '--' <variable>
18  <expr> ::= <integer>
19          | <variable>
20          | <expr> <operator> <expr>
21          | 'history' '->' 'size' '(' ')' //Retrieves the length of the concurrent history
22          | 'txn' '(' <expr> ')' //Retrieves transaction id associated with sequence number <expr>
23          | 'txn_tid' '(' <expr> ')' //Retrieves the thread id associated with sequence number <expr>
24          | 'txn_status' '(' <expr> ')' //Retrieves the transaction status associated with sequence number <
                ↪ expr>
25          | <expr> 'precedes' <expr> //Returns 1 if a transaction with id <expr> is ordered before
                ↪ another transaction with id <expr> in real-time; otherwise, returns 0
26          | <expr> 'causes' <expr> //Returns 1 if a transaction with id <expr> causes the effects of
                ↪ another transaction with id <expr>; otherwise, returns 0
27          | 'method' '(' <expr> ')' //Retrieves the method id associated with sequence number <expr>
28          | 'input' '(' <expr> ')' //Retrieves the method input associated with sequence number <expr>
29          | '(' <expr> ')'
30  <integer> ::= 0 | [1-9][0-9]*
31  <operator> ::= [+-*/<>] | '>=' | '<=' | '!=' | '==' | '&&' | '||'
32  <variable> ::= [a-zA-Z][a-zA-Z0-9_]*
```

Fig. 3. Grammar for the custom specification language.

Proof. If there exists some concurrent history $h \in H$ such that `IsHistoryCorrect`(h) returns false, then by Theorem 3.6, h does not satisfy the specified correctness condition. An implementation X satisfies the specified correctness condition with respect to unit test m if for all $h \in H$, `IsHistoryCorrect`(h) returns true. If given an arbitrary $h \in H$ such that `IsHistoryCorrect`(h) returns false, then the Boolean *oucome* is set to false on line 5.6. Implementation X does not satisfy the specified correctness condition and `IsUnitTestCorrect` (Algorithm 5) returns false. Therefore, the theorem holds. □

3.3 Specification Language

The context-free grammar for our custom specification language, presented in Figure 3, is described using the Backus-Naur form (BNF). Terminals are integers (line 30), operators (line 31), variables (line 32), and text enclosed in single quotes. Non-terminals are program (line 1), function (line 2), statement (line 3), statement list (line 15), partial statement (line 16), and expression (line 18). The specification language is designed to retrieve data from the concurrent history, which is a list of *ActionObjects*, defined on line 1.36.

The expression on line 22 retrieves the unique transaction identification number associated with the transaction descriptor at sequence number x in the concurrent history. The transactional happens-before graph is initially empty at the start of each generated concurrent history.

Since some transactional correctness conditions are properties on only a subset of the transactions within a history, we require that a transaction must be explicitly inserted in the transactional happens-before graph in the specification. The statement on line 7 enables a transaction to be inserted in the transactional happens-before graph.

To place a happens-before relation between two transactions as shown on line 6, information on these transactions pertinent to the correctness condition being evaluated must be extracted. The expression on line 21 retrieves the size of the concurrent history. The thread id of a transaction can be obtained by the expression on line 23. This information is necessary for correctness conditions that place an ordering constraint on transactions called by the same thread. The status of a transaction at sequence number x in the concurrent history can be obtained by the expression on line 24, which is relevant for correctness conditions that place an ordering constraint only on committed transactions. A real-time ordering between transactions can be determined by the expression on line 25, which evaluates to true if the response of the first transaction occurs before the response of the second transaction. A real-time ordering constraint is placed on transactions for correctness conditions such as strict serializability and opacity.

Not all transactional correctness conditions require a total ordering on the transactions in a history. Causal consistency is one such example in which threads may perceive transactions in a different order. Our specification language accommodates this property by maintaining a per-thread transactional happens-before graph given the case that a correctness condition requires only a partial ordering on the transactions. Algorithm 4 must be applied to each thread's transactional happens-before graph to evaluate correctness. The statement on line 9 allows a transaction to be inserted into the transactional happens-before graph belonging to the thread as evaluated by the expression within the brackets. The statement on line 8 allows a happens-before relation to be placed between two transactions in a thread's transactional happens-before graph. Causal consistency places an ordering constraint on two transactions if one transaction's effects causes another transaction's effects. The expression on line 26 evaluates to true if at least one of the operations in the first transaction causes the effects of at least one of the operations in the second transaction. Internally, the evaluation of cause and effect between operations is performed by mapping each operation's output to an operation's input if a mapping exists.

The challenge with organizing transactions in a happens-before graph is that a recursive topological sort with a worst-case time complexity of $O(n!)$ must be applied to the graph to derive all possible legal sequential histories. We reduce this worst-case time complexity by pruning a recursive call that would explore a reordering of commutative transactions. Since commutative transactions can be reordered without affecting the resultant abstract state of the data structure, the exploration of commutative transactions called in opposite order is unnecessary, because it will produce the same legal sequential history. The statement on line 10 allows two methods to be declared as commutative given that the condition in parenthesis is never false. The expression on line 27 retrieves the method id of the method invoked at sequence number x in the concurrent history.

The operational semantics of the specification language are provided in Figure 4. A state is described by the $(n+3)$-tuple $(M, G, G_1, \ldots, G_n, c)$, where n is the number of threads in the unit test, M is a Boolean two-dimensional matrix such that the value at position (i,j) indicates if transaction i and transaction j are commutative, G is the transactional happens-before graph, G_i is the local transactional happens-before graph for thread i in the unit test, and c is the program statement to evaluate next. A transition from state S_0 to state S_1 is expressed as $S_0 \rightarrow S_1$.

3.3.1 *Serializability.*

Definition 3.8. A history h is *serializable* if the subsequence of h consisting of all events of committed transactions is equivalent to a legal sequential history [29].

1 IF $\frac{c' = (b == true)\ ?\ (c_1;c) : (c_2;c)}{(M,G,G_1,..,G_n,\text{if (b) } \{c_1\} \text{ else } \{c_2\};c) \rightarrow (M,G,G_1,..,G_n,c')}$

2 FOR $\frac{c' = (b == true)\ ?\ (\text{for } (c_i; b; c_i')\ \{c_1\};\ c) : (c)}{(M,G,G_1,..,G_n,\text{for } (c_i; b; c_i')\ \{c_1\};\ c) \rightarrow (M,G,G_1,..,G_n,c')}$

3 HAPPENS_BEFORE $\frac{G' = (E', V),\ E' = E' \cup \{e_1,e_2\}}{(M,G,G_1,..,G_n,e_1 \text{ happens\_before } e_2;c) \rightarrow (M,G',G_1,..,G_n,c)}$

4 INSERT_TXN $\frac{G' = (E, V'),\ V' = V' \cup \{e_1\}}{(M,G,G_1,..,G_n,\text{insert\_txn } e_1;c) \rightarrow (M,G',G_1,..,G_n,c)}$

5 HAPPENS_BEFORE_PARTIAL $\frac{G'_{e_2} = (E', V),\ E' = E' \cup \{e_1,e_3\}}{(M,G,G_1,..,G_{e_2},..,G_n,e_1 \text{ happens\_before\_partial } [e_2]\ e_3;c) \rightarrow (M,G,G_1,..,G'_{e_2},..,G_n,c)}$

6 INSERT_TXN_PARTIAL $\frac{G'_{e_1} = (E, V'),\ V' = V' \cup \{e_2\}}{(M,G,G_1,..,G_{e_1},..,G_n,\text{insert\_txn\_partial } [e_1]\ e_2;c) \rightarrow (M,G,G_1,..,G'_{e_1},..,G_n,c)}$

7 COMMUTES_WITH $\frac{c' = (b == false)\ ?\ (\ M' = (E', V),\ E' = E' \setminus \{e_1,e_2\}) : (M' = M)}{(M,G,G_1,..,G_n,e_1 \text{ commutes\_with } e_2,\ (b);c) \rightarrow (M',G,G_1,..,G_n,c)}$

Fig. 4. Operational semantics for the custom specification language.

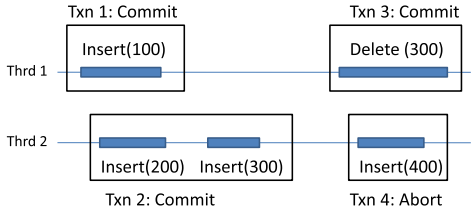

Fig. 5. Concurrent history example.

```
1    for(n = 0; n < history->size(); n++)
2    {
3      if(txn_status(n) == COMMITTED)
4      {
5        insert_txn(txn(n));
6      }
7    }
```

Fig. 6. Serializability specification.

```
Relation:
NULL

Graph:
1 ->
2 ->
3 ->

Legal Sequential Histories:
Insert(100);Insert(200);Insert(300);Delete(300);
Insert(100);Delete(300);Insert(200);Insert(300);
Insert(200);Insert(300);Insert(100);Delete(300);
Insert(200);Insert(300);Delete(300);Insert(100);
Delete(300);Insert(100);Insert(200);Insert(300);
Delete(300);Insert(200);Insert(300);Insert(100);
```

Fig. 7. Happens-before example for serializability.

Serializability requires that all committed transactions preserve atomicity and isolation. There is no ordering constraint placed on the individual transactions. The specification for serializability is shown in Figure 6. If the transaction status is determined to be committed on Line 3, then the transaction is inserted in the transactional happens-before graph on Line 5.

Figure 7 shows the happens-before graph and legal sequential histories generated from the specification of Figure 6 applied to the concurrent history of Figure 5. The happens-before graph contains all committed transactions without any ordering constraints. The legal sequential histories encompasses all topological sorts of the happens-before graph. Since there are 3! ways to order three items, there are six possible legal sequential histories, as shown in Figure 7.

3.3.2 *Strict Serializability.*

Definition 3.9. A history h is *strictly serializable* if the subsequence of h consisting of all events of committed transactions is equivalent to a legal sequential history in which these transactions execute sequentially in the order they commit [29].

```
1  for(n = 0; n < history->size(); n++) {
2    for(m = 0; m < n; m++) {
3      if(txn_status(m) == COMMITTED) {
4        insert_txn(txn(m));
5      }
6      if(txn_status(n) == COMMITTED) {
7        insert_txn(txn(n));
8      }
9      if((txn_status(m) == COMMITTED) && (
           ↪ txn_status(n) == COMMITTED)) {
10       if(txn(m) precedes txn(n)) {
11         txn(m) happens_before txn(n);
12       }
13     }
14   }
15 }
```

Fig. 8. Strict serializability specification.

```
Relation:
1 happens_before 3
2 happens_before 3

Graph:
1 -> 3
2 -> 3
3 ->

Legal Sequential Histories:
Insert(100);Insert(200);Insert(300);Delete(300);
Insert(200);Insert(300);Insert(100);Delete(300);
```

Fig. 9. Happens-before example for strict serializability.

Strict serializability requires that all committed transactions preserve real-time ordering, as well as atomicity and isolation. The specification for strict serializability is shown in Figure 8. If a transaction is committed, then it is inserted in the transactional happens-before graph on lines 4 and 7. The keyword *precedes* on line 10 evaluates to true if the response of transaction m occurs before the invocation of transaction n in real-time. If both transaction m and transaction n are committed, and transaction m precedes transaction n, then a happens-before relation is placed on transaction m and transaction n on line 11.

Figure 9 shows the happens-before graph and legal sequential histories generated from the specification of Figure 8 applied to the concurrent history of Figure 5. The happens-before graph contains all committed transactions, where transactions 1 and 2 are ordered before transaction 3 due to the real-time ordering constraint of strict serializability. Since there is no ordering constraint between transactions 1 and 2, there are two possible legal sequential histories, as shown in Figure 7.

3.3.3 Opacity. Opacity requires that all transactions (committed, aborted, or active) observe a consistent state of the system. Prior to defining opacity, we provide definitions to transform an incomplete history into a complete history by aborting or committing the active transactions. A *commit-try* event is a request to commit. An *abort-try* event is a request to abort. An active transaction that has issued a commit-try is *commit-pending*.

Definition 3.10. A history h is *well formed* if each individual transaction T_i comprises a sequence of invocation and matching response events such that (1) no event follows a commit or abort event, (2) only a commit or abort event can follow a commit-try event, and (3) only an abort event can follow an abort-try event [17].

Definition 3.11. A history h' is in *Complete(h)* if (1) h' is well formed, (2) h' is obtained from h by inserting a number of commit-try, commit, and abort events for transactions that are active in h, (3) every transaction that is active and not commit-pending in h is aborted in h', and (4) every transaction that is commit-pending in h is either committed or aborted in h' [17].

Definition 3.12. A history h is *opaque* if there exists a sequential history S equivalent to some history in the set Complete(h) such that (1) S preserves the real-time order of h and (2) every transaction $T_i \in S$ is legal in S [17].

Opacity requires that all transactions preserve real-time ordering, as well as atomicity and isolation. Since all transactions must observe a consistent state of the system, all transactions are

```
1   for(n = 0; n < history->size(); n++) {
2     for(m = 0; m < n; m++) {
3       insert_txn(txn(m));
4       insert_txn(txn(n));
5       if(txn(m) precedes txn(n))
6       {
7         txn(m) happens_before txn(n);
8       }
9     }
10  }
```

Fig. 10. Opacity specification.

```
Relation:                    Graph:
1 happens_before 3           1 -> 3 -> 4
2 happens_before 3           2 -> 3 -> 4
1 happens_before 4           3 ->
2 happens_before 4           4 ->
Legal Sequential Histories:
Insert(100);Insert(200);Insert(300);Delete(300);
      ↪ Insert(400);Insert_Inv(400);
Insert(100);Insert(200);Insert(300);Insert(400);
      ↪ Insert_Inv(400);Delete(300);
Insert(200);Insert(300);Insert(100);Delete(300);
      ↪ Insert(400);Insert_Inv(400);
Insert(200);Insert(300);Insert(100);Insert(400);
      ↪ Insert_Inv(400);Delete(300);
```

Fig. 11. Happens-before example for opacity.

inserted in the transactional happens-before graph regardless of their status. The specification for opacity is shown in Figure 10. If transaction m precedes transaction n in real-time, then a happens-before relation is placed between transaction m and transaction n on line 7.

Figure 11 shows the happens-before graph and legal sequential histories generated from the specification of Figure 10 applied to the concurrent history of Figure 5. The happens-before graph contains all transactions (committed, active, or aborted), where transactions 1 and 2 are ordered before transactions 3 and 4 due to the real-time ordering constraint of opacity. Since there is no ordering constraint between transactions 1 and 2, or transactions 3 and 4, there are four possible legal sequential histories, as shown in Figure 11. Since transaction 4 aborts, the inverse of `Insert`(400) (`Insert_Inv`(400)) must be applied to undo the effects of transaction 4 in the legal sequential histories.

3.3.4 Causal Consistency.

Definition 3.13. A *causality relation* consists of operation pairs (X, Y) such that operation X causes operation Y.

Definition 3.14. A history h is *causally consistent* if for each thread t_i, there exists a sequential history S_i equivalent to some history in the set Complete(h), such that (1) S_i preserves the causality relation and (2) every committed transaction executed by t_i is legal in S_i [30].

Causal consistency requires that committed transactions observe other transactions issued by the same thread and transactions that cause the observed effects, where the observed effects must preserve atomicity and isolation. Since each thread may observe a different ordering on the transactions, the committed transactions as a whole cannot be totally ordered. Therefore, each thread must maintain its own transactional happens-before graph. The specification for causal consistency is shown in Figure 12. If a transaction is committed, then it is inserted into its thread's transactional happens-before graph on lines 4 and 7. If both transaction m and transaction n are committed, then there are two scenarios in which a happens-before relation may be placed on the transactions. The first scenario occurs if both transaction m and transaction n are issued by the same thread and transaction m precedes transaction n, as shown on lines 10 and 11. The second scenario occurs if transaction m and transaction n are not issued by the same thread and transaction m causes transaction n, as shown on lines 12, 13, and 14. The happens-before relation is only placed between transaction m and transaction n in the graph of the thread that issued transaction n. This is due to the transaction that caused the effect is not necessarily aware of the effect.

Figure 13 shows the happens-before graph and legal sequential histories generated from the specification of Figure 12 applied to the concurrent history of Figure 5. The happens-before graph

```
1  for(n = 0; n < history->size(); n++) {
2    for(m = 0; m < n; m++) {
3      if(txn_status(m) == COMMITTED) {
4        insert_txn_partial[txn_tid(m)] txn(m);
5      }
6      if(txn_status(n) == COMMITTED) {
7        insert_txn_partial[txn_tid(n)] txn(n);
8      }
9      if((txn_status(n) == COMMITTED) && (txn_status
           ↪ (m) == COMMITTED)) {
10       if((txn_tid(m) == txn_tid(n)) && (txn(m)
             ↪ precedes txn(n))) {
11         txn(m) happens_before_partial[txn_tid(n)]
             ↪ txn(n);
12       } else if ((txn_tid(m) != txn_tid(n)) && (
             ↪ txn(m) causes txn(n))) {
13         insert_txn_partial[txn_tid(n)] txn(m);
14         txn(m) happens_before_partial[txn_tid(n)]
             ↪ txn(n);
15       }
16     }
17   }
18 }
```

Fig. 12. Causal consistency specification.

```
Thread 1 Relation:      Thread 1 Graph:
1 happens_before 3      1 -> 3
2 happens_before 3      2 -> 3
Thread 2 Relation:      3 ->
NULL                    Thread 2 Graph:
                        2 ->

Thread 1 Legal Sequential Histories:
Insert(100);Insert(200);Insert(300);Delete(300);
Insert(200);Insert(300);Insert(100);Delete(300);
Thread 2 Legal Sequential Histories:
Insert(200);Insert(300);
```

Fig. 13. Happens-before example for causal consistency.

```
1  for(n = 0; n < history->size(); n++)
2  {
3    if(txn_status(n) == COMMITTED)
4    {
5      insert_txn(txn(n));
6    }
7    for(m = 0; m < n; m++)
8    {
9      method(m) commutes_with method(n), (input(m)
           ↪ != input(n));
10   }
11 }
```

Fig. 14. Serializability specification with commutative methods specified.

```
commutes_matrix:        reorder_matrix:
  1 2 3                   1 2 3
1 - T T                 1 - T T
2 - - F                 2 - - T
3 - - -                 3 - - -

Legal Sequential Histories:
Insert(100);Insert(200);Insert(300);Delete(300);
Insert(100);Delete(300);Insert(200);Insert(300);

Pruned Legal Sequential Histories:
Insert(200);Insert(300);Insert(100);Delete(300);
Insert(200);Insert(300);Delete(300);Insert(100);
Delete(300);Insert(100);Insert(200);Insert(300);
Delete(300);Insert(200);Insert(300);Insert(100);
```

Fig. 15. Pruning example for serializability with commutative methods specified.

is maintained on a per-thread basis. Each thread i observes the committed transactions issued by thread i in commit order as well as committed transactions from other threads that cause the effects of thread i's transactions. Since `Delete`(300) of transaction 3 observes the effects of `Insert`(300) of transaction 2, thread 1's happens-before graph orders transaction 2 before transaction 3. Thread 2's happens-before graph only contains transaction 2, because none of thread 1's transactions cause the effects of transaction 2. Since there is no ordering constraint between transactions 1 and 2 in thread 1's happens-before graph, there are two possible legal sequential histories, as shown in Figure 13. Thread 2's happens-before graph only contains transaction 2, yielding one possible legal sequential history.

3.3.5 Commutativity Specification. The recursive topological sort in the correctness checking function, detailed in Algorithm 4, is optimized by specifying commutative methods in a transaction. Figure 14 depicts the specification for serializability with a specification identifying commutative methods for set operations on line 9. Set operations are commutative if they have different input arguments. Figure 15 shows the *commutes_matrix*, *reorder_matrix*, legal sequential histories,

Table 1. TxC-ADT Results for Transactional Data Structures

	Correctness Condition			
Transactional Data Structure	Causal Consistency	Serializability	Strict Serializability	Opacity
LFTT Linked List	Pass	Pass	Pass	Pass
LFTT Skiplist	Pass	Pass	Pass	Pass
TDSL Queue	Pass	Pass	Pass	Pass
TDSL Skiplist	Pass	Pass	Pass	Pass

and the pruned legal sequential histories for the example concurrent history in Figure 5. In this example, transactions 1 and 2 commute and transactions 1 and 3 commute, because the input passed to each method of the transaction is unique. Transaction 2 and 3 do not commute, because they both invoke a method with input 300. The *commutes_matrix* reflects this relationship, because position (1, 2) = true, position (1, 3) = true, and position (2, 3) = false. All positions of the *reorder_matrix* are true, because serializability does not enforce any particular order on the transactions. Since transaction 1 commutes with both transaction 2 and transaction 3, the only reordering that must be explored is between transaction 2 and transaction 3, as listed in the legal sequential histories of Figure 15. All other ordering may be pruned from the recursive topological sort, because they yield the same abstract state of the data structure as the orderings listed in the legal sequential histories.

4 CASE STUDIES

We evaluate TxC-ADT by checking the correctness of Lock-Free Transactional Transformation (LFTT) [38] and Transactional Data Structure Libraries (TDSL) [34]. The tests are conducted on a 64-core NUMA system (4 AMD opteron 6272 CPUs with 16 cores per chip @2.1GHz). Each unit test comprises three threads such that two threads each issue a transaction with two operations and one thread issues a transaction with one operation. The operations in the transactions are selected such that a high-level semantic conflict exists between two transactions issued by different threads. For the set abstract data type, one transaction invokes `Insert`(X) and another transaction invokes `Delete`(X). For the queue abstract data type, one transaction invokes `Dequeue`() and another transaction also invokes `Dequeue`(). We collect the concurrent histories from CDSChecker and equally distribute them among the 64 cores to check correctness using TxC-ADT. The correctness conditions incorporated in the evaluation include serializability, strict serializability, opacity, and causal consistency. The results are shown in Table 1. The data structures of both approaches meet opacity, the strongest transactional correctness property. These are the expected results, since LFTT is designed for strict serializability and TDSL is designed for opacity. Although the correctness proofs for LFTT [38] verify strict serializability, the approach of LFTT is also opaque, because the logical interpretation allows all transactions to observe a consistent state of the system regardless of the transaction status.

The execution times for the unit tests are shown in Table 2. The column abbreviated CDS is the time (in seconds) for CDSChecker to generate the concurrent histories of the transactional data structure unit test. The column abbreviated TxC is the time (in seconds) for TxC-ADT to analyze the concurrent histories and determine if the unit test meets the specified transactional correctness condition. Opacity takes the largest amount of time to check, because the effects of all transactions (active, committed, and aborted) must be evaluated for correctness. Strict serializability generally takes more time to check than serializability, because the specification for strict serializability takes additional time to incorporate real-time ordering in the transactional happens-before graph. The

Table 2. TxC-ADT Execution Time Results (in Seconds)

	LFTT Linked List (seconds)		LFTT Skiplist (seconds)		TDSL Queue (seconds)		TDSL Skiplist (seconds)	
Correctness Condition	TxC.	CDS.	TxC.	CDS.	TxC.	CDS.	TxC.	CDS.
Causal Consistency	1,076	901	30	57	18	15	569	640
Serializability	958	901	28	57	21	15	501	640
Strict Serializability	1,451	901	28	57	24	15	468	640
Opacity	1,729	901	28	57	22	15	532	640
Number of Concurrent Histories	405,524		11,323		8,401		210,967	

```
1  ThreadA:
2  TXBegin();
3  Insert(2);
4  Delete(3);
5  TXEnd();
```

```
1  ThreadB:
2  TXBegin();
3  Insert(3);
4  Insert(2);
5  TXEnd();
```

Fig. 16. LFTT linked list unit test.

```
1  Concurrent History 1:
2  A:Insert(2);
3  A:Delete(3);
4  A:Commit();
5  B:Insert(3);
6  B:Insert(2);
7  B:Abort();
```

```
1  Concurrent History 2:
2  B:Insert(3);
3  B:Insert(2);
4  B:Abort();
5  A:Insert(2);
6  A:Delete(3);
7  A:Commit();
```

Fig. 17. LFTT linked list concurrent histories.

time to check causal consistency increases as the number of transactions that satisfy the causality relation increases. This occurs because an increase in the per-thread transactional happens-before graph size requires more time to analyze. The variance in execution time for each data structure is due to the total number of concurrent histories computed for each unit test. The total number of concurrent histories computed by CDSChecker increases as the number of atomic operations called in the unit test increases. The large variance between the LFTT linked list and the LFTT skiplist is due to the logarithmic search time of a skiplist reduces the total number of possible atomic operations invoked in the unit test.

To demonstrate the ability of TxC-ADT to produce counterexamples, we inject design flaws within the source code accompanying LFTT and TDSL that may occur in the development of transactional data structures. The following subsections provide a brief overview of the LFTT and TDSL designs and explains the counterexamples resulting from the injected design flaws.

4.1 Lock-Free Transactional Transformation

LFTT is a methodology for transforming high-performance lock-free base data structures into high-performance lock-free transactional data structures. LFTT introduces a node-based conflict detection scheme that allows commutative operations to proceed concurrently using the thread-level synchronization of the lock-free base data structure. Non-commutative operations require transaction-level synchronization where the thread that detects a conflict will help finish the delayed transaction by utilizing a transaction descriptor that stores the instructions and arguments for operations and a transaction status. The penalties of rollbacks are minimized by incorporating a logical rollback where a transaction may interpret the logical status of a node based on the operation type and the transaction status recorded in the transaction descriptor.

We inject a design flaw into the original LFTT linked list source code to produce counterexamples when checking for opacity. The design flaw entails a disabling of the logical interpretation so the threads observe the concrete state of the system instead of the abstract state of the system. This design flaw causes the effects of a transaction to be visible to other transactions prior to a commit, which will violate the isolation property of transactional execution. The unit test is shown in Figure 16, the concurrent histories are shown in Figure 17, and the resulting counterexamples are shown in Figure 18. In the left counterexample, thread A's transaction executes first and commits, and thread B's transaction executes second and aborts. When executing these transactions in isolation, the `Delete(3)` operation of thread A should return false, because 3 has not been inserted

```
1  Concurrent History 1:
2  Sequential Output:
3  Insert(2):T Delete(3):F Insert(3):T Insert(2):F
4  Program output:
5  Insert(2):T Delete(3):T Insert(3):T Insert(2):F
```

```
1  Concurrent History 2:
2  Sequential Output:
3  Insert(3):T Insert(2):T Insert(2):T Delete(3):F
4  Program output:
5  Insert(3):T Insert(2):F Insert(2):T Delete(3):T
```

Fig. 18. LFTT linked list opacity counterexamples (with design flaws injected).

```
1  Main:
2  TXBegin();
3  Enq(1);
4  TXEnd();
```

```
1  ThreadA:
2  TXBegin();
3  Deq();
4  Enq(3);
5  TXEnd();
```

```
1  ThreadB:
2  TXBegin();
3  Deq();
4  Enq(2);
5  TXEnd();
```

Fig. 19. TDSL queue unit test.

```
1  Concurrent History 1:
2  Main:Enqueue(1);
3  Main:Commit();
4  A:Deq();
5  A:Enq(3);
6  A:Commit();
7  B:Deq();
8  B:Enq(2);
9  B:Commit();
```

```
1  Concurrent History 2:
2  Main:Enqueue(1);
3  Main:Commit();
4  B:Deq();
5  B:Enq(2);
6  B:Commit();
7  A:Deq();
8  A:Enq(3);
9  A:Commit();
```

Fig. 20. TDSL queue concurrent histories.

in the set, and the `Insert(2)` operation of thread B should return false, because thread A already inserted 2 in the set. However, the program output demonstrates that thread A's `Delete(3)` operation observes the effects of thread B's `Insert(3)` operation and successfully removes 3 from the set. Thread B's `Insert(2)` operation fails, because it observes the effects of thread B's `Insert(2)` operation.

In the right counterexample, thread B's transaction executes first and aborts, and thread A's transaction executes second and commits. Since opacity requires that all transactions observe a consistent state of the system, the output of all transactions must be evaluated. When executing these transactions in isolation, thread B's operations will both succeed, since the set is initially empty. However, since thread B aborts, its effects must be invisible to thread A. The `Insert(2)` operation by thread A should succeed and the `Delete(3)` operation by thread A should fail, since the abstract state of the set is an empty list after the abort by thread B. The program output demonstrates that thread B's `Insert(2)` operation fails, because it observes thread A's `Insert(2)` operation, and thread A's `Delete(3)` operation succeeds, because it observes the `Insert(3)` by thread B.

4.2 Transactional Data Structure Libraries

TDSL introduces a methodology for bundling sequences of data structure operations into atomic transactions. TDSL enables customizations to the read-set tracking and validation to incorporate standard software transactional memory techniques or optimizations such that the read-set only includes memory locations that represent real semantic conflicts. TDSL provides composition of transactional data structures, as well as support for singleton transactions consisting of an individual operation.

We inject a design flaw into the original TDSL queue source code to produce counterexamples when checking for opacity. The design flaw entails disabling the locking of the queue during a transactional commit and the preemptive locking during a dequeue operation. This design flaw causes the effects of a transaction to be visible prior to the commit, which violates the isolation property of transactional execution. The unit test is shown in Figure 19, the concurrent histories are shown in Figure 20, and the resulting counterexamples are shown in Figure 21. Thread A and thread B both invoke a `Dequeue` operation on a queue with one element. Since both threads hold a reference to the same head element and the queue is not locked during the commit, both dequeue 1, since it is at the head of the queue. In the left counterexample, the sequential output indicates

```
1  Concurrent History 1:
2  Sequential Output:
3  Enq(1):Void
4  Deq():1 Enq(3):Void Deq():3 Enq(2):Void
5  Program output:
6  Enq(1):Void
7  Deq():1 Enq(3):Void Deq():1 Enq(2):Void
```

```
1  Concurrent History 2:
2  Sequential Output:
3  Enq(1):Void
4  Deq():1 Enq(2):Void Deq():2 Enq(3):Void
5  Program output:
6  Enq(1):Void
7  Deq():1 Enq(2):Void Deq():1 Enq(3):Void
```

Fig. 21. TDSL queue opacity counterexamples (with design flaws injected).

that thread B should dequeue 3, because thread A commits first. In the right counterexample, the sequential output indicates that thread A should dequeue 2, because thread B commits first.

5 LIMITATIONS

TxC-ADT has several limitations inherent with model checking tools. The concurrent histories generated during model checking are for a unit test of the data structure. If the unit test does not expose an incorrect concurrent history, then TxC-ADT will report that the data structure satisfies the specified correctness condition. A corner case is an extreme configuration of a data structure, such as a full or empty queue. The unit test should be written to include all known corner cases to expose bugs that would go undetected in a general unit test.

Model checking is vulnerable to state space explosion due to the exploration of all possible thread interleavings. CDSChecker [27] uses dynamic partial order reduction [13] to minimize the exploration of redundant executions. Since CDSChecker cannot completely explore infinite state spaces, unbounded loops are explored under the restriction of a fair schedule. Additional effort is also required by the user to construct a unit test that is as small as possible while including all data structure operations and corner cases. Since a unit test that includes all possible inputs leads to a infinite state space, a minimal set of inputs should be chosen to explore the possible behaviors of the data structure.

TxC-ADT's recursive topological sort optimization is limited to operations such that commutativity is independent of the state of the data structure. For example, the `enqueue()` and `dequeue()` operations of a queue commute if the queue is not empty. Since the state of the queue is affected by any committed transaction, it is not possible to conclusively determine if two transactions comprising queue operations will always commute. Establishing a commutative relationship between transactions is limited to set operations, since two transactions on a set commute given that the operations in transaction t_1 are passed different inputs than the operations in transaction t_2.

6 CONCLUSION

We presented TxC-ADT, the first correctness tool that can check the correctness of transactional data structures. TxC-ADT's capabilities encompass the designed correctness guarantees of transactional data structures that employ a high-level semantic conflict detection by recasting correctness in terms of an abstract data type. Existing correctness verification tools for transactional memory systems evaluate correctness according to the thread transitions in the presence of low-level read/write conflicts, which is not applicable to state-of-the-art transactional data structures. We accommodate a diverse assortment of transactional correctness conditions by presenting a technique for defining correctness as a happens-before relation. Establishing the conditions for which a transaction happens before another transaction enables correctness to be evaluated automatically through an analysis of a transactional happens-before graph during model checking. Since our technique preserves atomicity and isolation, it can be easily extended to other transactional correctness conditions that may be adopted in the advancement of transactional data

structures. We account for transactional correctness conditions that do not require a total order on a transactional execution by maintaining a per-thread transactional happens-before graph. This strategy enables the verification of causal consistency in which a thread only observes the effects of transactions by other threads if the effect satisfies the causality relation. The case studies demonstrate the practical application of TxC-ADT to check the correctness of cutting-edge transactional data structures.

REFERENCES

[1] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M. Hellerstein. 2013. Consistency without borders. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. ACM, 23.

[2] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. 2010. Implementing and evaluating a model checker for transactional memory systems. In *Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'10)*. IEEE, 117–126.

[3] Annette Bieniusa and Peter Thiemann. 2011. Proving isolation properties for software transactional memory. In *European Symposium on Programming*. Springer, 38–56.

[4] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. 2006. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Arch. Lett.* 5, 2 (2006).

[5] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: A complete and automatic linearizability checker. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation 2010 (PLDI'10)*, Vol. 45. ACM, 330–340.

[6] Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. 2007. Verifying correctness of transactional memories. In *Formal Methods in Computer Aided Design 2007 (FMCAD'07)*. IEEE, 37–44.

[7] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. 2008. Mechanical verification of transactional memories with non-transactional memory accesses. In *International Conference on Computer Aided Verification*. Springer, 121–134.

[8] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*, Vol. 45. ACM, 67–78.

[9] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *International Symposium on Distributed Computing*. Springer, 194–208.

[10] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2013. Towards formally specifying and verifying transactional memory. *Formal Aspects Comput.* 25, 5 (2013), 1–31.

[11] Michael Emmi, Rupak Majumdar, and Roman Manevich. 2010. Parameterized verification of transactional memories. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation 2010 (PLDI'10)*, Vol. 45. ACM, 134–145.

[12] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation 2008 (PLDI'08)*. 293–303.

[13] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 2005 (POPL'05)*, Vol. 40. ACM, 110–121.

[14] Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. 2008. Model checking transactional memories. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation 2008 (PLDI'08)*. 372–382.

[15] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. 2008. Completeness and nondeterminism in model checking transactional memories. *Lect. Not. Comput. Sci.* 5201 (2008), 21–35.

[16] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. 2009. Software transactional memory on relaxed memory models. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'09)*, Vol. 9. Springer, 321–336.

[17] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 175–184.

[18] Maurice Herlihy and Eric Koskinen. 2008. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 207–216.

[19] Maurice Herlihy and J Eliot B Moss. 1993. *Transactional Memory: Architectural Support for Lock-free Data Structures*. Vol. 21. ACM.

[20] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Trans. Softw. Eng.* 23, 5 (1997), 279–295.

[21] Phillip W Hutto and Mustaque Ahamad. 1990. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems 1990*. IEEE, 302–309.

[22] Heiner Litz, Ricardo J Dias, and David R Cheriton. 2015. Efficient correction of anomalies in snapshot isolation transactions. *ACM Trans. Arch. Code Optimiz.* 11, 4 (2015), 65.

[23] Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. 2006. Testing implementations of transactional memory. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 134–143.

[24] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. 2006. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*.

[25] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and reproducing heisenbugs in concurrent programs. In *Proceedingso of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Vol. 8. 267–280.

[26] Peter Muth, Thomas C. Rakow, Gerhard Weikum, Peter Brossler, and Christof Hasse. 1993. Semantic concurrency control in object-oriented database systems. In *Proceedings of the 9th International Conference on Data Engineering 1993*. IEEE, 233–242.

[27] Brian Norris and Brian Demsky. 2013. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*, Vol. 48. ACM, 131–150.

[28] John O'Leary, Bratin Saha, and Mark R. Tuttle. 2009. Model checking transactional memory with Spin. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems 2009 (ICDCS'09)*. IEEE, 335–342.

[29] Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *J. ACM* 26, 4 (1979), 631–653.

[30] Michel Raynal, Gérard Thia-Kime, and Mustaque Ahamad. 1997. From serializable to causal transactions for collaborative applications. In *Proceedings of the 23rd EUROMICRO Conference New Frontiers of Information Technology (EUROMICRO'97)*. IEEE, 314–321.

[31] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 187–197.

[32] Manfred Schmidt-Schauß and David Sabel. 2013. *Correctness of an STM Haskell Implementation*. Vol. 48. ACM.

[33] Peter M. Schwarz and Alfred Z Spector. 1984. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.* 2, 3 (1984), 223–250.

[34] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 682–696.

[35] Gary D. Walborn and Panos K. Chrysanthis. 1995. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings of the 14th Symposium on Reliable Distributed Systems 1995*. IEEE, 31–40.

[36] Paul Wu and Alan Fekete. 2003. An empirical study of commutativity in application code. In *Proceedings of the 7th International Database Engineering and Applications Symposium 2003*. IEEE, 361–369.

[37] Paul Wu, Alan Fekete, and Uwe Rohm. 2008. The efficacy of commutativity-based semantic locking in a real-world application. *IEEE Trans. Knowl. Data Eng.* 20, 3 (2008), 427–431.

[38] Deli Zhang and Damian Dechev. 2016. Lock-free transactions without rollbacks for linked data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 325–336.

Received June 2017; revised September 2017; accepted September 2017