


2019

Correctness and Progress Verification of Non-Blocking Programs

Christina Peterson
University of Central Florida

 Part of the [Computer Sciences Commons](#)
Find similar works at: <https://stars.library.ucf.edu/etd>
University of Central Florida Libraries <http://library.ucf.edu>

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation

Peterson, Christina, "Correctness and Progress Verification of Non-Blocking Programs" (2019). *Electronic Theses and Dissertations*. 6557.
<https://stars.library.ucf.edu/etd/6557>



CORRECTNESS AND PROGRESS VERIFICATION OF NON-BLOCKING PROGRAMS

by

CHRISTINA PETERSON

B.S. in Civil Engineering, University of Central Florida, 2003

B.S. in Computer Science, University of Central Florida, 2013

M.S. in Computer Science, University of Central Florida, 2017

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2019

Major Professor: Damian Dechev

© 2019 Christina Peterson

ABSTRACT

The progression of multi-core processors has inspired the development of concurrency libraries that guarantee safety and liveness properties of multiprocessor applications. The difficulty of reasoning about safety and liveness properties in a concurrent environment has led to the development of tools to verify that a concurrent data structure meets a correctness condition or progress guarantee. However, these tools possess shortcomings regarding the ability to verify a composition of data structure operations. Additionally, verification techniques for transactional memory evaluate correctness based on low-level read/write histories, which is not applicable to transactional data structures that use a high-level semantic conflict detection.

In my dissertation, I present tools for checking the correctness of multiprocessor programs that overcome the limitations of previous correctness verification techniques. Correctness Condition Specification (CCSpec) is the first tool that automatically checks the correctness of a composition of concurrent multi-container operations performed in a non-atomic manner. Transactional Correctness tool for Abstract Data Types (TxC-ADT) is 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.

Many practical concurrent data structures, transactional data structures, and algorithms to facilitate non-blocking programming all incorporate helping schemes to ensure that an operation comprising multiple atomic steps is completed according to the progress guarantee. The helping scheme introduces additional interference by the active threads in the

system to achieve the designed progress guarantee. Previous progress verification techniques do not accommodate loops whose termination is dependent on complex behaviors of the interfering threads, making these approaches unsuitable. My dissertation presents the first progress verification technique for non-blocking algorithms that are dependent on descriptor-based helping mechanisms.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Damian Dechev for providing the inspiration for the ideas presented in my dissertation. I would like to thank my dissertation committee members Dr. Gary Leavens, Dr. Mostafa Bassiouni, and Dr. Mason Cash for their insightful feedback on my dissertation.

TABLE OF CONTENTS

LIST OF FIGURES	xi
LIST OF TABLES	xiv
LIST OF ALGORTIHMS	xv
CHAPTER 1: INTRODUCTION	1
Correctness	2
Progress	5
Leveraging Semantics to Increase Performance	6
Outline	7
CHAPTER 2: BACKGROUND	9
Terminology	9
Concurrent Execution	9
Correctness Property	10
Progress Property	10
Related Work	11

Correctness of Non-Blocking Data Structures	11
Correctness of Transactional Memory	13
Progress Verification of Non-Blocking Algorithms	16
Cooperative Schemes for Transactional Applications	19
CHAPTER 3: METHODOLOGY	22
CCSpec: A Correctness Condition Specification Tool	22
Data Structure and Abstract Function Layers	24
Approach for Checking Correctness	26
Specification Language	29
Linearizability	32
Sequential Consistency	33
Quiescent Consistency	34
Quasi-Linearizability	34
A Transactional Correctness Tool for Abstract Data Types	36
General Approach	39
A Unification of Transactional Correctness Conditions	41
Specification Language	48

Serializability	51
Strict Serializability	52
Opacity	54
Causal Consistency	56
Commutativity Specification	59
Practical Progress Verification of Descriptor-based Non-blocking Data Structures	60
Assertion Language	62
Verification of Descriptor-Based Helping Techniques	63
Verification Framework for Concurrent Programs	70
Transactional Merging to Optimize Semantic Conflict Detection in Transactional	
Data Structures	74
Transactional Merging	75
Generalizing Transactional Merging	77
Operation Precondition	77
Recovery Scheme	79
LFTT With Transactional Merging	79
Semantic Conflict Resolution Policy	80
Transformed Map Functions	85

CHAPTER 4: EXPERIMENTAL EVALUATION	86
CCSpec	86
Results	86
Counterexamples	90
k-FIFO Queue	91
Priority Queue	92
Adjacency List	93
TXC-ADT	94
Lock-Free Transactional Transformation	97
Transactional Data Structure Libraries	99
Limitations	101
Progress Verification	103
Lock-Free Transactional List	103
Wait-Free Queue	105
Transactional Merging	108
Transactional List and Red-Black Tree	108
Transactional Dictionary	113

CHAPTER 5: CONCLUSION	116
APPENDIX A: CORRECTNESS OF CCSPEC	119
APPENDIX B: CORRECTNESS OF TXC-ADT	124
APPENDIX C: CORRECTNESS OF TRANSACTIONAL MERGING	129
Definitions	130
Rules	131
Strict Serializability and Recovery	132
Progress Guarantees	137

LIST OF FIGURES

Figure 1.1: Data Structure Composition Example	3
Figure 3.1: Method Call Annotation Example	24
Figure 3.2: Grammar for the Custom Specification Language	30
Figure 3.3: Operational Semantics for the Custom Specification Language	32
Figure 3.4: Linearizability Specification	33
Figure 3.5: Sequential Consistency Specification	34
Figure 3.6: Quiescent Consistency Specification	35
Figure 3.7: Quasi-Linearizability Specification	35
Figure 3.8: Abstract Data Type Annotation Example	39
Figure 3.9: Correctness Condition Declaration	43
Figure 3.10: Grammar for the Custom Specification Language	49
Figure 3.11: Operational Semantics for the Custom Specification Language . . .	51
Figure 3.12: Serializability Specification	52
Figure 3.13: Happens-Before Example for Serializability	52
Figure 3.14: Concurrent History Example	53

Figure 3.15: Strict Serializability Specification	54
Figure 3.16: Happens-Before Example for Strict Serializability	54
Figure 3.17: Opacity Specification	55
Figure 3.18: Happens-Before Example for Opacity	56
Figure 3.19: Causal Consistency Specification	58
Figure 3.20: Happens-Before Example for Causal Consistency	58
Figure 3.21: Serializability Specification with Commutative Methods Specified . .	60
Figure 3.22: Pruning Example for Serializability with Commutative Methods Specified	60
Figure 3.23: The Assertion Language	63
Figure 3.24: Descriptor-Based Lock-Free Algorithm	65
Figure 3.25: Helping Function for Descriptor-Based Wait-Free Algorithm	67
Figure 3.26: Descriptor-Based Wait-Free Algorithm	68
Figure 3.27: Overview of Progress Guarantee Verification Framework	70
Figure 3.28: Definitions for the Rely/Guarantee Conditions Mechanized in Coq .	72
Figure 3.29: Transactional Merging for a Map	76
Figure 4.1: k-FIFO Linearizability Counterexamples	91

Figure 4.2: Priority Queue Linearizability Counterexamples	93
Figure 4.3: Adjacency List Unit Test	94
Figure 4.4: Adjacency List Linearizability Counterexamples	94
Figure 4.5: LFTT Linked List Unit Test	98
Figure 4.6: LFTT Linked List Concurrent Histories	98
Figure 4.7: LFTT Linked List Opacity Counterexamples (with design flaws injected)	98
Figure 4.8: TDSL Queue Unit Test	100
Figure 4.9: TDSL Queue Concurrent Histories	100
Figure 4.10: TDSL Queue Opacity Counterexamples (with design flaws injected)	101
Figure 4.11: Specification for FinishPendingTxn	104
Figure 4.12: Shared Resources for Wait-Free Queue	106
Figure 4.13: Specification for help_enq	107
Figure 4.14: List and Red-Black Tree Results on NUMA system	110
Figure 4.15: List and Red-Black Tree Results on Dell Precision	112
Figure 4.16: TPC-C Benchmark on NUMA system	113
Figure 4.17: TPC-C Benchmark on Dell Precision	113

LIST OF TABLES

Table 4.1: CCSpec Results for the Data Structure Layer	87
Table 4.2: CCSpec Results for the Abstract Function Layer	87
Table 4.3: Concurrent Histories Generated by CDSChecker	88
Table 4.4: CCSpec Execution Time Results (in hours)	90
Table 4.5: TxC-ADT Results for Transactional Data Structures	95
Table 4.6: Concurrent Histories Generated by CDSChecker	96
Table 4.7: TxC-ADT Execution Time Results (in hours)	97

LIST OF ALGORTIHMS

Algorithm 1: Type Definitions	26
Algorithm 2: Correctness Checking Algorithm for Concurrent History	28
Algorithm 3: Correctness Checking Algorithm for Implementation	29
Algorithm 4: Type Definitions	42
Algorithm 5: Recursive Topological Sort	43
Algorithm 6: Pruned Recursive Topological Sort	44
Algorithm 7: Correctness Checking Algorithm for Concurrent History	46
Algorithm 8: Correctness Checking Algorithm for Implementation	47
Algorithm 9: Type Definitions	80
Algorithm 10: Conflict Resolution Policy	81
Algorithm 11: Logical Status	82
Algorithm 12: Update NodeInfo	83
Algorithm 13: Template for Transformed Insert Function	85

CHAPTER 1: INTRODUCTION

Designing scalable multiprocessor programs is essential for achieving performance benefits from the hardware developments in increasing the number of cores per chip. The challenge with designing multiprocessor programs is preserving the *safety* and *liveness* properties expected of a program operating in a multi-threaded environment. A safety property defines correct behavior for a multiprocessor program. A liveness property defines the progress guarantee for a multiprocessor program. Traditional multiprocessor programs maintain safety by protecting critical sections with a coarse-grained lock. Since this solution limits scalability, fine-grained locking algorithms have been developed. Fine-grained locking provides improved scalability over a coarse-grained lock because a thread only acquires the locks protecting the memory locations that it accesses, allowing threads that access different parts of the memory to run concurrently. However, the usage of locks is vulnerable to violations of liveness such as *deadlock* (a thread fails to release a lock, halting system-wide progress) or *starvation* (a thread never acquires a lock because it is in use by other threads).

The liveness vulnerabilities associated with locks has inspired non-blocking data structures. Non-blocking data structures achieve safe memory accesses through *Compare-And-Swap* (CAS). CAS is an atomic instruction that accepts a memory location, old value, and new value as parameters. If the contents of a memory location is equivalent to the old value, the memory location is updated to contain the new value and the boolean value true is returned; otherwise, no change is made and false is returned. An update to a memory location is achieved through a loop where CAS is continuously attempted until it succeeds. Progress guarantees have been defined for non-blocking data structures to express the level of progress expected in a multi-threaded system. A non-blocking data

structure is *lock-free* if at least one thread is guaranteed to make progress. A non-blocking data structure is *wait-free* if all threads are guaranteed to make progress.

The difficulty of reasoning about correctness and progress properties of non-blocking data structures originates from all possible ways the threads may interleave. Verification tools are therefore necessary to ensure that non-blocking data structures deliver the safety and liveness properties they are designed to provide. This necessity motivates my thesis on the verification of correctness and progress guarantees for non-blocking data structures.

Correctness

Several correctness conditions have been defined for non-blocking data structures. A *legal sequential history* is a history such that the first event is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response [46]. Linearizability [47] is a correctness condition such that a history consisting of all invocation and response events is equivalent to a legal sequential history, and each method appears to take effect instantaneously at some moment between its invocation event and response event, preserving real-time ordering. Sequential consistency is a correctness condition such that a history consisting of all invocation and response events is equivalent to a legal sequential history, and each method appears to take effect in program order [58]. Linearizability and sequential consistency are both appropriate correctness conditions for different types of systems. Linearizability is more suitable for complex systems with multiple components, while sequential consistency is more suitable for self-contained systems. Other correctness conditions such as quiescent consistency [5] and quasi-linearizability [1] have been introduced to complement the needs of counting networks and complex systems with high performance demands, respectively.

Although non-blocking data structures that are designed for the previously mentioned correctness conditions guarantee safety at the granularity of the data structure operations, the composition of operations may be vulnerable to undefined behavior. For example, consider a linearizable concurrent adjacency list data structure that maintains a list of vertices implemented using the set abstract data type. The code snippet listed in Figure 1.1 presents an example of a composition of the concurrent adjacency list operations that is erroneous.

```
1  if(!graph.vertex_list.contains(key))
2  {
3      value = ... //compute vertex value
4      graph.vertex_list.insert(key, value);
5  }
```

Figure 1.1: Data Structure Composition Example

The code in Figure 1.1 checks the vertex list for a specific key on line 1. If the vertex list does not contain the key of interest, then a new vertex with this key and computed value is inserted in the vertex list on line 4. The fallacy in this logic is that another thread may insert a vertex with the same key of interest and different computed value between the instant that `contains` returns and `insert` is invoked. The previously stored value would be unintentionally overwritten due to the non-atomic composition of operations. To overcome this challenge, my thesis presents Correctness Condition Specification (CCSpec), the first tool that automatically checks the correctness of a composition of concurrent multi-container operations performed in a non-atomic manner. A reference to a container is associated with each method called in a concurrent history to enable the evaluation of correctness for a composition of multiple containers. I develop a lightweight custom specification language that allows the user to define a correctness condition associated with the concurrent algorithm and a correctness condition associated with the concurrent data structures. CCSpec can check non-blocking data structures for a user-specified correctness condition, or

for correctness conditions discussed in literature including linearizability [47], sequential consistency [58], quiescent consistency [5], and quasi-linearizability [1].

The potential violation of data structure semantics associated with a composition of non-blocking data structure operations has inspired transactional data structures. A data structure is considered *transactional* if it supports executing operations atomically and in isolation in a multi-threaded environment. Software Transactional Memory (STM) [91] has been proposed to enable a composition of operations to be executed atomically and in isolation as a software transaction. Using STM to construct transactional data structures is vulnerable to excessive aborts due to read/write conflicts on frequently accessed memory locations such as the head of a linked list. State-of-the-art transactional data structures [107, 94] improve the concurrency control of traditional STM by using high-level semantic conflict detection. *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. High-level semantic conflict detection leverages semantic knowledge of the data structure to provide explicit transactional synchronization for only non-commutative operations. Commutative operations are allowed 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 [44].

The exploitation of data structure semantics substantially improves performance by abandoning the isolation property of low-level reads and writes. Since the read/write histories do not exhibit the isolation property expected from transactional memory systems, the correctness of transactional data structures cannot be judged according to the histories of low-level reads and writes. This presents a challenge for verification techniques [17, 36, 30] that evaluate the correctness of transactional memory systems based on low-level reads

and writes. My thesis addresses this challenge by presenting Transactional Correctness tool for Abstract Data Types (TxC-ADT), the first tool that can check the correctness of transactional data structures. TxC-ADT recasts the standard definitions of transactional correctness in terms of an abstract data type, as introduced in [87]. To accommodate a diverse assortment of transactional correctness conditions, correctness is defined 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 the approach applicable to transactional correctness conditions that do not enforce a total order on a transactional execution. TxC-ADT accommodates a variety of widely-accepted transactional correctness conditions including serializability [80], strict serializability [80], opacity [39], and causal consistency [51].

Progress

The strategies for verifying lock-freedom [34, 49, 53] are centered on CAS-based loops. Since a CAS return value of false indicates that some other thread made progress, the CAS-based loop is capable of achieving lock-freedom. The techniques for verifying lock-freedom assume that a thread exiting a CAS-based loop indicates that system-wide progress is being made. However, data structures that are vulnerable to cyclic dependencies require helping mechanisms to ensure lock-free progress and cannot be verified under the same assumption. Additional reasoning must be applied to verify that the helping mechanism allows lock-free progress.

Thread helping mechanisms are also fundamental for wait-free non-blocking data struc-

tures. Wait-free progress is achieved by allowing a delayed thread to post their operation in a shared array of descriptor objects [43]. Prior to starting their own operation, every thread is required to check the array of descriptor objects to help complete a pending operation. In a worst case scenario, all threads will be recruited to assist the delayed operation such that the helped thread is guaranteed to complete the operation in a finite number of steps. Wait-freedom has previously been considered easier to verify than lock-freedom since wait-freedom is a thread local property. However, the helping mechanism makes it difficult to reason about wait-freedom in a thread local manner.

Progress guarantees for descriptor-based helping schemes such that loop termination is dependent on the actions of the interfering threads are difficult to reason about by existing verification techniques because progress is not ensured simply by the ability of a thread to exit a CAS-based loop. To overcome this challenge, my thesis presents the first progress verification technique that accounts for non-blocking algorithms that require a descriptor-based helping mechanism to achieve the desired progress guarantee. I provide algorithms for the communication through descriptor objects with interfering threads and define a loop invariant specification to prove progress of descriptor-based non-blocking data structures. To verify that all loops in a non-blocking data structure terminate according to the specification, I implement a practical framework that enables the semi-automatic verification of concurrent programs written in the C programming language.

Leveraging Semantics to Increase Performance

Previous research on transactional memory has investigated relaxing atomicity and isolation to address common problems associated with transactional memory. Strategies are

presented that incorporate a cooperative transactional model [33, 74, 102, 54] that violates atomicity and isolation with the benefit of reducing aborts for long-lived transactions. The suspension of isolation is proposed in [82, 93, 65, 60, 64, 101] for transactions that utilize synchronization primitives such as barriers or condition variables. These optimization strategies, which are centered on traditional STM systems, have already been incorporated into the semantic conflict detection of transactional data structures since permitting commutative operations to proceed without transactional synchronization abandons atomicity and isolation at the read/write level. However, the potential benefits of optimizing the semantic conflict detection mechanism have not yet been explored in the literature.

In my dissertation, I present *transactional merging*, a technique that relaxes the semantic conflict resolution of transactional data structures such that a transaction that conflicts with another transaction will merge the conflicting operations into one operation rather than aborting itself. I provide a function that can be configured by the designer to specify which semantic conflicts are eligible to be eliminated by merging operations. Transactional merging is the first technique to propose an optimization for the semantic conflict detection scheme utilized by state-of-the-art transactional data structures. The performance evaluation demonstrates that transactional merging significantly improves the throughput of committed transactions by reducing the total number of aborts. Such performance benefits make transactional merging ideal for high-performance transaction processing required by domains such as in-memory databases [98, 16].

Outline

The remainder of the dissertation is organized as follows. In Chapter 2, I provide definitions for the terminology used within the dissertation. I then discuss related work

regarding correctness and progress guarantees for non-blocking programs and cooperative schemes for transactional applications. In Chapter 3, I present the methodology for my correctness tools CCSpec and TXC-ADT, my progress verification technique and framework for descriptor-based non-blocking programs, and the general approach for the transactional merging technique. In Chapter 4, I provide the experimental results and case studies used to evaluate CCSpec, TXC-ADT, the progress verification technique and framework, and transactional merging. Concluding remarks are provided in Chapter 5.

CHAPTER 2: BACKGROUND

Terminology

In this section, I provide definitions for the terms relevant to a concurrent execution, and correctness and progress properties for the concurrent execution.

Concurrent Execution

An *operation* is a procedure that updates shared data. A *concurrent data structure* is a shared container for data that provides a set of operations, also referred to as *methods*, to manipulate the data [46]. *Atomicity* is a property over a set of operations such that either all operations are committed to memory, or none of the operations are committed to memory. *Isolation* is a property over a set of operations such that the operations appear to take effect in sequential order. A *transaction* is a sequence of operations that appear to be performed atomically and in isolation. An *event* is (1) a change in the status of a transaction including transaction-begin, commit, or abort, or (2) a change in the status of a method including an invocation or response. A *history* is a finite series of instantaneous events [46]. A *sequential history* is a history such that the first event is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response [46]. A *concurrent history* is a history in which the finite series of events are ordered according to a thread schedule.

Correctness Property

Herlihy et al. [47] present a formal method for verifying the correctness of a concurrent data structure which is based on Hoare's [48] formal correctness proof method for data representations. The concurrent data structure is defined in terms of an abstract type ABS and its representation type REP. ABS defines the type being implemented and REP defines the implementation of ABS. The *rep invariant*, denoted as $I: \text{REP} \rightarrow \text{BOOL}$, characterizes the set of REP values that are legal representations. The *abstraction function*, denoted as $A: \text{REP} \rightarrow \text{ABS}$, is a mapping function that maps REP values to ABS values, when the rep invariant is satisfied.

An ABS operation, α , is implemented by a sequence of REP operations, ρ , that carries REP to a legal value. The implementation, ρ , of ABS operation, α , is correct if there exists a rep invariant, I , and abstract function, A , such that when ρ carries a legal REP value r to r' , α carries the ABS value from $A(r)$ to $A(r')$ [47]. Since concurrent operations are permitted to make progress at any instant of the execution, the rep invariant and abstract function must be satisfied continually at each REP operation, rather than satisfied only between ABS operations [47].

Progress Property

Lock-freedom is a property over multiprocessor programs such that at least one thread is guaranteed to make progress. *Wait-freedom* is a property over multiprocessor programs such that all threads are guaranteed to make progress. A concurrent data structure is *blocking* if the delay of any one thread can delay other threads. A concurrent data structure is *non-blocking* if the delay of a thread cannot delay the other threads.

Related Work

In this section, I discuss related work regarding techniques for verifying the correctness of non-blocking data structures, the correctness of transactional memory, and progress guarantees for non-blocking data structures.

Correctness of Non-Blocking Data Structures

Techniques that focus on verifying the correctness of concurrent data structures are proposed by [99, 13, 10, 108, 79]. Vechev et al. [99] present an experience report on verifying linearizability of non-blocking concurrent data structures. The concurrent data structure can be checked for linearizability automatically by using the model checker SPIN [50] to iterate through all permutations of the concurrent history and verify that each permutation matches a legal sequential history. Burckhardt et al. [13] present Line-Up, a complete and automatic tool that checks deterministic linearizability. Baumler et al. [10] use Linear Temporal Logic (LTL) to prove linearizability using the KIV interactive theorem prover. Zhang et al. [108] present Round-up, a runtime verification method for checking quasi-linearizability in the source code implementations of concurrent data structures. Ou et al. [79] present a non-deterministic correctness model that encompasses the relaxed behaviors provided by the C/C++ memory model. The approaches presented in [99, 13, 10, 108, 79] are capable of checking the correctness of concurrent data structures, but do not provide a strategy for checking that a composition of data structure operations preserves the intended semantics of the concurrent algorithm. Shacham et al. [90] present a tool that checks linearizability for a composition of concurrent operations invoked by a single container. CCSpec is able to check that a composition of concurrent operations

invoked by multiple containers meets a correctness condition specified by the user. Such capabilities enable CCSpec to check a high-level concurrent algorithm comprising multiple containers, where the specified correctness condition for the concurrent algorithm may be different from the specified correctness condition of the containers utilized in the algorithm.

Several techniques propose a formal logic for verifying the correctness of concurrent algorithms. Sergey et al. [88, 89] present a framework for the verification of concurrent programs. Oortwijn et al. [78] present an abstraction technique that uses process algebras to describe the behavior of shared-memory concurrent programs. The logic presented in [89, 88, 78] does not evaluate algorithms that use a composition of data structure operations. To use the logic proposed by these techniques for checking the correctness of a composition of data structure operations, the correctness of each data structure operation would need to be proved separately, followed by a proof that the composition of data structure operations satisfies the program invariants and pre-/postconditions of the abstract functions. The drawback of these approaches is that the proofs must be mechanized using manually constructed formal logic.

Generalized verification tools are proposed in [50, 52, 19, 26] that provide a higher degree of automation over fully mechanized proofs using formal logic. Holzmann [50] presents SPIN, a verification tool that can perform bounded model checking and can verify correctness properties specified in LTL. Jacobs et al. [52] present VeriFast, a verification tool for single-threaded and multithreaded C and Java programs. Cohen et al. [19] present VCC, a tool suite for low-level concurrent system code written in C that can prove the correctness of function contracts, state assertions, and type invariants. Dwyer et al. [26] develop the Concurrency Intermediate Verification Language (CIVL). CIVL provides verification and analysis tools for checking properties of programs using a symbolic execution-based

model checker. The challenge with using a generalized verification tool for verifying the correctness of a composition of concurrent data structure operations is that the specifications require additional auxiliary code [89] to define the allowable permutations of the methods called in a concurrent history.

Correctness of Transactional Memory

A significant amount of research focuses on the correctness verification of transactional memory. Several approaches [30, 27, 62] propose automatic techniques to verify correctness of transactional memory systems. Flanagan et al. [30] 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. [27] 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. [62] 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 [71] 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 [75] 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 [31] technique to minimize the exploration of redundant thread schedules. Line-Up [13], a tool that automatically checks deterministic linearizability, uses CHESS [71] 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 [36, 37, 38, 77, 6] propose techniques based on model checking to verify correctness of transactional memory systems. Guerraoui et al. [36] 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. [77] verify the correctness of Intel’s McRT STM [85] using the model checker Spin [50]. Baek et al. [6] 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 which 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 [12, 66, 17, 18, 11, 25, 86] propose a formal logic to verify correctness of transactional memory systems. Blundell et al. [12] 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. [12] 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. [17] 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. [66] 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. [11] 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. [25] 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. [25]’s approach requires that the correctness proofs be constructed manually using formal logic. Schmidt-Schauß et al. [86] 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.

Progress Verification of Non-Blocking Algorithms

Gotsman et al. [34] present a tool that automatically verifies progress guarantees for non-blocking data structures. The verification of lock-freedom is reduced to verifying that all threads terminate regardless of the environment interference. Tofan et al. [96] present a technique for verifying lock-freedom based on rely-guarantee reasoning with interval temporal logic. The authors develop a decomposition theorem for lock-freedom which states that the continuous fulfillment of the rely condition by the global environment implies that the existence of an active operation will result in the completion of some active

operation. Since these approaches do not account for loop termination guaranteed through strategic maintenance of auxiliary structures, they are not applicable to non-blocking data structures that achieve progress through descriptor-based helping mechanisms.

Hoffmann et al. [49] present a quantitative compensation scheme for the verification of lock-freedom. Their technique is based on the intuition that a thread that successfully makes progress in an operation has to provide resources to the other threads for the interference that it caused. The limitation of this technique is that it assumes that a thread exiting a CAS-based loop implies that the thread will never revisit the loop due to a cyclic dependency. A quantitative compensation scheme does not adequately verify progress for non-blocking data structures vulnerable to cyclic dependencies because such designs guarantee progress by detecting and preventing the cyclic dependency rather than by reasoning that a failed CAS implies that another CAS succeeded and made system-wide progress.

Jia et al. [53] propose a technique that instruments the source code with assignments to auxiliary variables and uses assertions to verify lock-freedom. The limitation of this approach is that it is designed for the “read, compute, and update” loop pattern. The authors provide a refinement to their proof method to handle loops that do not terminate in a single iteration with no contention. However, if a looping method of a concurrent library is wrapped inside a terminating loop, this technique will fail to prove that the concurrent library is lock-free.

The approaches presented by Gotsman et al. [34], Hoffmann et al. [49], and Jia et al. [53] each verify lock-freedom of Hendler’s elimination-backoff stack [42]. The elimination-backoff stack uses a helping scheme to improve performance by maintaining a collision layer where threads that fail to perform their operation announce their operation by

writing their id to the collision array by applying CAS. An attempt is made to eliminate the operations of the previous thread and the new thread written in the collision array if they are complementary operations. Although the elimination-backoff stack uses a helping scheme to remove complementary operations, the helping scheme is irrelevant to lock-free progress. The elimination-backoff stack guarantees lock-freedom because if a CAS performed to eliminate an operation fails, then some other thread must have collided with the operation or the operation is not available for collision.

The program logic Total-TaDA is presented by da Rocha Pinto et al. [23] to verify that concurrent programs both terminate and produce the correct result. The logic extends the concurrent program logic TaDA [22] with well-founded termination reasoning to verify progress properties such as lock-freedom and wait-freedom. The authors parameterize the loop invariant of a relation with an ordinal number that places a bound on the number of times a CAS can fail, which is not adequate for lock-free designs that require helping mechanisms to prevent cyclic dependencies that may occur in CAS-based loops. Additionally, Total-TaDA's parameterization of the loop invariant assumes that wait-freedom is a thread local property, while descriptor-based wait-free data structures are dependent on each thread helping a delayed thread.

Liang et al. [61] present a simulation RGSim-T that verifies termination-preserving refinement of concurrent programs. RGSim-T ensures that the target program preserves the termination/divergence behaviors of the source program by parameterizing the simulation with the environment interference that specifies which environment steps may make the current thread take more silent steps due to a failed CAS that corresponds to no additional steps by the source. Although RGSim-T's simulation and logic are general enough to verify lock-freedom, their parameterization of the environment interference places a bound on the number of loop iterations under the assumption that failure to

exit a loop implies that some other thread exited the loop and will not perform additional silent steps. Such logic is adequate for most lock-free designs but does not account for data structures that require helping mechanisms to achieve lock-freedom. The authors present a proof technique for wait-freedom that does not allow an environment step to increase the number of silent steps due to additional loop iterations, which is too restrictive for descriptor-based wait-free data structures.

Cooperative Schemes for Transactional Applications

Several approaches [33, 74, 102, 54] present a cooperative transactional strategy that relaxes isolation and atomicity to assist large transactions to successfully commit to memory. Garcia-Molina et al. [33] divide a long-lived transaction into a sequence of smaller transactions that commit individually to improve performance. If any of the smaller transactions abort, compensating transactions are executed to undo the effects of the other committed transactions. Nodine et al. [74] define a transaction framework, Cooperative Transaction Hierarchy, that relaxes atomicity and serializability to support cooperative applications. Weikum et al. [102] propose relaxing atomicity and isolation in the open nested transaction model. Isolation is relaxed by allowing uncommitted updates by incomplete transactions to be visible to other transactions if the operations commute with the partial updates. Atomicity is relaxed by lifting the “all-or-nothing” property to the system state in terms of abstract operations as opposed to the low-level reads/writes. The nested transaction model is further enhanced by an STM implementation [73], an extension of the Java programming language [15], and a hybrid HTM and STM scheme [14] that supports open and closed nested transactions. Kaiser et al. [54] introduce transaction restructuring operations, including split-transaction and join-transaction, to accommodate transactions that comprise sections of operations that are independent. The transactional merging

technique builds upon the previous approaches [33, 74, 102, 54] by relaxing the semantic conflict resolution of transactional data structures by merging conflicting operations to improve the system commit rate.

Relaxing isolation and incorporating a cooperative transactional strategy has been explored [93, 65, 60, 64, 101] to overcome incompatibilities between transactions and synchronization mechanisms. Luchangco et al. [65] propose transaction communicators to overcome the incompatibilities of barriers and condition variables with isolated transactions. Transaction communicators are objects that allow concurrent transactions to communicate such that a transaction may only commit if all transactions that observe its effects must also commit. Luchangco et al. [64] propose xCondition, a condition variable compatible with transactions such that it does not abort a transaction that waits on it. The xCondition variable allows an active waiting transaction to receive a notification from another active waiting transaction. The waiter and notifier will either both commit or both abort. Unlike the previous approaches [65, 64], transactional merging does not force the collaborating transactions to either both commit or both abort, which provides higher throughput by maximizing committed transactions.

Smaragdakis et al. [93] present a concurrent programming model, Transactions with Isolation and Cooperation (TIC), that addresses the problem of I/O operations that cannot be undone by a rollback. TIC allows transactions to cooperate by temporarily suspending the atomicity and isolation properties of a transaction within an atomic block. Lesani et al. [60] present Communicating Memory Transactions (CMT), a transactional memory model that provides opacity and safe asynchronous message passing to ensure that every committed transaction has only received messages from committed transactions. Wang et al. [101] present an implementation of condition variables that is compatible with locks, hardware transactional memory, and software transactional memory. Wait and notify algorithms are

provided that enable safe thread communication through condition variables within the transactional region. Transactional merging extends these techniques [93, 65, 60, 64, 101] to offer a collaboration strategy that is focused on improving performance.

Several approaches combine operations to improve performance for concurrent data structures [42, 41, 7]. Hendler et al. [42] present the elimination backoff stack, an algorithm that enables `PUSH` and `POP` operations to exchange values without modifying the shared lock-free stack via a collision array. The algorithm retains linearizable stack semantics because eliminated `PUSH` and `POP` operations do not change the abstract state of the data structure. Hendler et al. [41] present flat combining, a technique that enables threads to write their operation to a thread-local publication record that is applied to the shared data structure by a combiner thread that acquires a lock, scans the publication record, combines the requests, modifies the data structure according to the requests, then releases the lock. The requests are combined such that multiple requests can be fulfilled over a single pass of the data structure, reducing synchronization overhead and overall time complexity compared to performing the operations individually. Bar-Nissan et al. [7] present a dynamic elimination-combining stack, an algorithm that eliminates operations that have reverse semantics and combines operations that have identical semantics. Transactional merging is comparable to the dynamic elimination-combining stack [7] because it combines operations that have identical semantics with the discerning feature of optimizing transaction throughput.

CHAPTER 3: METHODOLOGY

CCSpec: A Correctness Condition Specification Tool

In this section, I propose Correctness Condition Specification (CCSpec), the first tool that automatically checks the correctness of a composition of concurrent multi-container operations performed in a non-atomic manner. Concurrent algorithms such as parallel simulation techniques [55] and parallel machine learning [63] require a composition of multiple concurrent containers of various types. CCSpec checks the correctness of a composition of multiple containers by associating a reference to a container for each method called in a concurrent history.

CCSpec accommodates existing as well as new correctness conditions through a technique that characterizes a correctness condition as a *happens-before* relation. The happens-before relation is a partial ordering between two method calls invoked in a concurrent history. A happens-before graph representing the partial ordering of all methods called in a history is constructed automatically during model checking. All possible legal sequential histories are derived from the happens-before graph through a recursive topological sort algorithm. A unit test is correct if all concurrent histories are equivalent to a legal sequential history.

CCSpec will provide a strong impact on the optimization of a concurrent system. To improve performance, it may be beneficial to adopt a relaxed correctness condition that is tailored for the current needs of the system. Data structures such as a k-FIFO queue [56] and a quiescently consistent priority queue [106] both have demonstrated significant performance benefits with a design that conforms to a relaxed correctness condition. CCSpec will allow the user to explore potential performance gains in a concurrent algorithm by

checking if the usage of a data structure designed for a relaxed correctness condition will violate correctness at the abstract function layer. Such optimizations are also useful for data structure library designers that strive to find the optimal balance between correctness and performance.

The contributions of this work include:

1. The first tool that automatically checks the correctness of a composition of concurrent multi-container operations performed in a non-atomic manner. Existing tools for checking the correctness of data structures [99, 13, 10, 108, 79] do not provide the ability to check that a composition of data structure operations used for a high-level concurrent algorithm exhibits correct behavior.
2. A technique for defining a correctness condition for concurrent data structures as a happens-before relation. CCSpec can verify a broad assortment of specifications ranging from traditional correctness conditions such as linearizability to the uncommon quasi-linearizability.
3. I demonstrate the practical application of CCSpec by checking the correctness of a variety of concurrent algorithms. The experimental evaluation explores correctness for concurrent algorithms that utilize data structures designed for a relaxed correctness condition. Such data structures include a priority queue [106] that meets the quiescent consistency correctness condition and a k-FIFO queue [56] that meets the quasi-linearizable correctness condition.

Data Structure and Abstract Function Layers

An *abstract function* is defined as a user-specified composition of data structure operations performed by a thread. The correctness checking algorithm is divided into two layers: the data structure layer and abstract function layer. This strategy allows the design flaw to be identified as an incorrect data structure, an incorrect usage of the data structure operations, or a combination of both. Evaluating the correctness of a composition of data structure operations in a concurrent algorithm can be problematic if the correct behavior defined for the algorithm is different than the correct behavior defined for the concurrent data structures. I address this challenge by developing a lightweight specification language that allows the user to define a correctness condition associated with the data structure layer and a correctness condition associated with the abstract function layer. The lightweight custom specification language is designed such that it can be integrated into model checking tools to enable the expression of concurrent histories in terms of the data structure method calls and abstract function calls.

```
1  template<typename T>
2  T Method(T x1, T x2){
3      begin(&Container, &Method, 2, x1, x2);
4      //Method body
5      end(&Container, &Method, 1, y);
6      return y;
7  }
```

Figure 3.1: Method Call Annotation Example

An example of the annotation usage at the data structure layer is shown in Figure 3.1. The invocation of a method call is specified by passing a reference to a container, a function pointer of the method, the number of inputs, and associated inputs to the `begin` function on line 3. The response of a method call is specified by passing a reference to a container, a function pointer of the method, the number of outputs and associated outputs to the `end` function on line 5. The annotation usage at the abstract function layer is similar to the

example in Figure 3.1, except the `begin_abstract` function is called instead of the `begin` function and the `end_abstract` function is called instead of the `end` function.

The information from each annotation is extracted during model checking and stored in an action object, shown in Algorithm 1. The *ConcurrentHistory* type on line 1.32 is a list of action objects that represents a single generated concurrent history. A method descriptor is created for each method call and each abstract function call in a concurrent history. An *active* status indicates that a method call or abstract function call is in progress. An *inactive* status indicates that a method call or abstract function call is not in progress. A method descriptor contains a unique identification number, the method status (active, or inactive), a sequence number for the beginning and ending of a method call or abstract function call, a reference to the container that the method call or abstract function call is invoked upon, a function pointer to the corresponding method or abstract function, and associated input and observed output values, as shown in Algorithm 1.

The *method_map* on line 1.29 maps each method call to a unique identification number. The *abstract_func_map* on line 1.30 maps each abstract function call to a unique identification number. The *LegalHistory* type on line 1.33 is a list of method identification numbers that corresponds to a legal ordering of the method calls or abstract function calls according to the correctness condition.

ALGORITHM 1: Type Definitions

```
1 typedef int MethodId;
2 enum ActionType
3 |   Invocation;
4 |   Response;
5 |   Correctness_Condition;
6 |   Abstract_Invocation;
7 |   Abstract_Response;
8 |   Abstract_Correctness_Condition;
9 struct ActionObject
10 |   int sequence_number;
11 |   ActionType type;
12 |   int tid;
13 |   MethodId method_id;
14 |   void *(method)(int64_t);
15 |   void *input;
16 |   void *observedOutput;
17 enum MethodStatus
18 |   INACTIVE;
19 |   ACTIVE;
20 struct MethodDesc
21 |   MethodId id;
22 |   MethodStatus status;
23 |   int begin;
24 |   int end;
25 |   void *container;
26 |   void *(func_ptr)(int64_t);
27 |   void *input;
28 |   void *observedOutput;
29 Map<MethodId, MethodDesc> method_map;
30 Map<MethodId, MethodDesc> abstract_func_map;
31 typedef List<List<MethodId>> Graph;
32 typedef List<ActionObject> ConcurrentHistory;
33 typedef List<MethodId> LegalHistory;
```

Approach for Checking Correctness

CCSpec characterizes a correctness condition according to a happens-before relation, which is a partial ordering on the methods called in a concurrent history. By developing a tool that enables the user to specify the allowable method call ordering, a concurrent data structure can be checked for any correctness condition. Such capabilities are essential for accommodating diverse concurrent systems in which a non-conventional correctness condition may be more suitable than the standard correctness conditions.

Definition 3.0.1. The *happens-before* relation, denoted $<_H$, is a partial order defined over the set of method calls in a history h such that for any two method calls m_1 and m_2 , if $m_1 <_H m_2$, then the response event of method call m_1 precedes the invocation event of method call m_2 in history h .

Definition 3.0.2. The *happens-before graph* is a directed graph such that for any two method calls m_1 and m_2 in history h , if an edge exists from m_1 to m_2 , then $m_1 <_H m_2$.

CCSpec maintains a separate happens-before graph for the data structure method calls and abstract functions calls in a concurrent history. The happens-before graph is maintained as a two-dimensional list of identification numbers, as shown on line 1.31. A similar approach for maintaining the ordering constraints of a correctness condition as a happens-before graph was presented by Peterson et al. [81]. However, their approach is specifically designed for transactions, and cannot verify correctness of a high-level concurrent algorithm that invokes the transactions.

Algorithm 2 presents the algorithm for checking a correctness condition. The `IsHistoryCorrect` function generates the happens-before graph on line 2.2 from the *ConcurrentHistory* object and the correctness condition specification. A recursive topological sort on the graph, shown on line 2.3, computes all possible legal sequential histories of the method calls or abstract function calls. For each possible legal sequential history, the concurrent output and sequential output are generated from the method descriptor *MethodDesc* detailed in Algorithm 1. For each method call in a legal sequential history, the observed output is amended to the concurrent history on line 2.8, and the method's function pointer is invoked on line 2.9 and amended to the sequential output on line 2.10.

The comparison between concurrent history output and legal sequential history output is performed on line 2.11. If the concurrent history output and legal sequential history output are equivalent, then the individual concurrent history is correct and `IsHistoryCorrect` returns true. Otherwise, the counterexample is documented on line 2.12 and the for-loop on line 2.4 iterates through the remaining legal sequential histories. The concurrent history is not correct if the concurrent history output is not equivalent to any legal sequential history output. In this case, `IsHistoryCorrect` returns false and the counterexamples collected are reported to the user at the end of the correctness checking algorithm.

ALGORITHM 2: Correctness Checking Algorithm for Concurrent History

```
1 Function IsHistoryCorrect(ConcurrentHistory* history)
2   Graph  $g \leftarrow \text{GenerateGraph}(\text{history})$ ;           // Generate happens-before graph
3   LegalHistory  $S[] \leftarrow \text{RecTopologicalSort}(g)$ ;      // Generate set of legal sequential
   histories
4   foreach  $s \in S[]$  do
5     list concurrent_output;
6     list sequential_output;
7     foreach method  $\in s$  do
8       concurrent_output.push_back(method.ObserveOutput); // method's observed output
9       void *temp = (*method.func_ptr)(method.container, method.input); // Invoke method
       sequentially
10      sequential_output.push_back(temp); // method's sequential output
11      if concurrent_output == sequential_output then
12        return true; // Concurrent output is equivalent to a legal sequential
        history
      // Document counterexample
    // Report all documented counterexamples
13 return false; // Concurrent output is not equivalent to a legal sequential history
```

Algorithm 3 presents the algorithm for checking the correctness of a unit test. The `IsUnitTestCorrect` generates all concurrent histories from a model checker for the unit test parameter on line 3.2. The `foreach`-statement on line 3.4 checks if each concurrent history generated from a model checker is correct. The unit test meets the correctness condition if all concurrent histories are correct. The outcome of the `IsUnitTestCorrect` function is relevant only to the unit test because the correctness evaluation is limited to the generated concurrent histories. To evaluate correctness for an implementation, the unit test must include all methods and a minimal set of inputs such that all behaviors of the implementation are explored. The correctness of CCSpec's approach is provided in Appendix A.

ALGORITHM 3: Correctness Checking Algorithm for Implementation

```
1 Function IsUnitTestCorrect(UnitTest m)
2   ConcurrentHistory H[]  $\leftarrow$  GenerateConcurrentHistories(m) ;    // Generate concurrent
   histories from unit test
3   bool outcome = true;
4   foreach h  $\in$  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
```

Specification Language

The context-free grammar for CCSpec's custom specification language, presented in Figure 3.2, is described using the Backus-Naur form (BNF). The specification language enables the retrieval of data from the concurrent history, which is a list of *ActionObjects*, defined on line 1.32 of Algorithm 1. The expression on line 19 retrieves the size of the concurrent history. The expression on line 20 retrieves the number of threads in the concurrent history. The method id of a method call or abstract function call at sequence number x in the concurrent history can be retrieved from the expression on line 21. The thread id of a method call or abstract function call at sequence number x in the concurrent history can be obtained by the expression on line 22.

A happens-before relation can be placed between two method calls using the expression on line 6. A happens-before relation can be placed between two abstract function calls using the expression on line 7. To determine if a happens-before relation exists between two method calls or abstract function calls, information relevant to the correctness condition being evaluated must be extracted. The expression *is_active*(x, j) on line 25 determines if a method call or abstract function call is in progress at sequence number x by thread j in the concurrent history. This information is relevant for correctness conditions that

place an ordering constraint on method calls if the entire system encounters a period of inactivity, such as quiescent consistency. A real-time ordering between method calls can be determined by the expression on line 26, which evaluates to true if the response of the first method call occurs before the invocation of the second method call. A real-time ordering constraint is placed on method calls for correctness conditions such as linearizability and quiescent consistency.

```

1 <program> ::= <function>
2 <function> ::= <function> <stmt> | /* NULL */
3 <stmt> ::= ';'
4 | <expr> ';'
5 | <variable> '=' <expr> ';'
6 | <expr> 'happens_before' <expr> ';'
7 | <expr> 'happens_before_abstract' <expr> ';'
8 | <expr> 'commutes_with' <expr> ',' '(' <expr> ')'
   ↳ ','
9 | 'if' '(' <expr> ')' <stmt>
10 | 'if' '(' <expr> ')' <stmt> 'else' <stmt>
11 | 'for' '(' <stmt> <expr> ';' <stmt_partial> ')'
   ↳ <stmt>
12 | '{' <stmt_list> '}'
13 <stmt_list> ::= <stmt> | <stmt_list> <stmt>
14 <stmt_partial> ::= <variable> '=' <expr> | <
   ↳ variable> '++' | '++' <variable>
15 | <variable> '--' | '--' <variable>
16 <expr> ::= <integer>
17 | <variable>
18 | <expr> <operator> <expr>
19 | 'history' '->' 'size' '(' ')'
20 | 'history' '->' 'num_threads' '(' ')'
21 | 'method' '(' <expr> ')'
22 | 'tid' '(' <expr> ')'
23 | 'container' '(' <expr> ')'
24 | 'input' '(' <expr> ')'
25 | 'is_active' '(' <expr> ',' <expr> ')'
26 | <expr> 'precedes' <expr>
27 | 'forall' <variable> ':' <expr> '..' <expr> ','
   ↳ '(' <expr> ')'
28 | 'exists' <variable> ':' <expr> '..' <expr> ','
   ↳ '(' <expr> ')'
29 | '(' <expr> ')'
30 <integer> ::= 0 | [1-9][0-9]*
31 <operator> ::= [+*/< >] | '>=' | '<=' | '!=' | '
   ↳ ==' | '&&' | '//'
32 <variable> ::= [a-zA-Z][a-zA-Z0-9_]*

```

Figure 3.2: Grammar for the Custom Specification Language

The drawback of using a recursive topological sort on the method calls or abstract function calls in a happens-before graph to derive all possible legal sequential histories is that the search space exploration has a worst case time complexity of $O(n!)$. This worst case time complexity is optimized by pruning recursive calls from the search space that would explore a reordering of commutative method calls or commutative abstract function calls. The reduction in the search space due to pruning a reordering of commutative method calls is computed in the following way. Given n method calls, there are $(n - 1)$ positions where commutative method calls are adjacent to each other. For each of these positions, there are $2!$ ways to order the commutative method calls and $(n - 2)!$ ways to order the

remaining method calls. Since one out of the $2!$ ways to order the commutative method calls must be considered, the worst case time complexity when pruning a reordering of commutative method calls is $n! - (n-1) \cdot (2! - 1) \cdot (n-2)!$. After simplification, the worst case time complexity when pruning a reordering of commutative method calls is presented in Equation 3.1.

$$n! - (n-1)! \quad (3.1)$$

This reduction can also be expressed as shown in Equation 3.2.

$$(n-1) \cdot (n-1)! \quad (3.2)$$

The statement on line 8 allows two method calls (or abstract function calls) to be declared as commutative given that the condition in parenthesis is true. The expression on line 23 retrieves the container reference associated with sequence number x in the concurrent history. This information is essential for establishing commutativity because method calls or abstract function calls invoked by different containers are always commutative. The expression on line 24 retrieves the method input associated with sequence number x in the concurrent history. This information is necessary for determining commutativity when method calls commute if they are passed different input, such as the set abstract data type.

The operational semantics of the specification language are provided in Figure 3.3. A state is described by the 4-tuple (M, G, G_{abs}, c) where M is a boolean two-dimensional matrix such that the value at position (i, j) indicates if method call (or abstract function call) i and method call (or abstract function call) j are commutative, G is the happens-before graph

for method calls, G_{abs} is the happens-before graph for abstract function calls, and c is the program statement to evaluate next. A transition from state S_0 to state S_1 is expressed as $S_0 \rightarrow S_1$.

$$\begin{array}{l}
\text{IF} \quad \frac{c' = (b == \text{true}) ? (c_1; c) : (c_2; c)}{(M, G, G_{abs}, \text{if } (b) \{c_1\} \text{ else } \{c_2\}; c) \rightarrow (M, G, G_{abs}, c')} \\
\text{FOR} \quad \frac{c' = (b == \text{true}) ? (\text{for } (c_i; b; c'_i) \{c_1\}; c) : (c)}{(M, G, G_{abs}, \text{for } (c_i; b; c'_i) \{c_1\}; c) \rightarrow (M, G, G_{abs}, c')} \\
\text{HAPPENS_BEFORE} \quad \frac{E' = E \cup \{e_1, e_2\}; G' = (E', V); c}{(M, G, G_{abs}, e_1 \text{ happens_before } e_2; c) \rightarrow (M, G', G_{abs}, c)} \\
\text{HAPPENS_BEFORE_ABSTRACT} \quad \frac{E' = E \cup \{e_1, e_2\}; G'_{abs} = (E', V); c}{(M, G, G_{abs}, e_1 \text{ happens_before_abstract } e_2; c) \rightarrow (M, G, G'_{abs}, c)} \\
\text{COMMUTES_WITH} \quad \frac{c' = (b == \text{false}) ? (E' = E \setminus \{e_1, e_2\}; M' = (E', V); c) : (M' = M; c)}{(M, G, G_{abs}, e_1 \text{ commutes_with } e_2, (b); c) \rightarrow (M', G, G_{abs}, c)}
\end{array}$$

Figure 3.3: Operational Semantics for the Custom Specification Language

Linearizability

Definition 3.0.3. A history h is *linearizable* if the subsequence of h consisting of all events is equivalent to a legal sequential history, and each method call appears to take effect instantaneously at some moment between its invocation event and response event, preserving real-time ordering[46].

The specification for linearizability is shown in Figure 3.4. Since linearizability requires that the effects of the method calls are equivalent to a legal sequential history, the specification must indicate the circumstances for which a method call precedes another method call. The two for-loops on lines 1 and 2 iterate through the action objects of the concurrent history. The parameters for the for-loop on line 2 guarantee that m is strictly less than n . The condition on line 3 checks if the response event of the method called at sequence number m occurs before the invocation event of the method called at sequence number n . If this condition is satisfied, a happens-before relationship can be placed between the

methods called at sequence numbers m and n . Since overlapping method calls will not be constrained by a happens-before relationship, the generated legal sequential histories will represent all possible combinations in which these overlapping method calls could be ordered. The output is compared to each of these legal sequential histories, which eliminates the requirement for knowledge of the linearization points.

```

1  for(n = 0; n < history->size(); n++) {
2      for(m=0; m < n; m++) {
3          if(method(m) precedes method(n)) {
4              method(m) happens_before method(n);
5          }
6      }
7  }

```

Figure 3.4: Linearizability Specification

Sequential Consistency

Definition 3.0.4. A history h is *sequentially consistent* if the subsequence of h consisting of all events is equivalent to a legal sequential history, and each method call appears to take effect in program order[46].

The specification for sequential consistency is shown in Figure 3.5. Sequential consistency is similar to linearizability except it only places an ordering constraint on methods called by the same thread. The if-statement on line 3 checks if the method calls at sequence numbers m and n are called by the same thread. Given that all conditions of the if-statement are satisfied, a happens-before relationship can be placed between the methods called at sequence numbers m and n .

```

1  for(n = 0; n < history->size(); n++) {
2      for(m=0; m < n; m++) {
3          if((tid(m) == tid(n)) && (method(m) precedes method(n))) {
4              method(m) happens_before method(n);
5          }
6      }
7  }

```

Figure 3.5: Sequential Consistency Specification

Quiescent Consistency

Definition 3.0.5. A history h is *quiescently consistent* if the subsequence of h consisting of all events is equivalent to a legal sequential history, and each method call appears to take effect in real-time order if separated by a period of quiescence[46].

The specification for quiescent consistency is shown in Figure 3.6. Lines 1 through 3 are very similar to linearizability such that the response event of the method called at sequence number m occurs before invocation event of the method called at sequence number n . However, an additional condition needs to be placed that evaluates to true if there exists a period of quiescence. This condition is stated in lines 4 through 6. If there exists some atomic step between action objects m and n in which all threads are inactive, then the quiescence condition is satisfied. Given that all conditions of the if-statement are satisfied, a happens-before relationship can be placed between the methods called at sequence numbers m and n .

Quasi-Linearizability

Definition 3.0.6. A history h is *quasi-linearizable* if the subsequence of h consisting of all events is equivalent to a legal sequential history, and method calls separated by a distance of length k should appear to take effect in real-time order[1].

```

1  for(n = 0; n < history->size(); n++) {
2    for(m=0; m < n; m++) {
3      if(method(m) precedes method(n) &&
4         exists k: m .. n
5         (forall j: 0 .. history->num_threads()
6          (is_active(k, j) == false)))
7        {
8          method(m) happens_before method(n);
9        }
10   }
11 }

```

Figure 3.6: Quiescent Consistency Specification

The specification for quasi-linearizability is shown in Figure 3.7. The variable k is set to an arbitrary constant on line 1. The two for-loops on lines 2 and 4 iterate backwards through the action objects of the concurrent history. The parameters for the for-loop on line 4 guarantee that m is strictly less than n . The condition of the if-statement on line 5 checks if the response event of the method call at sequence number m occurs before the invocation event of the method call at sequence number n . If the if-statement is satisfied, the counter i is incremented and a happens-before relationship is only assigned if the method called at sequence number m is separated by a distance of k with the method called at sequence number n .

```

1  k = CONSTANT;
2  for(n = history->size() - 1; n > 0; n--) {
3    i = 0;
4    for(m = n - 1; m >= 0; m--) {
5      if(method(m) precedes method(n)) {
6        i = i + 1;
7        if(i > k)
8          method(m) happens_before method(n);
9      }
10   }
11 }

```

Figure 3.7: Quasi-Linearizability Specification

A Transactional Correctness Tool for Abstract Data Types

In this section, I propose Transactional Correctness tool for Abstract Data Types (TxC-ADT), the first tool that can check the correctness of transactional data structures. Correctness is evaluated based on the abstract data type history rather than the read/write history. I address several challenges in order 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. I address this challenge by providing the user with lightweight annotations that identify the invocation and response of an abstract data type. The model checker CDSChecker [75] is utilized 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. I 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. The recursive topological sort is optimized 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. I 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 [72, 100, 2] and data analysis [104, 103] 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. I demonstrate the practical applications of TxC-ADT by checking the correctness of the transactional data structures presented by Zhang et al. [107] and Spiegelman et al. [94].

My dissertation makes the following contributions:

1. I present the first tool that can check the correctness of transactional data structures. Correctness is evaluated based on an abstract data type, making the 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 tech-

niques impractical for state-of-the-art transactional data structures.

2. I 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. I 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. This is accomplished by allowing the user to specify the conditions for which two operations commute.
4. I 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 my knowledge, this is the first verification strategy capable of checking a transactional memory system for causal consistency.
5. I 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 [107] and Transactional Data Structure Libraries [94].

General Approach

The correctness of transactional data structures is evaluated by elevating the standard definitions of transactional correctness to be properties on an abstract data type [87]. The user is provided with lightweight annotations to indicate the invocation and response of a method invoked on an abstract data type and use the model checker CDSChecker [75] to generate the concurrent histories based on the defined abstract data type. An example of the annotation usage is shown in Fig. 3.8. 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.

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

Figure 3.8: Abstract Data Type Annotation Example

The 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 4. The *ConcurrentHistory* type on line 4.36 is a list of action objects that represents a single generated concurrent history. To facilitate the correctness checking algorithm presented in Algorithm 2, a transaction descriptor is assembled 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 terminated 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. 4. Definitions that are fundamental for the correctness checking strategy used by TxC-ADT are now provided:

Definition 3.0.7. 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 3.0.8. 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 4.34 maps each transaction to a unique identification number in order to maintain the transactional happens-before graph as a two-dimensional list of transaction ids, as shown on line 4.35. The *LegalHistory* type on line 4.37 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.

If the transaction status is aborted, a list of inverse operations is maintained on line 4.33 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 that its effects do not propagate throughout the remaining generated legal sequential history.

A Unification of Transactional Correctness Conditions

In order 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, TxC-ADT is designed to accommodate well-known transactional correctness conditions including serializability, strict serializability, opacity, and causal consistency.

ALGORITHM 4: Type Definitions

```
1 typedef int TxnId;
2 enum TxStatus
3 |   Active;
4 |   Committed;
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;
```

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 [30]. 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.

The 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 Fig. 3.9. 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. All possible legal sequential histories can be derived for the transactional execution by applying a recursive topological sort to the transactional happens-before graph.

```

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

Figure 3.9: Correctness Condition Declaration

Algorithm 5 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. This time complexity can be reduced by pruning the recursive topological sort to not explore ordering variations for commutative transactions.

ALGORITHM 5: 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  $n \in N$  do
6           $\lfloor$  PrunedRecTopologicalSort( $S[], n, L, N, g$ );
7       $\lfloor$  return  $S[]$ ; // Return all legal sequential histories
```

Algorithm 6 presents the pruned recursive topological sort function called within Algorithm 5. 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 con-

tains 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. 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, the orderings in which $L.back() < n$ could also be chosen to not be explored. If pruning is not possible, then all edges m outgoing from transaction n are removed from an updated graph g' on line 6.8 and the pruned recursive topological sort function is called on the updated list N of transactions with no incoming edges.

ALGORITHM 6: Pruned Recursive Topological Sort

```

1 Function PrunedRecTopologicalSort(LegalHistory  $S[]$ , TxnId  $n$ , list  $L$ , list  $N$ , Graph  $g$ )
2   if ( $L.size() \neq 0$ ) && (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  $m \in g'[n]$  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  $n' \in N$  do
12    PrunedRecTopologicalSort( $S[], n', L, N, g'$ );
13  if  $N.size() == 0$  then
14     $S[].push\_back(L)$ ;
15  return;

```

The algorithm for checking a correctness condition is presented in Algorithm 7. The

`IsHistoryCorrect` function generates the transactional happens-before graph on line 7.2 from the *ConcurrentHistory* object in conjunction with the correctness condition specification. A recursive topological sort on the graph, shown on line 7.3, 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 4. For each method in a transaction's *method_list*, the observed output is amended to the concurrent history on line 7.9, and the method's function pointer is invoked on line 7.10 and amended to the sequential output on line 7.11. 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 [44].

The concurrent history output is compared with the legal sequential history output on line 7.15. 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 7.4 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 condi-

tions preserve atomicity and isolation, the approach may be extended to other correctness conditions that may be adopted in the advancement of transactional data structures.

ALGORITHM 7: Correctness Checking Algorithm for Concurrent History

```

1 Function IsHistoryCorrect(ConcurrentHistory* history)
2   Graph  $g \leftarrow \text{GenerateGraph}(\text{history})$ ; // Generate transactional happens-before
   graph
3   LegalHistory  $S[] \leftarrow \text{RecTopologicalSort}(g)$ ; // Generate set of legal
   sequential histories
4   foreach  $s \in S[]$  do
5     list concurrent_output;
6     list sequential_output;
7     foreach  $\text{txn} \in s$  do
8       foreach  $m \in \text{txn.method\_list}$  do
9         concurrent_output.push_back(m.observedOutput); //  $m$ 's observed output
10        uint64_t  $\text{temp} = (*m.\text{func\_ptr})(m.\text{input})$ ; // Invoke  $m$  sequentially
11        sequential_output.push_back(temp); //  $m$ 's sequential output
12        if  $\text{txn.status} \neq \text{COMMITTED}$  then
13          foreach  $m\_inv \in \text{txn.method\_list\_inv}$  do
14             $(*m\_inv.\text{func\_ptr})(m\_inv.\text{input})$ ; // Invoke  $m\_inv$  sequentially
15        if  $\text{concurrent\_output} == \text{sequential\_output}$  then
16          return true; // Concurrent output is equivalent to a legal sequential
           history
17        else
18          // Document counterexample
           // Report all documented counterexamples
18   return false; // Concurrent output is not equivalent to a legal sequential
           history

```

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 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 8. 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 8.2. The `foreach`-statement on line 8.4 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. The correctness of TXC-ADT's approach is provided in Appendix B.

ALGORITHM 8: Correctness Checking Algorithm for Implementation

```

1 Function IsUnitTestCorrect(UnitTest m)
2   ConcurrentHistory H[]  $\leftarrow$  GenerateConcurrentHistories(m) ;           // Generate
   concurrent histories from unit test
3   bool outcome = true;
4   foreach h  $\in$  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

```

Specification Language

The context-free grammar for the custom specification language, presented in Fig. 3.10, 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 4.32.

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, 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.

In order 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.

```

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_]*

```

Figure 3.10: Grammar for the Custom Specification Language

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. The 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 7 must be applied to each thread's transactional happens-before graph in order to evaluate cor-

rectness. 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. This worst case time complexity is reduced 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 Fig. 3.11. A state is described by the $(n+3)$ -tuple $(M, G, G_1, \dots, G_n, c)$ 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 $S_0 \rightarrow S_1$.

$$\begin{array}{l}
\text{IF} \quad \frac{c' = (b == \text{true}) ? (c_1; c) : (c_2; c)}{(M, G, G_1, \dots, G_n, \text{if } (b) \{c_1\} \text{ else } \{c_2\}; c) \rightarrow (M, G, G_1, \dots, G_n, c')} \\
\text{FOR} \quad \frac{c' = (b == \text{true}) ? (\text{for } (c_i; b; c'_i) \{c_1\}; c) : (c)}{(M, G, G_1, \dots, G_n, \text{for } (c_i; b; c'_i) \{c_1\}; c) \rightarrow (M, G, G_1, \dots, G_n, c')} \\
\text{HAPPENS_BEFORE} \quad \frac{G' = (E', V), E' = E \cup \{e_1, e_2\}}{(M, G, G_1, \dots, G_n, e_1 \text{ happens_before } e_2; c) \rightarrow (M, G', G_1, \dots, G_n, c)} \\
\text{INSERT_TXN} \quad \frac{G' = (E, V'), V' = V \cup \{e_1\}}{(M, G, G_1, \dots, G_n, \text{insert_txn } e_1; c) \rightarrow (M, G', G_1, \dots, G_n, c)} \\
\text{HAPPENS_BEFORE_PARTIAL} \quad \frac{G'_{e_2} = (E', V), E' = E \cup \{e_1, e_2\}}{(M, G, G_1, \dots, G_{e_2}, \dots, G_n, e_1 \text{ happens_before_partial } [e_2] e_3; c) \rightarrow (M, G, G_1, \dots, G'_{e_2}, \dots, G_n, c)} \\
\text{INSERT_TXN_PARTIAL} \quad \frac{G'_{e_1} = (E, V'), V' = V \cup \{e_2\}}{(M, G, G_1, \dots, G_{e_1}, \dots, G_n, \text{insert_txn_partial } [e_1] e_2; c) \rightarrow (M, G, G_1, \dots, G'_{e_1}, \dots, G_n, c)} \\
\text{COMMUTES_WITH} \quad \frac{c' = (b == \text{false}) ? (M' = (E', V), E' = E \setminus \{e_1, e_2\}) : (M' = M)}{(M, G, G_1, \dots, G_n, e_1 \text{ commutes_with } e_2, (b); c) \rightarrow (M', G, G_1, \dots, G_n, c)}
\end{array}$$

Figure 3.11: Operational Semantics for the Custom Specification Language

Serializability

Definition 3.0.9. A history h is *serializable* if the subsequence of h consisting of all events of committed transactions is equivalent to a legal sequential history [80].

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 Fig. 3.12. 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.

Fig. 3.13 shows the happens-before graph and legal sequential histories generated from the specification of Fig. 3.12 applied to the concurrent history of Fig. 3.14. 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.

```

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

```

Figure 3.12: Serializability Specification

Relation:	Graph:
NULL	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);

```

Figure 3.13: Happens-Before Example for Serializability

Since there are $3!$ ways to order three items, there are six possible legal sequential histories, as shown in Fig. 3.13.

Strict Serializability

Definition 3.0.10. 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 [80].

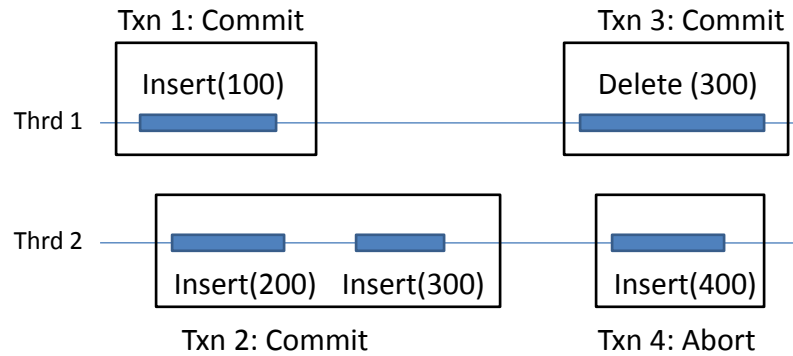


Figure 3.14: Concurrent History Example

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 Fig. 3.15. If a transaction is committed, 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.

Fig. 3.16 shows the happens-before graph and legal sequential histories generated from the specification of Fig. 3.15 applied to the concurrent history of Fig. 3.14. 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 Fig. 3.13.

```

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 }

```

Figure 3.15: Strict Serializability Specification

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

Figure 3.16: Happens-Before Example for Strict Serializability

Opacity

Opacity requires that all transactions (committed, aborted, or active) observe a consistent state of the system. Prior to defining opacity, definitions are provided 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 3.0.11. 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 [39].

Definition 3.0.12. 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' [39].

Definition 3.0.13. 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 . [39].

```

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 }
```

Figure 3.17: Opacity Specification

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 inserted in the transactional happens-before graph regardless of their status. The specification for opacity is shown in Fig. 3.17. 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.

Fig. 3.18 shows the happens-before graph and legal sequential histories generated from the specification of Fig. 3.17 applied to the concurrent history of Fig. 3.14. 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 Fig. 3.18. 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.

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);	

Figure 3.18: Happens-Before Example for Opacity

Causal Consistency

Definition 3.0.14. A *causality relation* consists of operation pairs (X, Y) such that operation X causes operation Y .

Definition 3.0.15. 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 $\text{Complete}(h)$, such that (1) S_i preserves the causality relation, and (2) every committed transaction executed by t_i is legal in S_i . [83].

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 Fig. 3.19. If a transaction is committed, 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, 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.

Fig. 3.20 shows the happens-before graph and legal sequential histories generated from the specification of Fig. 3.19 applied to the concurrent history of Fig. 3.14. The happens-before graph 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 Fig. 3.20. Thread 2's happens-before graph only contains transaction 2, yielding one possible legal sequential history.

```

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 }

```

Figure 3.19: 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);	

Figure 3.20: Happens-Before Example for Causal Consistency

Commutativity Specification

The recursive topological sort in the correctness checking function, detailed in Algorithm 7, is optimized by specifying commutative methods in a transaction. Fig. 3.21 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. Fig. 3.22 shows the *commutes_matrix*, *reorder_matrix*, legal sequential histories, and the pruned legal sequential histories for the example concurrent history in Fig. 3.14. 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 $(1, 2) = \text{true}$, position $(1, 3) = \text{true}$, and position $(2, 3) = \text{false}$. 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 Fig. 3.22. 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.

```

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 }

```

Figure 3.21: Serializability Specification with Commutative Methods Specified

<p>commutes_matrix:</p> <pre> 1 2 3 1 - T T 2 - - F 3 - - - </pre>	<p>reorder_matrix:</p> <pre> 1 2 3 1 - T T 2 - - T 3 - - - </pre>
--	---

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);

```

Figure 3.22: Pruning Example for Serializability with Commutative Methods Specified

Practical Progress Verification of Descriptor-based Non-blocking Data Structures

In this chapter, I present the first progress verification technique that accounts for non-blocking algorithms that require a descriptor-based helping mechanism to achieve the desired progress guarantee. Previous progress verification techniques do not accommodate loops whose termination is dependent on the actions of the interfering threads,

making these approaches unsuitable for descriptor-based non-blocking data structures. A *loop invariant* is an assertion that defines the state of the local and shared resources at each iteration of the loop [3]. A *functional specification* is a mathematical model of a function in a C program expressed in terms of integers, tuples, and sequences [3]. Functional specifications are utilized in the loop invariant to define the changes to the shared state of the auxiliary structure of descriptor objects at every iteration of the loop.

In order to verify that the loop invariant holds, I implement a framework for formally reasoning about practical concurrent programs written in the C programming language. The framework extends the Verified Software Toolchain [3] to accommodate Local-Rely-Guarantee reasoning [29] in order to provide the necessary logic for formally reasoning about concurrent programs. The framework automatically converts a C program to an abstract syntax tree expressed in the formal modeling language of the Coq Proof Assistant [8], enabling the user to directly verify the specification of the concurrent program. A *symbolic execution* is the execution of a program using an abstract characterization of the program state. Once the specifications are written, the framework enables the semi-automatic verification of progress guarantees through the symbolic execution of a concurrent program.

Practical lock-free data structures [24, 106, 107], wait-free data structures [95, 9, 28], and algorithms to facilitate non-blocking programming [70, 32] incorporate thread communication through descriptor objects to ensure that operations comprising multiple atomic steps are completed according to the progress guarantee. Descriptor-based techniques are also utilized in software transactional memory implementations, including Word-Based Software Transactional Memory (WSTM) [32], Object-Based Software Transactional Memory (OSTM) [32], a library of concurrent skip lists and red-black trees built from a multiword compare-and-swap operation [32], Adaptive Software Transactional Memory

[68], Rochester Software Transactional Memory [69], the DSTM2 Software Transactional Memory library [45], and wait-free Software Transactional Memory for hard real-time multicore embedded systems [21]. My thesis makes the following contributions:

1. The first methodology for verifying progress guarantees for non-blocking data structures that require a descriptor-based helping mechanism to achieve the designed progress guarantee.
2. A framework for formally reasoning about concurrent programs written in the C programming language. To my knowledge, this is the first framework that allows for the verification of rely-guarantee specifications in a theorem proving environment for practical concurrent programs written in the C programming language.
3. The presentation of two case studies that demonstrates the application of the methodology to practical descriptor-based non-blocking data structures. I formally verify progress for a lock-free transactional list [107] and a wait-free queue [57]. The helping-mechanisms employed by the lock-free transactional list and wait-free queue are significantly different from each other, demonstrating the versatility of the methodology and framework.

Assertion Language

The assertion language is adapted from Feng [29] as shown in Figure 3.23. A state is a pair of a store and a heap [61]. The store is finite partial mapping from resources to values and the heap maps memory addresses to values [61]. B is a boolean expression that holds over a state if it evaluates to true. The assertion *emp* is the empty heaps and stores. The singleton heap with E_2 stored at location E_1 is denoted as $E_1 \mapsto E_2$. The notation $E \mapsto$

(E_1, \dots, E_n) is used to denote a struct with n fields is stored at location E . The separating conjunction $p * q$ denotes that p and q hold over disjoint parts of the state. Actions specify state transitions. The action $p \bowtie q$ denotes all sub-transitions such that the initial state of the transition satisfies p and the resulting state satisfies q . The action $a_1 * a_2$ denotes that actions a_1 and a_2 start from disjoint states and finish in disjoint states. The action $a_1 \vee a_2$ denotes that either action a_1 occurs or action a_2 occurs.

$$\begin{aligned} (\text{Assertion}) \quad p, q, I : &= B \mid \text{emp} \mid E_1 \mapsto E_2 \mid p * q \\ (\text{Action}) \quad a, R, G : &= p \bowtie q \mid a_1 * a_2 \mid a_1 \vee a_2 \end{aligned}$$

Figure 3.23: The Assertion Language

Verification of Descriptor-Based Helping Techniques

Descriptor-based non-blocking data structures manage interference from multiple active threads by allowing updates to the data structure through CAS-based loops while maintaining auxiliary structures of descriptors that contain information on the operations that require assistance from other threads to be completed. All threads are required to check the auxiliary structure prior to entering a CAS-based loop to determine if an action needs to be taken to preserve the progress guarantee. Since the updates to the auxiliary structure are ultimately what guarantees that the thread whose CAS operation failed will eventually succeed and exit the loop, functional specifications must be incorporated in the loop invariant to define the state of the auxiliary structure at every iteration of the loop. The progress guarantee is verified by proving that the loop invariant holds.

Definition 3.0.16. An *operation* is a procedure that updates a shared data structure atomically using CAS.

Definition 3.0.17. A *transaction* is a sequence of operations that appear to be performed atomically and in isolation.

Definition 3.0.18. A *descriptor object*, denoted $\text{desc}(\text{op})$ is a class object that contains the instructions and arguments required to perform operation op .

Definition 3.0.19. Let s be the contents of an auxiliary structure of descriptor objects. Let desc be a descriptor object for a pending operation. Let an atomic update be either an atomic write or CAS depending on the expected interference by the other threads. A *descriptor update function*, denoted as $\text{desc_update}(s[i], \text{desc})$, comprises the code body that atomically updates auxiliary structure s such that $s[i] \leftarrow \text{desc}$.

Definition 3.0.20. Let s be the contents of an auxiliary structure of descriptor objects. A *helper function*, denoted as $\text{help_finish}(s[i])$, comprises the code body that will continuously perform the operation referenced in descriptor object $s[i]$ until the operation is completed successfully by any arbitrary thread.

Lock-free non-blocking data structures require descriptor-based helping techniques when cyclic dependencies may occur during the execution of a CAS-based loop. Wait-free data structures further require helping techniques to ensure that all threads terminate in a finite number of steps. Algorithms for lock-free and wait-free loops with an arbitrary number of threads are presented in Figure 3.24 and Figure 3.26. Following concurrent separation logic the proof of correctness is applied to a frame [35] with the loop invariant and the auxiliary data structure. The principle of bisimilarity can be applied [4] using the labels in the algorithms to generalize the correctness to any helper function whose effect is contained to the frame and called in the same control flow.

The descriptor-based lock-free algorithm for an arbitrary thread is shown in Figure 3.24. Prior to helping operation op_m , a cyclic dependency check is performed on line 2 by

determining if auxiliary structure s contains descriptor $\text{desc}(\text{op}_m)$. If a cyclic dependency is detected the thread will return to prevent infinite recursive helping calls. Otherwise, the thread will write the descriptor $\text{desc}(\text{op}_m)$ to auxiliary structure s at location $++X$ on line 7. The boolean CAS_RESULT on line 8 represents the return value of operation op_m . At the start of the loop body, the thread will check if a conflict exists between operation op_m and arbitrary operation op_n . Given that a conflict exists, the thread will help complete operation op_n on line 10 prior to performing operation op_m . If the boolean CAS_RESULT on line 12 evaluates to true or $\text{desc}(\text{op}_m)$ is NULL, then operation op_m is completed and the loop terminates. Otherwise, the loop is restarted. Once operation op_m on line 12 is completed, the thread will clear the descriptor from auxiliary structure s on line 14 and set descriptor $\text{desc}(\text{op}_m)$ to NULL on line 15 to indicate that the operation is completed.

```

1  help_finish_lf(Desc * desc(opm)) {
2    if (s contains desc(opm)) {
3      desc_update(desc(opm), NULL);
4      return;
5    }
6    Xinit = ++X;
7    desc_update(s[Xinit], desc(opm));
8    while (!CAS_RESULT && desc(opm) != NULL) {
9      if (conflict with opn) {
10       help_finish_lf(desc(opn));
11     }
12     CAS_RESULT = opm;
13   }
14   desc_update(s[Xinit], NULL);
15   desc_update(desc(opm), NULL);
16 }

```

Figure 3.24: Descriptor-Based Lock-Free Algorithm

Theorem 3.0.1. Let each transaction performed by a thread comprise a finite number of operations. Let the effect of the helper function `help_finish_lf` and descriptor update function `desc_update` be limited to a frame that is a separating conjunction to the loop invariant. Suppose that the loop invariant holds, i.e. the precondition of the loop body

implies the loop invariant, the postcondition of the loop body implies the loop invariant. Then the descriptor-based algorithm of Figure 3.24 is lock-free.

Proof. Correctness proofs for concurrency assume that memory used by a lock is not accessible by the rest of the program. This convention, formalized as a frame in separation logic [84] [76], is followed to include the auxiliary data structure s . Let F be the frame containing the helper function and descriptor update function. In F , let an array of k shared resource descriptors be predefined, denoted by R_k . Allow instantiation of any number of helper function threads T_n , each owning a pending transaction. Allow a transaction to be composed of any number of atomic operations op_i on the shared resources. Assume that the atomic operations called by the helper function are guaranteed to terminate. Assume the loop invariant holds. Linearizability of the helper function operations has been demonstrated in [107]. In the case of no cyclic dependencies the loop becomes a traditional CAS-based loop on the resource.

The proof of lock-freedom uses induction on the number of resources involved in the cyclic dependency. A proof by induction on a statement involves first showing that a base case is true. The inductive hypothesis is an assumption that the statement holds true for n . The inductive step is a claim that the statement holds true for $n + 1$. The proof by induction is completed by using the inductive hypothesis to prove the inductive step. The base case is two resources, the minimum required. Transaction A is exclusively using resource R_1 and requires resource R_2 to finish. Transaction B executes sequentially op_1, op_2, \dots, op_i . Assume op_n of B uses resource R_2 and op_{n+1} requires resource R_1 which is already in use by transaction A. The conflict on R_1 is found (line 9) so B calls the helper function to complete transaction A. This call sees the conflict on R_2 with B and recurses again to help B. Within the second call to B, line 2 detects that B is already being helped,

so the descriptor $\text{desc}(op_{n+1})$ is updated to NULL in line 3, aborting transaction B and freeing the resource R_2 so A can proceed.

For the inductive step assume that A is using resource R_1 and B is using resource R_2 , which A will need to finish (as in the base case). Transaction C enters and takes resource R_3 , which B needs to finish. Let op_n of C need resource R_1 still in use by A. C will call itself to help A and seeing that A needs resource R_2 held by B will recurse again to help B. This call will see the conflict with resource R_3 held by C. Finally, the recursion will call C for the second time and seeing that C is already being helped (line 2), will kill C (line 3). This breaks the cyclic dependency.

The inductive step can be applied from two to the maximum of k resources. The recursion will always terminate upon seeing that a transaction needs help a second time, indicating a cyclic dependency. The depth of recursion is limited to the number of resources, k . The algorithm will continue to make progress after the cyclic dependency is resolved. Since the atomic operations are guaranteed to terminate, lock-freedom is ensured.

□

```

1  help_finish_wf(Desc * desc(op)) {
2      while (!CAS_RESULT && desc(op) != NULL) {
3          CAS_RESULT = op;
4      }
5  }
```

Figure 3.25: Helping Function for Descriptor-Based Wait-Free Algorithm

Wait-free non-blocking data structures require descriptor-based helping techniques to detect a delayed thread that needs help completing an operation. The helper function for a descriptor-based wait-free algorithm is shown in Figure 3.25. A thread will continuously apply the operation on line 3 as defined by the descriptor object until either the operation succeeds or some other thread completes the operation. The descriptor-based wait-free

algorithm for arbitrary thread tid is shown in Figure 3.26. Thread tid will help complete the operation defined by the descriptor at location X in auxiliary structure s on line 1. Once the operation is completed, thread tid will set descriptor $s[X]$ to NULL on line 2. Line 3 defines the update of X to the next location in auxiliary structure s to be helped. The boolean CAS_RESULT on line 5 represents the return value of operation op_{tid} . Thread tid will write its descriptor object $desc(op_{tid})$ to auxiliary structure s at location tid on line 9 if the CAS-based loop exceeds the allowed number of iterations.

```

1  help_finish_wf(s[X]);
2  desc_update(s[X], NULL);
3  X = (X+1)%NUM_THRDS;
4  trials = 0;
5  while (!CAS_RESULT && trials++ < LIMIT) {
6    CAS_RESULT = optid;
7    if(CAS_RESULT) return;
8  }
9  desc_update(s[tid], desc(optid));

```

Figure 3.26: Descriptor-Based Wait-Free Algorithm

Theorem 3.0.2. Let the variable ‘ $trials$ ’ on line 5 of Figure 3.26 be an ordinary conjunction in the loop invariant. Let the effect of the helper function `help_finish_wf` and descriptor update function `desc_update` be limited to a frame that is a separating conjunction to the loop invariant. Suppose that the loop invariant holds, i.e. the precondition of the loop body implies the loop invariant, the postcondition of the loop body implies the loop invariant. Then the descriptor-based algorithm of Figure 3.26 is wait-free.

Proof. Let F be the frame containing the helper function and descriptor update function. Allow instantiation of n helper function threads T_n , each owning a pending atomic operation. Assume that the atomic operations called by the helper function are guaranteed to terminate. Assume the loop invariant holds. Linearizability of the helper function operations has been demonstrated in [57].

The proof of wait-freedom uses induction on the number of active threads in the system. The base case is one thread, denoted T_1 . Thread T_1 calls `help_finish_wf` (line 1), which instantly returns since T_1 is the only active thread. T_1 proceeds to call `desc_update` to update auxiliary structure s and increment counter X . Upon entering the while loop, op_{tid} is guaranteed to return true since it comprises a single CAS that is guaranteed to succeed in the absence of thread interference. It follows that T_1 terminates in a finite number of steps.

For the inductive step assume $n+1$ active threads, denoted T_1, T_2, \dots, T_{n+1} . For an arbitrary thread T_i , it will call `help_finish_wf` on descriptor $s[X]$. Since the helper function will continuously attempt the operation defined by descriptor $s[X]$ until it is successfully completed by some arbitrary thread, in a worst case T_i will be stuck in the loop until all other active threads call `help_finish_wf` on descriptor $s[X]$. An arbitrary thread will help complete the operation posted at $s[X]$ in at most $O(n+1)$ operations, due to the worst case of helping n operations prior to helping $s[X]$. Since each of the other n threads may take $O(n+1)$ operations before helping the operation posted at $s[X]$, the total number of steps required to complete this operation is $O(n^2)$. It follows that T_i 's `help_finish_wf` call will return in $O(n^2)$ steps. Upon entering the while loop, in a worst case op_{tid} will not return true in the allowed number of attempts defined by *LIMIT* due to the interference by the other n threads. In this case, T_i will post its descriptor in $s[tid]$ on line 9, which will take at most $O(n^2)$ steps to be helped by all $n+1$ threads. It follows that T_i terminates in a finite number of steps.

The inductive step can be applied from one to the maximum of n threads. The call to `help_finish_wf` will always terminate since all active threads will eventually help each descriptor posted in auxiliary structure s . Since the atomic operations are guaranteed to terminate, wait-freedom is ensured.

□

Verification Framework for Concurrent Programs

The progress guarantee verification framework is built on top of the Verified Software Toolchain (VST) [3], a tool that includes static analyzers to check properties of C programs. The VST uses the formal language of the Coq Proof Assistant [8] to enable the expression of executable algorithms and theorems with an environment for semi-interactive development of machine-checked proofs. An overview of the progress guarantee verification framework is presented in Figure 3.27. A C source program is passed to CompCert’s *clightgen* utility [59] to produce the Abstract Syntax Tree (AST) of a C program expressed in Coq. The user must write a program specification following the methodology for descriptor-based helping mechanisms in order to verify that the program AST meets the designed progress guarantee.

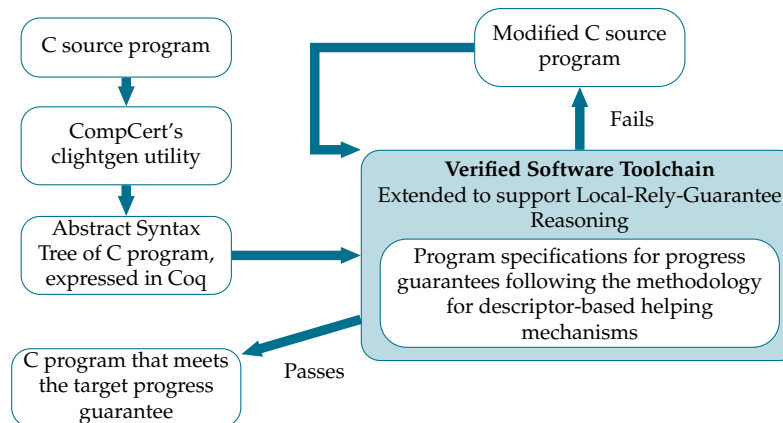


Figure 3.27: Overview of Progress Guarantee Verification Framework

The VST requires specifications for the verification of a C program to consist of an Application Programming Interface (API) specification and a function-body correctness proof for each of the functions called in the program [3]. The API specification is defined by

its precondition and postcondition, which comprises the propositional conjuncts, local conjuncts, and separation conjuncts. Propositional conjuncts are independent of the program resources and the memory. Local conjuncts are dependent on program resources, but not on memory. Separation conjuncts may be dependent on both program resources and memory.

Every state transition of a concurrent program is subject to interference from the environment. The rely condition specifies the assumption that can be made regarding the state transitions made by the environment. The guarantee condition specifies the restrictions on the state transitions made by a particular thread to the shared state. To specify the interference between the threads, the rely and guarantee conditions must be defined for each function in the C program. The rely and guarantee conditions are predicates over state transitions [29]. Let p be the precondition, let q be the postcondition, let R be the rely condition, let G be the guarantee condition, and let (p, R, G, q) be the specification of a thread. A thread satisfies its specification (p, R, G, q) if its initial state satisfies p and the environment satisfies R , and each atomic transition made by the thread satisfies G and the final state satisfies q [29].

In order to satisfy the rely and guarantee conditions, every change to the state made by a program must conform to the restrictions of the guarantee conditions, and the precondition at each state transition must be *stable* with respect to the rely condition [20, 29]. Stability is defined formally as follows [29]:

Definition 3.0.21. Stability. An assertion p is stable with respect to an action a , denoted $\text{Sta}(p, a)$, if and only if for all states s and s' , if $s \vdash p$ and $(s, s') \vdash a$, then $s' \vdash p$.

Figure 3.28 depicts the Coq mechanization of the definitions required to express the rely and guarantee conditions. The type `environ` is the Coq type for a run-time local variable

frame. The type `mpred` is the Coq type for a predicate on some part of the memory [3]. A *program assertion* is a predicate on its local variable `environ` and its memory. The Coq type for a program assertion is `environ→mpred` [3]. A *relation* is a proposition about pairs of `environ` arguments. A *transition* is a relation over program assertions. The *rely* and *guarantee* conditions, abbreviated *RG* in the definition in Figure 3.28, are each a relation over a transition of predicates on the memory.

```

Definition relation := environ→environ→Prop.
Definition transition:= (environ→mpred)→(environ→mpred) →relation.
Definition RG (trans:transition) (memval:mpred) (memval':mpred) :relation :=
  trans memval memval'.
Definition stable (R:relation) (P:environ→mpred): Prop :=
  forall x y:environ, P x && prop(R x y) |-- P y.

```

Figure 3.28: Definitions for the Rely/Guarantee Conditions Mechanized in Coq

The Coq mechanization of stability is depicted in Figure 3.28. The property `stable` accepts the arguments 1) a relation over a transition of predicates on the memory and 2) a program assertion. The VST function `prop`, used in the Coq definition for stability, accepts a proposition argument and returns an `mpred`. Stability holds if for all `environ` variables x and y , the program assertion on x and the transition of predicates on the memory over x and y entails the program assertion on y .

The proposed framework is used to prove in a semi-automatic manner that the loop invariant holds true for every iteration of the loop. The following subgoals must be proven for each loop invariant:

1. The precondition of the whole loop implies the loop invariant [3].
2. The environment satisfies the rely condition.

3. The loop body implies the loop invariant [3].
4. The loop body satisfies the guarantee condition.

The loop invariant proof is performed using inductive reasoning on the loop body. The existential variables in the loop body are instantiated with the initial values to show that the precondition is true at the beginning of the loop. The VST enables a symbolic execution of a program by application of Hoare logic inference rules through the *forward* tactic [3]. The forward tactic is modified to assert stability of the precondition for the state transition. A symbolic execution is performed to advance through the loop body until the end of the loop body is reached. The functional specifications are proved during the symbolic execution of the loop body. The existential variables in the loop body are then instantiated with a variable representing the general case to show that all values will satisfy the loop invariant.

The parallel composition inference rule for k parallel commands, shown in Equation 3.3, is derived from the parallel composition rule presented in [29]. Induction is used on the premises of Equation 3.3 and Theorem 3.0.1 or Theorem 3.0.2 is applied to prove that a descriptor-based algorithm meets the progress guarantee.

$$\begin{array}{c}
R \vee (G_2 \wedge \dots \wedge G_k); G_1; I \vdash \{p_1 * r\} C_1 \{q_1 * r_1\} \\
\vdots \\
R \vee (G_1 \wedge \dots \wedge G_{k-1}); G_k; I \vdash \{p_k * r\} C_k \{q_k * r_k\} \\
\hline
R; (G_1 \vee \dots \vee G_k); I \vdash \{p_1 * \dots * p_k * r\} C_1 \parallel \dots \parallel C_k \{q_1 * \dots * q_k * (r_1 \wedge \dots \wedge r_k)\}
\end{array} \tag{3.3}$$

Transactional Merging to Optimize Semantic Conflict Detection in Transactional Data Structures

In this section, I present *transactional merging*, a technique that optimizes the semantic conflict resolution of transactional data structures by merging conflicting operations into a single operation to reduce aborts. Transactional merging treats the transactional data structure methodology as a white box since it affects the program logic for semantic conflict resolution and recovery from aborted transactions. I first discuss the general approach for modifying a transactional data structure methodology to support transactional merging. Unlike optimization strategies for concurrent data structures [42, 41, 7] that combine operations with related semantics, transactional merging must provide the ability to recover the correct abstract state under the circumstance that a transaction with merged operations aborts. The challenges associated with transactional merging include ensuring that merging conflicting operations 1) does not jeopardize the ability of either transaction to commit to memory, and 2) does not violate correctness of the transactional data structure. I then demonstrate how to address these challenges by showcasing the transactional merging technique applied to Lock-Free Transactional Transformation (LFTT) [107]. Correctness statements are provided which prove that transactional merging is strictly serializable when applied to LFTT.

I apply transactional merging to a lock-free transactional linked list [107], a lock-free transactional red-black tree [92], and a lock-free transactional dictionary [105]. The experimental evaluation demonstrates that transactional merging achieves an average speedup of up to 162% over LFTT for the linked list, 229% over LFTT for the red-black tree, and 123% over the Masstree [67] indexing used in the Silo database [98].

The contributions of this work include:

1. I present transactional merging, a technique that enables conflicting transactions to merge operations such that the conflict is eliminated. The benefit is that the throughput of committed transactions is improved by reducing the total number of aborts.
2. I generalize the concept of transactional merging such that it can be applied to any transactional data structure methodology.
3. I showcase the application of transactional merging to LFTT to demonstrate how to 1) merge operations without causing aborts due to side-effects of the merge and 2) retain correctness of the transactional data structure. I achieve this by developing a strategy that enables a correct recovery of the abstract state under the circumstances that a transaction attempting to merge operations aborts. I provide correctness statements proving that transactional merging applied to LFTT is strictly serializable.
4. I integrate a lock-free transactional dictionary into the Silo database [98] to demonstrate a practical application of transactional merging.

Transactional Merging

Transactional merging is inspired by techniques utilized by elimination stacks [42, 7] and flat combining [41] because it improves throughput by combining operations. However, transaction merging distinguishes itself from the previous approaches by presenting a strategy to optimize performance of transactional data structures. The main idea behind transactional merging is that transactions that perform operations with a high-level semantic conflict can collaborate rather than force one transaction to abort. For example, consider two transactions on a map, depicted in Figure 3.29. This example assumes a

transactional synchronization scheme in which a thread helps to complete a pending transaction with a conflicting operation. Thread 1's first transaction performs `INSERT(2,B)` and Thread 2's transaction performs `INSERT(2,X)`, resulting in a semantic conflict on key 2. Rather than forcing Thread 2 to abort its transaction, Thread 2 helps to complete Thread 1's transaction and then merges the conflicting operations by converting its `INSERT(2, X)` to a `PUT(2, X)`. Similarly, Thread 2's transaction performs `DELETE(4)`, and Thread 1's second transaction performs `DELETE(4)`, resulting in a semantic conflict on key 4. In this case, Thread 1 helps to complete Thread 2's transaction and then merges the conflicting operations by eliminating its `DELETE(4)` operation, since it has already been performed by Thread 2. If the actions taken in the previous scenario to eliminate semantic conflicts are more relaxed than the transactional system is able to tolerate, the conditions for which merging operations is allowed to take place can be defined by the designer to be more restrictive. Additionally, the designer could further relax the conditions for merged operations such that semantic conflicts between operations of different types (for example, `INSERT` and `PUT`) are permissible to achieve greater performance gains.

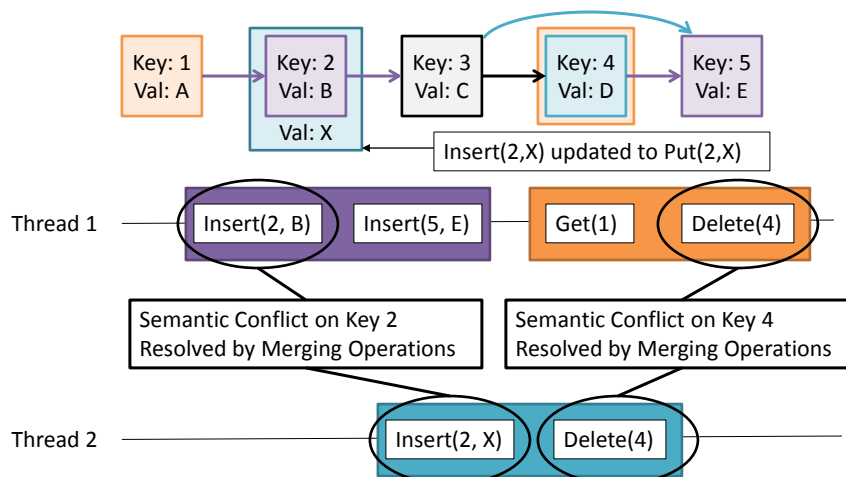


Figure 3.29: Transactional Merging for a Map

Generalizing Transactional Merging

Transactional merging can be applied to any transactional data structure methodology that incorporates a high-level semantic conflict detection. Such transaction data structure methodologies include transactional boosting [44], LFTT [107], transactional data structure libraries [94], and transactional data structures constructed using STM. The generalized strategy requires the transactional data structure to be treated as a white box since transactional merging must be integrated directly into the program logic for semantic conflict resolution and recovery scheme invoked after an abort. There are two aspects of transactional data structure methodologies that must be modified to support transactional merging. The first aspect is the operation precondition as defined by the data structure semantics. The second aspect is the recovery scheme for aborted transactions. There are several vulnerabilities that a transactional data structure methodology is susceptible to when modified to support transactional merging. These vulnerabilities include 1) aborts caused by side-effects of merged operations and 2) violations of the transactional correctness condition. The following subsections discuss the modifications required to support transactional merging in conjunction with guidance regarding how to prevent the previously mentioned vulnerabilities.

Operation Precondition

Data structure operations use a precondition as defined by the data structure semantics in the determination of a return value for the operation. For example, the precondition for an $\text{INSERT}(x)$ operation is that x does not exist in the set. Traditionally, $\text{INSERT}(x)$ returns false if the precondition is not satisfied. Data structure operations that terminate unsuccessfully can lead to aborts depending on the transactional data structure methodology

or a reduction in throughput. Transactional merging takes a different approach when handling the operation precondition. If the precondition is not satisfied for the operation, it is possible that the goals of the operation have already been performed by another thread. In this case the pending operation can merge with the completed operation and terminate successfully, which reduces aborts and increases throughput.

Formal definitions are now provided to describe transactions with merged operations.

Definition 3.0.22. Let T_1 and T_2 be transactions that contain at least one semantic conflict. Let the first semantic conflict be on element e_i . If transaction T_1 completes its operation on element e_i before transaction T_2 in real time, then T_1 is the *prefix transaction*.

Definition 3.0.23. Let T_1 and T_2 be transactions that contain at least one semantic conflict. Let the first semantic conflict be on element e_i . If transaction T_2 detects a semantic conflict with T_1 because T_1 completed its operation on element e_i before transaction T_2 in real time, then T_2 is the *suffix transaction*.

To ensure that aborts caused by side-effects of the merged operations do not occur, it is essential that the prefix transaction successfully commits to memory. Otherwise, the suffix transaction that intends to merge operations with the prefix transaction would have to abort or restart if the effects of the operation being merged with were rolled back. The general strategy for preventing aborts due to side-effects of merged operations is to utilize the transactional synchronization protocol of the transactional data structure methodology (i.e. acquisition of locks [94, 44] or helping scheme [107]) to commit a transaction to memory prior to merging with its operations. The transactional synchronization protocol enables the suffix transaction performing the merge to take ownership of the operation after the prefix transaction commits and alert other threads that this operation is still active.

Recovery Scheme

Transactional data structure methodologies provide the ability to recover the correct abstract state given that a transaction aborts. The potential correctness violation that can occur due to merged operations resides in the recovery scheme. If a suffix transaction performing a merge aborts and the rollback appears as though the merged operation was never performed, this would result in an incorrect abstract state since the operation being merged with was successfully committed to memory prior to the merge. To prevent an incorrect recovery of the abstract state after a rollback, a suffix transaction that intends to perform a merge must set a *merge* flag for the operation after taking ownership of the operation. The *merge* flag indicates that the operation must be restored to the state it was in prior to the merge given that an abort occurs. A *descriptor* is a shared object used to announce information regarding an operation to be performed. The operation state information prior to the merge must be stored in a descriptor object for correct recovery of the abstract state given that the transaction performing the merge aborts.

LFTT With Transactional Merging

Algorithm 9 provides the constants and data type definitions adapted from LFTT for the map abstract data type. The `OpType` on line 9.5 lists the map operations, including `INSERT` (inserts a new key-value pair into the map), `DELETE` (deletes a key and its associated value from the map), `PUT` (updates an existing key in the map with a new value, or inserts a new key-value pair if the key doesn't exist), and `GET` (retrieves the value associated with an existing key in the map). The `Operation` type is updated from the LFTT type definitions to include a value *val* (line 9.13) and a reference to a map object (line 9.14). The `Desc` type, line 9.15, contains fields for the transaction size, the transaction status (Active, Committed,

or Aborted), and an array of operations to be performed in the transaction. The `NodeInfo` type, line 9.19, contains fields for the `Desc` and the current operation index *opid*. A `Desc` reference and operation index are embedded in the `NodeInfo` to enable a transaction to help complete the pending transaction associated with the node to be operated on starting at the current operation index. The `NodeInfo` type is updated from the `LFTT` type definitions to include a value *val* (line 9.22), the value *oldval* held by the previously committed transaction that updated the node (line 9.23), and a boolean flag *merge* to indicate if the transaction is attempting to merge operations (line 9.24). The purpose of storing the value held by the previously committed transaction that updated the node is to recover the correct value given the circumstances that the transaction performing the merge aborts. The `Node` type on line 9.25 contains a `NodeInfo` reference *info* and a unique integer *key*.

ALGORITHM 9: Type Definitions

<pre> 1 enum TxStatus 2 Active; 3 Committed; 4 Aborted; 5 enum OpType 6 Insert; 7 Delete; 8 Put; 9 Get; 10 struct Operation 11 OpType type; 12 int key; 13 int val; 14 Map* map; </pre>	<pre> 15 struct Desc 16 int size; 17 TxStatus status; 18 Operation ops[]; 19 struct NodeInfo 20 Desc* desc; 21 int opid; 22 int val; 23 int oldval; 24 bool merge; 25 struct Node 26 NodeInfo* info; 27 int key; 28 ...; </pre>
---	---

Semantic Conflict Resolution Policy

The vulnerabilities susceptible to transactional merging in LFTT are addressed by retaining the cooperative transaction execution scheme and modifying the logical rollback presented

by Zhang et al. [107] to eliminate semantic conflicts without causing aborts due to side-effects of merged operations and preserve transactional correctness. The suffix transaction that intends to merge operations with the prefix transaction must first help complete the prefix transaction. This action ensures that the prefix transaction is committed to memory and will not be affected by the merged operations. If the merge is unsuccessful, the suffix transaction will abort and undo its effects through the logical rollback adapted for transactional merging to preserve strict serializability.

ALGORITHM 10: Conflict Resolution Policy

```

1 Function IsMergePossible(NodeInfo* oldinfo, NodeInfo* currinfo)
2   OpType nodeOp  $\leftarrow$  oldinfo.desc.ops[oldinfo.opid].type;
3   OpType currOp  $\leftarrow$  currinfo.desc.ops[currinfo.opid].type;
4   TxStatus nodeStatus  $\leftarrow$  oldinfo.desc.status;
5   return
    ((nodeOp = currOp and (currOp = Insert or currOp = Delete)) and nodeStatus = Committed);

```

When a suffix transaction detects a semantic conflict and finishes helping the prefix transaction to complete its operations, it will attempt to merge conflicting operations based on the semantic conflict resolution policy. Function `IsMergePossible` detailed in Algorithm 10 accepts as arguments the `NodeInfo` of the node to be operated on and the `NodeInfo` for the current operation. If the node's operation is equivalent to the current operation, the current operation is either an `INSERT` or `DELETE`, and the transaction associated with the node has committed, then the transaction performing the current operation can merge its operations with the previously committed transaction. For sets, the merge will remove redundant `INSERT` and `DELETE` operations. For maps, the merge will remove redundant `DELETE` operations and update the conflicting `INSERT` operation to a `PUT` operation. If converting a conflicting `INSERT` operation to a `PUT` operation is too relaxed for the transactional system, the designer can enforce a stricter semantic conflict resolution policy such that only `INSERT` operations with identical key-value pairs are merged by updating line 10.5 to

include the following condition: **and** (*oldinfo.val* = *currinfo.val*).

ALGORITHM 11: Logical Status

```

1 Function IsNodePresent(Node* node, int key)
2   return node.key = key;
3 Function IsKeyPresent(NodeInfo* info, Desc* desc)
4   OpType op  $\leftarrow$  info.desc.ops[info.opid].type;
5   TxStatus status  $\leftarrow$  info.desc.status;
6   switch status do
7     case Active do
8       if info.desc = desc then
9         return op = Get or op = Insert or op = Put;
10      return op = Get or op = Delete or (op = Put and info.oldval  $\neq$  NULL);
11     case Committed do
12       return op = Get or op = Insert or op = Put;
13     case Aborted do
14       return op = Get or (op = Put and info.oldval  $\neq$  NULL) or
15       (op = Delete and info.merge = false) or (op = Insert and info.merge = true);

```

The logical status for the map abstract data type is presented in Algorithm 11. The `IsNodePresent` function on line 11.1 returns true if the key of the node argument is equivalent to the specified key. The `IsKeyPresent` function on line 11.3 performs the logical interpretation of the existence of a node based on its operation type, line 11.4, and transaction status, line 11.5. Given that the transaction is *Active* and the node is accessed by operations in the same transaction, then `IsKeyPresent` returns true if the operation type is `GET`, `INSERT`, or `PUT`. Otherwise, if the transaction is *Active* and the node is accessed by operations in a different transaction, then `IsKeyPresent` returns true if the operation type is `GET`, `DELETE`, or `PUT` (if *info.oldval* \neq NULL). If the field *info.oldval* is not NULL, then the `PUT` operation only updates the value since the key must already exist in the map. If the transaction is *Committed*, then `IsKeyPresent` returns true if the operation type is `GET`, `PUT`, or `INSERT`. If the transaction is *Aborted*, then `IsKeyPresent` returns true if the operation type is `GET`, `PUT` (if *info.oldval* \neq NULL), `DELETE` (if *info.merge* = *false*), or `INSERT` (if *info.merge* = *true*). To explain the reasoning behind the `INSERT` and `DELETE` cases,

assume that a transaction's *info.merge* field is set to true. If a transaction is attempting to merge with a conflicting DELETE operation and aborts, then the node is still deleted in the recovered abstract state. Therefore, the `IsKeyPresent` function should return true for a DELETE if *info.merge* = *false*. If a transaction is attempting to merge with a conflicting INSERT operation and aborts, then the node is still inserted in the recovered abstract state. Therefore, the `IsKeyPresent` function should return true for an INSERT if *info.merge* = *true*.

ALGORITHM 12: Update NodeInfo

```

1 Function UpdateInfo(Node* node, NodeInfo* info, bool wantkey)
2   NodeInfo* oldinfo ← node.info;
3   if IsMarked(oldinfo) then
4     Do_DELETE(node);
5     return retry;
6   if oldinfo.desc ≠ info.desc then
7     EXECUTEOps(oldinfo.desc, oldinfo.opid + 1);
8   else if oldinfo.opid ≥ info.opid then
9     return success;
10  if oldinfo.desc.status = Committed then
11    info.oldval ← oldinfo.val;
12  else if oldinfo.desc.status = Aborted then
13    info.oldval ← oldinfo.oldval;
14  OpType currOp ← info.desc.ops[info.opid].type;
15  if currOp = Get or currOp = Delete then
16    info.val = info.oldval;
17  bool haskey ← IsKeyPresent(oldinfo, info.desc);
18  if (!haskey and wantkey) or (haskey and !wantkey) or (currOp ≠ Put) then
19    if (!IsMergePossible(oldinfo, info)) then
20      return fail;
21    node.info.merge ← true;
22  if info.desc.status ≠ Active then
23    return fail;
24  if CAS(&node.info, oldinfo, info) then
25    return success;
26  return retry;

```

UPDATEINFO, presented in Algorithm 12, helps to complete the pending transaction associated with the node of interest by invoking EXECUTEOps on line 12.7. The EXECUTEOps function finishes a pending transaction associated with a node by first checking for a

cyclic dependency, which would lead to infinite recursive helping calls, and aborting if a dependency exists. If a cyclic dependency does not exist, then EXECUTEOPS proceeds by starting at the current operation index and executing all operations in the transaction until the transaction descriptor status is updated to Committed or Aborted by a COMPARE-ANDSWAP (CAS) operation. CAS is an atomic synchronization primitive that accepts as arguments a memory location, expected value, and update value. If the data referenced by the memory location is equivalent to the expected value, then the data referenced by the memory location is changed to the update value and true is returned; otherwise, no change is made and false is returned.

To enable the recovery of the correct value of a node if a transaction attempting to merge conflicting operations fails, the *info.oldval* must be set to the node value associated with the previously committed transaction. If *oldinfo.desc.status = Committed*, then *info.oldval* is set to *oldinfo.val* on line 12.11. If *oldinfo.desc.status = Aborted*, then *info.oldval* is set to *oldinfo.oldval* on line 12.13. Since INSERT and PUT have a value input parameter for the key-value pair, only GET and DELETE need to recover the correct value associated with the existing node, performed on line 12.16.

ISKEYPRESENT is invoked on line 12.17 to determine if the node's key logically exists, stored in the boolean *haskey*. Several scenarios that may force UPDATEINFO to fail based on the logical interpretation. The boolean *wantkey* is true if the operation requires the key to logically exist in the list, and is false otherwise. PUT can perform its operation regardless of the logical status of the key. If *haskey* is false and the boolean argument *wantkey* is true or if *haskey* is true and the boolean argument *wantkey* is false, or *currOp* \neq *Put*, then UPDATEINFO is forced to fail unless it can merge its operation associated with the node of interest. If ISMERGEPOSSIBLE returns true on line 12.19, then *n.info.merge* is set to true on line 12.21. If ISMERGEPOSSIBLE returns false, then UPDATEINFO is forced to fail.

The `NodeInfo` is updated on line 12.24 by invoking CAS on *n.info*. The correctness of transactional merging is provided in Appendix C.

Transformed Map Functions

The LFTT template for the transformed `INSERT` function is presented in Algorithm 13. The `Do LocatePred` function is a member of the base lock-free map that locates the node with the key of interest. If `IsNodePresent` on line 13.6 returns true, then `UpdateInfo` is called on line 13.7. Otherwise, a new node is inserted in the map by calling the `Do Insert` function from the base lock-free map. If `Do Insert` fails, then the loop is continued and `IsNodePresent` returns true and `UpdateInfo` is invoked since some other thread inserted the element into the map. The transformed `PUT`, `DELETE`, and `GET` functions are identical to the transformed `INSERT` function except that for `PUT` the boolean argument passed to `UpdateInfo` on line 13.7 is true, and for `DELETE` and `GET` the boolean argument passed to `UpdateInfo` on line 13.7 is true and the *ret* value is set to *fail* if `IsNodePresent` on line 13.6 returns false.

ALGORITHM 13: Template for Transformed Insert Function

```

1 Function Insert(int key, int val, Desc* desc, int opid)
2   NodeInfo* info ← new NodeInfo;
3   info.desc ← desc, info.opid ← opid, info.val ← val, info.oldval ← NULL, info.merge ← false;
4   while true do
5     Node* curr ← Do LocatePred(key);
6     if IsNodePresent(curr, key) then
7       ret ← UpdateInfo(curr, info, false);
8       if ret = fail then
9         return false;
10    Node* node ← new Node;
11    node.key ← key, node.info ← info;
12    ret ← Do Insert(node);
13    if ret = success then
14      return true;

```

CHAPTER 4: EXPERIMENTAL EVALUATION

CCSpec

CCSpec is evaluated by checking the correctness of an assortment of hand-crafted concurrent algorithms. The concurrent depth-first search algorithm uses the non-blocking stack by Treiber et al. [97] to store the next items to be searched in a graph. The concurrent breadth-first search algorithm uses the non-blocking k-FIFO queue by Kirsch et al. [56] to store the next items to be searched in a graph. The concurrent Dijkstra’s shortest path algorithm uses the non-blocking priority queue by Zhang et al. [106] to store nodes prioritized by their distances. The concurrent adjacency list uses the non-blocking linked list by Harris et al. [40] to store vertices and adjacent edges of a graph.

Results

CCSpec and the experiments are publicly released at <http://ucf-cs.github.io/CCSpec/> as an AREA 67 lab project under a BSD open-source license. The tests are conducted on a 64-core NUMA system (4 AMD opteron 6272 CPUs with 16 cores per chip @2.1 GHz). The concurrent histories are collected from CDSChecker [75] and are equally distributed among the 64 cores to check correctness using CCSpec.

The results for the data structure layer are shown in Table 4.1. As expected, the Treiber stack and Harris list meet linearizability, the strongest correctness condition. The k-FIFO queue fails linearizability, sequential consistency, and quiescent consistency, but passes quasi-linearizability. The priority queue fails linearizability and sequential consistency,

but passes quiescent consistency and quasi-linearizability. Although the priority queue is designed for quiescent consistency, it could still meet quasi-linearizability if a period of quiescence occurs within a distance of k between method calls.

Table 4.1: CCSpec Results for the Data Structure Layer

Non-Blocking Data Structure	Linearizability	Sequential Consistency	Quiescent Consistency	Quasi-Linearizability
Treiber Stack [97] (Depth-First Search)	Pass	Pass	Pass	Pass
k-FIFO Queue [56] (Breadth-First Search)	Fail	Fail	Fail	Pass
Priority Queue [106] (Dijkstra's Shortest Path)	Fail	Fail	Pass	Pass
Harris List [40] (Adjacency List)	Pass	Pass	Pass	Pass

The results for the abstract function layer are shown in Table 4.2. The basis of correctness evaluation is the final observed abstract state of the shared resources of the concurrent algorithms. This approach for checking algorithm correctness is equivalent to verifying the postcondition for a function. For the depth-first search and breadth-first search algorithms, the shared resource is the *marked* array indicating which nodes have been visited. The shared resources of Dijkstra's shortest path is the *dist* array storing the distance between a node and the source node, and the *prev* array storing the predecessor of a node in the shortest path. The shared resource of the adjacency list is the structure of the adjacency list itself.

Table 4.2: CCSpec Results for the Abstract Function Layer

Concurrent Algorithm	Algorithm Correctness
Depth-First Search	Pass
Breadth-First Search	Pass
Dijkstra's Shortest Path	Pass
Adjacency List	Pass

A critical observation is that the use of a k-FIFO queue for the breadth-first search and the use of a quiescently consistent priority queue for Dijkstra's shortest path did not

affect correctness at the abstract function layer. For the breadth-first search, a thread will enqueue all adjacent edges of a dequeued node that have not been visited into a k-FIFO queue. The order in which these nodes are dequeued does not matter since the overall goal of the algorithm is to explore all nodes. Since the final *marked* array indicates that all nodes have been visited, breadth-first search is correct at the abstract function layer. For Dijkstra's shortest path, a thread will update the distance of all adjacent edges after removing a node from the priority queue. If a node is removed that does not contain the minimum distance due to the quiescently consistent nature of the priority queue, the *dist* and *prev* arrays will be correctly updated once the node with the minimum distance is removed from the queue. Since the abstract state of the final *dist* and *prev* arrays are correct, Dijkstra's shortest path is correct at the abstract function layer. Such results imply that potential performance gains can be achieved in a concurrent algorithm by utilizing data structures that are optimized for a relaxed correctness condition. Special design considerations are required for sequential consistency because unlike linearizability, quiescent consistency, and quasi-linearizability, a composition of sequentially consistent objects provides no guarantee that the execution as a whole will be sequentially consistent.

Table 4.3: Concurrent Histories Generated by CDSChecker

	Depth-First Search	Breadth-First Search	Dijkstra's Shortest Path	Adjacency List
Number of Concurrent Histories	143650	1081148	370800	4408787

The number of concurrent histories generated by CDSChecker for the unit tests are shown in Table 4.3. The execution times for the unit tests are shown in Table 4.4. The column abbreviated CDS is the time (in hours) for CDSChecker to generate the concurrent histories of the unit test. The column abbreviated CCSpec is the time (in hours) for CCSpec to run the correctness checking algorithm to determine if the unit test meets the correctness

condition. The execution time for CDSChecker is displayed in Table 4.4 to provide the reader with an understanding of how long the entire process of checking correctness takes, which is the sum of the model checking time and correctness checking time. The depth-first search launches three threads, where each thread operates on a graph with three vertices and two edges. Breadth-first search and Dijkstra's shortest path launch four threads, where each thread operates on a graph with three vertices and two edges. The adjacency list algorithm launches three threads, where one thread perform six operations, one thread performs five operations, and one thread performs four operations. The unit test for depth-first search is written to execute fewer threads than breadth-first search because the large amount of contention on the stack top utilized by the depth-first search results in a large amount of method call overlap. Since methods called by different threads can be ordered in any arbitrary way for sequential consistency and methods that do not encounter a period of quiescence can be ordered in any arbitrary way for quiescent consistency, the number of possible legal sequential histories grows at the rate of a factorial function for every additional thread. Since linearizability has the lowest number of valid sequential histories due to the real-time ordering constraint, linearizability takes the least amount of time to check. For the breadth-first search and Dijkstra's shortest path, quiescent consistency takes approximately the same amount of time to check as linearizability. This occurs if each method is separated by a period of quiescence, which places a real-time ordering constraint on each method, making quiescent consistency equivalent to linearizability. Sequential consistency takes the most amount of time to check because the methods called in the concurrent histories are constrained only by program order. Quasi-linearizability enforces real-time ordering on method calls separated by a distance of two, which takes more time to check than linearizability.

Table 4.4: CCSpec Execution Time Results (in hours)

Correctness Condition	Depth-First Search (hours)		Breadth-First Search (hours)		Dijkstra's Shortest Path (hours)		Adjacency List (hours)	
	CC-Spec	CDS.	CC-Spec	CDS.	CC-Spec	CDS.	CC-Spec	CDS.
Linearizability	0.25	0.06	5.30	0.61	1.16	0.40	10.47	0.79
Sequential Consistency	6.75	0.06	129.78	0.61	30.04	0.40	14.64	0.79
Quiescent Consistency	76.85	0.06	10.29	0.61	1.96	0.40	17.22	0.79
Quasi-Linearizability (k=2)	4.24	0.06	36.43	0.61	9.09	0.40	11.38	0.79

The adjacency list is an exception to the previously discussed execution times. The discrepancy in the execution times in comparison to the other algorithms is due to the optimizations to the recursive topological sort such that commutative operations are pruned from the search space. The adjacency list benefits from this optimization because it comprises a list of lists. Since each list is a separate container, methods commute if they operate on different lists. Additionally, method calls on the same list commute if they operate on different elements. With the redundant legal sequential histories pruned from the search space, all correctness conditions take approximately the same amount of time to check for the adjacency list.

Counterexamples

Counterexamples are produced by CCSpec under the linearizability correctness condition for the data structure layer and abstract function layer and the results obtained are discussed.

k-FIFO Queue

Figure 4.1 shows a subset of the counterexamples produced for the *k*-FIFO queue under the linearizability correctness condition. The parameter *k* is set to one in the left counterexample of Figure 4.1. In this scenario, the sequential specification for the three Dequeue operations is `A.Deq(): 1`, `A.Deq(): 2`, and `A.Deq(): 3`. However, the program output is `A.Deq(): 2`, `A.Deq(): 1`, and `A.Deq(): 3`. In this case, the Enqueue operations by the main thread are out of order by a distance of one. The parameter *k* is set to two in the right counterexample of Figure 4.1. The program output for this scenario is `A.Deq(): 3`, `A.Deq(): 2`, and `A.Deq(): 1`. In this case, the Enqueue operations by the main thread are out of order by a distance of two.

The bounded *k*-FIFO queue maintains an array of *k*-segments each with *k* slots in order to reduce contention on the head and tail pointers. At most *k* enqueue and *k* dequeue operations may be performed simultaneously, allowing for elements to be dequeued out of order by at most *k* dequeue operations. Although the counterexamples demonstrate that the concurrent histories are not linearizable, they do meet the quasi-linearizability correctness condition for *k* set to one and two, respectively.

1	Sequential Output	Program Output	1	Sequential Output	Program Output
2	Main: A.Enq(1); Void	Void	2	Main: A.Enq(1); Void	Void
3	Main: A.Enq(2); Void	Void	3	Main: A.Enq(2); Void	Void
4	Main: A.Enq(3); Void	Void	4	Main: A.Enq(3); Void	Void
5	Thread1: A.Deq(); 1	2	5	Thread1: A.Deq(); 1	3
6	Thread2: A.Deq(); 2	1	6	Thread2: A.Deq(); 2	2
7	Thread2: A.Deq(); 3	3	7	Thread2: A.Deq(); 3	1

Figure 4.1: *k*-FIFO Linearizability Counterexamples

Priority Queue

Figure 4.2 shows a subset of the counterexamples produced for the priority queue under linearizability at the data structure layer. The left counterexample of Figure 4.2 demonstrates a scenario where thread 2 invokes `DeleteMin` and calls `Insert(3)` followed by an `Insert(4)`, then thread 1 calls `DeleteMin`. Since priority is given to elements with the lowest key, 3 should be the first element removed by a call to `DeleteMin`. However, `DeleteMin` by thread 1 removes 4. The right counterexample of Figure 4.2 demonstrates a scenario where thread 2 invokes `DeleteMin` and calls `Insert(3)`, thread 1 calls `DeleteMin`, and thread 2 calls `Insert(4)`. The `DeleteMin` by thread 1 removes `NULL` instead of the expected value 3.

The priority queue inserts an element by mapping a scalar key to a D-dimensional coordinate vector, where a target position is located using the vector as the coordinates. The priority queue deletes the minimum element by removing the head node and setting the element with the next smallest key to the new head node. Since the complexity of the search for the element with the next smallest key grows exponentially with more nodes, a deletion stack is used to provide hints regarding the location of the next smallest node. As proven in [106] and verified with CCSpec, the `Insert` and `DeleteMin` operations respect real-time ordering when separated by a period of quiescence. This is due to the thread executing `DeleteMin` reading the deletion stack prior to the update by the thread calling `Insert`.

1	Sequential	Output	Program	Output	1	Sequential	Output	Program	Output
2	Main: A.Ins(0);	Void	Void		2	Main: A.Ins(0);	Void	Void	
3	Thread2: A.DelMin();	0	0		3	Thread2: A.DelMin();	0	0	
4	Thread2: A.Ins(3);	Void	Void		4	Thread2: A.Ins(3);	Void	Void	
5	Thread2: A.Ins(4);	Void	Void		5	Thread1: A.DelMin();	3	NULL	
6	Thread1: A.DelMin();	3	4		6	Thread2: A.Ins(4);	Void	Void	

Figure 4.2: Priority Queue Linearizability Counterexamples

Adjacency List

Figure 4.3 shows the unit test and Figure 4.4 shows a subset of the counterexamples produced for the incorrect usage of the adjacency list under linearizability at the abstract function layer. An abstract function is specified for thread 1's body, and another abstract function is specified for thread 2's body. For the `Add_Ver` and `Edge_List_Empty` methods, the argument is the vertex id. For the `Add_Edge` and `Contains_Edge` methods, the first argument is the vertex id and the second argument is the edge node id. The top counterexample of Figure 4.4 shows a scenario in which thread 1's body is executed before thread 2's body. If thread 1 inserts edge node 500 into vertex 100's edge list, then the if-statement by thread 2's body will be false. In this case, the `Contains_Edge(100, 500)` by thread 3 should return true, and the `Contains_Edge(100, 600)` by thread 3 should return false. However, the program output shows that both `Contains_Edge(100, 500)` and `Contains_Edge(100, 600)` return true. The bottom counterexample of Figure 4.4 shows a scenario in which thread 2's body is executed before thread 1's body. If thread 2 inserts edge node 600 into vertex 100's edge list, then the if-statement by thread 1's body will be false. In this case, the `Contains_Edge(100, 500)` by thread 3 should return false, and the `Contains_Edge(100, 600)` by thread 3 should return true. However, the program output shows that both `Contains_Edge(100, 500)` and `Contains_Edge(100, 600)` return true.

<pre> 1 Main: 2 A.Add_Vertex(100); 3 4 Thread1: 5 if(A.Edge_List_Empty(100)) 6 A.Add_Edge(100, 500); </pre>	<pre> 1 Thread2: 2 if(A.Edge_List_Empty(100)) 3 A.Add_Edge(100, 600); 4 5 Thread3: 6 A.Contains_Edge(100, 500); 7 A.Contains_Edge(100, 600); </pre>
---	---

Figure 4.3: Adjacency List Unit Test

	Sequential Output	Program Output
1		
2 Main: A.Add_Vertex(100);	True	True
3 Thread1: Thread1.body();	Void	Void
4 Thread2: Thread2.body();	Void	Void
5 Thread3: A.Contains_Edge(100, 500); True		True
6 Thread3: A.Contains_Edge(100, 600); False		True
1	Sequential Output	Program Output
2 Main: A.Add_Vertex(100);	True	True
3 Thread2: Thread2.body();	Void	Void
4 Thread1: Thread1.body();	Void	Void
5 Thread3: A.Contains_Edge(100, 500); False		True
6 Thread3: A.Contains_Edge(100, 600); True		True

Figure 4.4: Adjacency List Linearizability Counterexamples

TXC-ADT

TxC-ADT is evaluated by checking the correctness of Lock-Free Transactional Transformation (LFTT) [107] and Transactional Data Structure Libraries (TDSL) [94]. The tests are conducted on a 64-core NUMA system (4 AMD opteron 6272 CPUs with 16 cores per chip @2.1 GHz). The unit tests for the LFTT skiplist and TDSL queue comprise three threads such that one thread issues a transaction with one operation and a transaction with three

operations, one thread issues a transaction with two operations, and one thread issues a transaction with one operation. The unit tests for the LFTT linked list and TDSL skiplist comprise three threads such that one thread issues a transaction with one operation and a transaction with two operations, one thread issues 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()`. The concurrent histories are collected from CDSChecker and equally distributed 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 4.5. 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 [107] 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.

Table 4.5: TxC-ADT Results for Transactional Data Structures

Transactional Data Structure	Correctness Condition			
	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

The number of concurrent histories generated by CDSChecker for the unit tests are shown in Table 4.6. The execution times for the unit tests are shown in Table 4.7. The column abbreviated CDS is the time (in hours) for CDSChecker to generate the concurrent histories of the transactional data structure unit test. The column abbreviated TxC is the time (in hours) for TxC-ADT to analyze the concurrent histories and determine if the unit test meets the specified transactional correctness condition. The execution time for CDSChecker is displayed in Table 4.7 to provide the reader with an understanding of how long the entire process of checking correctness takes, which is the sum of the model checking time and correctness checking time. Opacity generally takes the largest amount of time to check because the effects of all transactions (active, committed, and aborted) must be evaluated for correctness. The 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. Serializability takes more time to check than strict serializability because serializability has more possible legal sequential histories than strict serializability due to the real-time ordering constraint for committed transactions required by strict serializability. 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.

Table 4.6: Concurrent Histories Generated by CDSChecker

	LFTT Linked List	LFTT Skiplist	TDSL Queue	TDSL Skiplist
Number of Concurrent Histories	5815894	4421091	2055894	3414414

Table 4.7: TxC-ADT Execution Time Results (in hours)

Correctness Condition	LFTT Linked List (hours)		LFTT Skiplist (hours)		TDSL Queue (hours)		TDSL Skiplist (hours)	
	TxC.	CDS.	TxC.	CDS.	TxC.	CDS.	TxC.	CDS.
Causal Consistency	4.66	5.58	3.08	11.50	1.65	1.71	2.21	4.14
Serializability	3.67	5.58	2.93	11.50	2.66	1.71	2.19	4.14
Strict Serializability	3.63	5.58	2.81	11.50	1.67	1.71	2.04	4.14
Opacity	3.77	5.58	3.09	11.50	2.53	1.71	2.36	4.14

In order to demonstrate the ability of TxC-ADT to produce counterexamples, design flaws are injected 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.

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.

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

Figure 4.5: LFTT Linked List Unit Test

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

Figure 4.6: LFTT Linked List Concurrent Histories

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

Figure 4.7: LFTT Linked List Opacity Counterexamples (with design flaws injected)

A design flaw is injected 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 that 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 Fig. 4.5, the concurrent histories are shown in Fig. 4.6, and the resulting counterexamples are shown in Fig. 4.7. 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 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.

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 valida-

tion 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.

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

Figure 4.8: TDSL Queue Unit Test

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

Figure 4.9: TDSL Queue Concurrent Histories

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

Figure 4.10: TDSL Queue Opacity Counterexamples (with design flaws injected)

A design flaw is injected 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 Fig. 4.8, the concurrent histories are shown in Fig. 4.9, and the resulting counterexamples are shown in Fig. 4.10. 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 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.

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 [75] uses dynamic partial order reduction [31] 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 .

Progress Verification

The progress verification methodology is applied to a lock-free transactional list [107] and a wait-free queue [57]. Both data structures require descriptor-based mechanisms to achieve the designed progress guarantee.

Lock-Free Transactional List

Lock-free transactional transformation implements the lock-free algorithm of Figure 3.24 because *FinishPendingTxn* will check the help stack prior to entering a loop to finish the transaction, allowing for the application of Theorem 3.0.1 to prove lock-freedom. The shared resource for the lock-free transactional list is the head of the list. The specification for *FinishPendingTxn* is shown in Figure 4.11. *FinishPendingTxn* will only be called if a conflict is detected with an existing node in the list. Therefore, the head must point to some node *next* since the list is not empty, as indicated in the shared state of the precondition. Each thread maintains a thread local help stack, so it is hidden from the other functions of the lock-free transactional list. If the help stack contains the descriptor *desc*, then a cyclic dependency is detected and the transaction is aborted. If the help stack does not contain the descriptor *desc*, then it will push the descriptor onto the help stack and execute all operations in the transaction descriptor, indicated by the head pointing to an updated list *next'* in the shared state of the postcondition. Once all operations are executed, the descriptor is popped from the help stack. Therefore in the postcondition, the top of the help stack should not point to descriptor *desc*.

```

I : = m_head : list Node, _m_head ↦ m_head
PRE: LOCAL: ∃ desc : Desc, ∃ opid : int, _desc ↦ desc ∧ _opid ↦ opid
      SEP: m_head : list Node, _m_head ↦ m_head * ∃ next : Node, (m_head ↦ _, next, _),
      ∃ helpStack : HelpStack, _helpStack ↦ helpStack
POST: SEP: m_head : list Node, _m_head ↦ m_head*
if (!helpStack.Contains(desc)) then
  ∃ next' : Node, (m_head ↦ _, next', _) ∧ helpStack.Top()! = desc
else desc.status ↦ ABORTED
R : = FinishPendingTxn.G ∧ Insert.G ∧ Delete.G ∧ Find.G
G : = (helpStack.Contains(desc) × desc.status ↦ ABORTED) ∨ (!helpStack.Contains(desc) × helpStack.Push(desc)
  ∧ (∀ opid' : int, opid' ≥ opid ∧ opid' < desc.size ∧ ∃ op : Operator, op ↦ desc.ops[opid] ∧
  ((op.type = INSERT ∧ Insert.G) ∨ (op.type = DELETE ∧ Delete.G) ∨ (op.type = FIND ∧ Find.G))) × helpStack.pop())
LOOP INVARIANT: ∃ j : int,
PROP: opid ≤ j ≤ desc.size
LOCAL: _opid ↦ j
SEP: ∃ op : Operator, op ↦ desc.ops[j] ∧ ((op.type = INSERT ∧ Insert.POST.SEP) ∨
  (op.type = DELETE ∧ Delete.POST.SEP) ∨ (op.type = FIND ∧ Find.POST.SEP))

```

Figure 4.11: Specification for FinishPendingTxn

The rely condition assumes potential interference from all functions in the lock-free transactional list. *FinishPendingTxn* provides the guarantee that the transaction will be aborted if the help stack contains the descriptor. Otherwise, the descriptor is pushed onto the help stack and proceeds to perform the operations in the transaction descriptor. Since the help stack contains the descriptor while the operations are being performed, any cyclic dependencies will be prevented when invoking *FinishPendingTxn*, ensuring lock-free progress.

The loop invariant reflects the invocation of the operations in the transaction descriptor starting from *desc.ops[opid]* and ending at *desc.ops[desc.size]*. Inductive reasoning is used to prove that the loop invariant holds. The existential variable *j* is instantiated with the value *opid*. The existential variable *op* is instantiated with the value *desc.ops[opid]*. The value of *op.type* will decide which function (Insert, Delete, or Find) to invoke. In order to prove a function call, the input parameters provided to the function call in the framework

must demonstrate that the current precondition implies the precondition of the function, separated by a Frame assertion on the free resources [3].

It now must be shown that the postcondition of the function call holds in order to prove that the precondition implies the loop invariant. For each function, the postcondition specifies that the function will complete without an abort given that no conflict is detected. If a conflict is detected, the function that detects the conflict will set out to help the conflicting transaction by calling *FinishPendingTxn*. The postcondition of *FinishPendingTxn* specifies that it will either 1) abort due to a cyclic dependency, or 2) push the descriptor onto the help stack until it finishes the transaction it started to help. In either case, the postcondition of the function (Insert, Delete, or Find) holds as provided by the guarantee condition of *FinishPendingTxn*. Therefore, the precondition implies the loop invariant.

A symbolic execution is performed on the loop body using the forward tactic until the end of the loop body is reached. It must be shown that the loop body implies the loop invariant. The existential variable j is instantiated with the value $j+1$ to show that the loop invariant holds for all values of j . The variable op is instantiated with the value $desc.ops[j+1]$. The same logic used to prove that the precondition implies the loop invariant may be used to prove that the loop body implies the loop invariant. By Theorem 3.0.1, *FinishPendingTxn* is lock-free.

Wait-Free Queue

The wait-free queue [57] ensures wait-free progress by requiring each thread to check its designated location in the *helpRecords* array to determine if it is required to help a thread prior to starting its own operation. The wait-free queue therefore implements the wait-free algorithm of Figure 3.26, allowing for the application of Theorem 3.0.2 to prove

wait-freedom. The shared resources for the wait-free queue are provided in Figure 4.12. Resource $r1$ is the *state* array, where each cell i of the array stores information regarding the operation that thread i needs help performing. Resource $r2$ is the *helpRecords* array, where each cell i of the array stores information regarding the thread to be helped by thread i . Resource $r3$ is the tail pointer, where a *NULL* value for the tail indicates an empty queue, and a non-*NULL* value for the tail indicates that the queue is not empty and the tail's next pointer is *NULL*. Resource $r4$ is the head pointer, where a *NULL* value for the head indicates an empty queue, and a non-*NULL* value for the head indicates that the queue is not empty.

$$\begin{aligned}
r1 &:= \exists \textit{state} : \textit{list OpDesc}, \textit{state} \mapsto \textit{state} \\
r2 &:= \exists \textit{helpRecords} : \textit{list HelpRecords}, \textit{helpRecords} \mapsto \textit{helpRecords} \\
r3 &:= \exists \textit{last} : \textit{Node}, \textit{tail} \mapsto \textit{last} \wedge (\textit{last} \mapsto \textit{NULL} \wedge \textit{emp}) \vee \\
&(\textit{last} \neq \textit{NULL} \wedge \exists \textit{next} : \textit{Node}, (\textit{last} \mapsto \textit{next} \wedge \textit{next} \mapsto \textit{NULL})) \\
r4 &:= \exists \textit{first} : \textit{Node}, \textit{head} \mapsto \textit{first} \wedge (\textit{first} \mapsto \textit{NULL} \wedge \textit{emp}) \\
&\vee (\textit{first} \neq \textit{NULL} \wedge \exists \textit{next} : \textit{Node}, (\textit{first} \mapsto \textit{next}))
\end{aligned}$$

Figure 4.12: Shared Resources for Wait-Free Queue

The specification for *help_enq* is shown in Figure 4.13. The postcondition specification of the shared state reflects that the last node's next pointer is updated to $\textit{state}'[\textit{tid}].\textit{node}$. The variable \textit{state}' is used because the *state* array may change at any moment due to interference by the other threads. When the last node's next pointer is successfully updated, the shared state includes changes by *help_finish_enq* to change the tail to point to $\textit{state}'[\textit{tid}].\textit{node}$. The rely condition of *help_enq* includes functions that modify the *state* array and functions that modify the tail.

$I := r4$
PRE: **LOCAL:** $\exists tid : int, \exists phase : int, _tid \mapsto tid, _phase \mapsto phase$
SEP: $r1 * r2 * r3$
POST: **SEP:** $\exists last : Node, \exists state' : list OpDesc, (last \mapsto _, state'[tid].node) \wedge (state'[tid].node \neq NULL \wedge (state'[tid].node \mapsto _, NULL)) * _state \mapsto state' * help_finish_enq.POST.SEP$
 $R := help_if_needed.G \wedge help_finish_enq.G \wedge wf_deq.G \wedge help_deq.G \wedge help_finish_deq.G \wedge fix_tail.G \wedge CompareAndSwap.G$
 $G := help_enq.PRE.SEP \ltimes help_enq.POST.SEP$
LOOP INVARIANT: $\exists tid' : int,$
PROP: $0 \leq tid' \leq NUM_THRDS$
LOCAL:
SEP: $\exists helpRecords' : list HelpRecords, \exists last : Node, \exists next : Node,$
 $((helpRecords'[tid'].nextCheck \mapsto helpRecords[tid'].nextCheck - 1) \vee ((help_enq.POST.SEP \vee help_deq.POST.SEP) * helpRecords'[tid'].curTid \mapsto (helpRecords[tid'].curTid + 1) \% NUM_THRDS) * last \mapsto next) \vee$
 $(\exists state' : list OpDesc, _state \mapsto state' * ((last \mapsto _, state'[tid'].node) \wedge (state'[tid'].node \neq NULL \wedge (state'[tid'].node \mapsto _, NULL)))) * help_finish_enq.POST.SEP$

Figure 4.13: Specification for `help_enq`

The loop invariant specifies that there exists some thread tid' such that it either updates the last node's next pointer when invoking `enq`, or when invoking `help_enq`. If the update occurs during `enq`, then the thread to succeed must have either decremented `helpRecords[tid'].nextCheck`, or helped an operation that needed help (enqueue or dequeue) and incremented `helpRecords[tid'].curTid`. Otherwise, if the update occurs during `help_enq`, then thread tid' will have updated the last node's next pointer to point to `state'[tid'].node` and updated the tail to point to `state'[tid'].node` by calling `help_finish_enq`. The existential variable tid' is instantiated with the current thread id TID to show that the precondition of the whole loop body implies the loop invariant. After a symbolic execution through the loop body, the existential variable tid' is instantiated with $TID + 1$ to show that the loop body implies the loop invariant for any thread that succeeds at applying CAS to the last node's next pointer. By Theorem 3.0.2, `help_enq` is wait-free.

Transactional Merging

The performance of transactional merging is evaluated for the set and map abstract data types. For the set type, transactional merging is applied to a lock-free transactional list [107] and a lock-free transactional red-black tree and compared against LFTT. A micro-benchmark is used to evaluate performance for a write-dominated workload (50% INSERT, 50% DELETE) and mixed workload (33% INSERT, 33% DELETE, 34% FIND). This evaluation method [40, 107] consists of a tight loop that performs a fixed size transaction comprising a random mixture of INSERT, DELETE, and FIND operations based on the workload type.

For the map type, transactional merging is applied to a lock-free transactional dictionary and integrated into the Silo database [98] by replacing the Masstree [67] indexing structure and epoch-based group commit protocol with the lock-free transactional dictionary. The performance of the database incorporating the lock-free transactional dictionary is compared to the performance of the original Silo database on the TPC-C benchmark. The workloads evaluated on the TPC-C benchmark include write-dominated (100% new-order) and mixed (20% new-order, 20% payments, 20% delivery, 20% order-status, and 20% stock-level). The tests are conducted on two different systems, including a 64-core NUMA system (4 AMD opteron 6272 CPUs with 16 cores per chip @2.1 GHz) and 24-core Dell Precision (Intel Xeon Platinum 8160 @2.1 GHz).

Transactional List and Red-Black Tree

The performance results for the lock-free transactional list and lock-free transactional red-black tree evaluated on the NUMA system and Dell Precision are shown in Figure 4.14 and Figure 4.15, respectively. The performance results graphs for both the NUMA system and Dell Precision use a logarithmic scale for the y-axis. The transaction size is varied

between two and 16. The throughput is measured according to the number of completed operations per second, which is calculated by dividing the product of the number of committed transactions and the transaction size by the execution time (in seconds). Each thread performs 10^4 transactions comprising operations applied to randomly selected keys, where the key range is set to 10^3 . The transactional list and transactional red-black tree using the transactional merging technique are denoted as LFTTM- X , where X is the transaction size. The transactional list and transactional red-black tree using lock-free transactional transformation are denoted as LFTT- X , where X is the transaction size. The general trend observed for both LFTTM and LFTT for all testing scenarios is that the throughput increases as the transaction size decreases. This occurs because a smaller transaction endures a lesser penalty in comparison with a larger transaction due to an abort since fewer operations are discarded. Another trend observed for both LFTTM and LFTT for all testing scenarios is that the throughput scales well up to a transaction size of four. For a transaction size greater than four, LFTTM and LFTT no longer scale well. The probability that two transactions of size m with a key range of n contain a conflicting operation is $1 - \frac{(n-m)^m}{n^m}$, since there are $(n - m)^m$ key selections that do not contain the keys of interest. As the transaction size increases, the probability for a semantic conflict with another transaction increases. This results in poor scalability because 1) the execution times are longer due to the overhead of invoking the helping scheme for semantic conflicts, and 2) fewer transactions are committed due to unresolvable semantic conflicts.

The write-dominated workload on the NUMA system for the transactional list is depicted in the first graph of Figure 4.14. For executions within one CPU chip (16 threads or less), the average speedup of LFTTM over LFTT is 80% (size 2), 79% (size 4), 85% (size 8), and 100% (size 16). In general, the speedup of LFTTM over LFTT increases as the transaction size increases for the transactional list. The reason for this trend is that transactional

merging reduces the number of aborts by merging operations of two transactions with a semantic conflict rather than forcing one of the transactions to abort. The performance improvement obtained by a reduction in aborts increases as the transaction size increases because aborting a transaction forces all operations in the transaction to be discarded. For executions with more than one CPU chip (more than 16 threads), the average speedup is reduced due to the cost of remote memory accesses. The speedup of LFTTM over LFTT averaged over executions with more than 16 threads is 68% (size 2), 54% (size 4), 23% (size 8), and 64% (size 16).

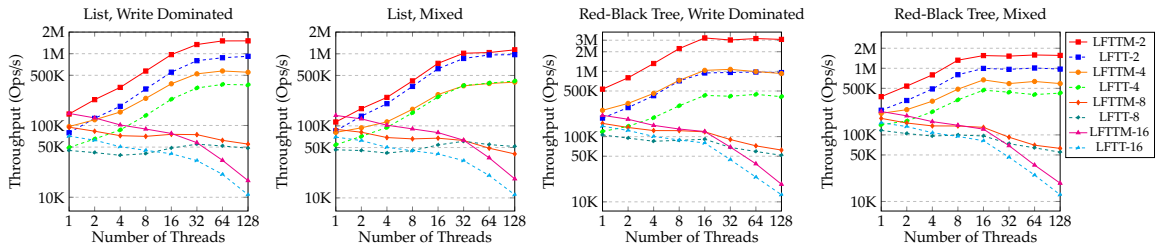


Figure 4.14: List and Red-Black Tree Results on NUMA system

The mixed workload on the NUMA system for the transactional list is depicted in the second graph of Figure 4.14. Since transactional merging only reduces semantic conflicts for INSERT operation pairs or DELETE operation pairs, the addition of the FIND operation reduces the chance on a conflict to merge operations from 50% to 22% (2 outcomes that can merge out of 9 total outcomes). For executions within one CPU chip (16 threads or less), the average speedup of LFTTM over LFTT is 24% (size 2), 24% (size 4), 61% (size 8), and 100% (size 16). For executions with more than one CPU chip (more than 16 threads), the speedup of LFTTM over LFTT averaged over executions with more than 16 threads is 14% (size 2), and 78% (size 16). LFTTM-4 and LFTTM-8 experience a slowdown when compared to LFTT for executions with more than 16 threads resulting from remote memory accesses.

The transactional red-black tree does not follow the same trends as the transactional list because inserted nodes that cause the red-black tree to become unbalanced require a tree repair. All threads that traverse a node that requires a repair must continuously apply the lock-free repair operation until some thread succeeds. As the transaction size increases, the chance that a thread will need to help complete a pending transaction and traverse a node that needs repairing also increases. Since repairing the tree structure is a computationally expensive sequential bottleneck, performance is degraded for both LFTTM and LFTT as the transaction size increases. Another difference between the transactional red-black tree and transactional list is that the red-black tree has a logarithmic search time, resulting in fewer remote memory accesses than the list. This leads to the red-black tree yielding better consistency with respect to speedup across multiple CPUs on the NUMA system.

The write-dominated workload on the NUMA system for the transactional red-black tree is depicted in the third graph of Figure 4.14. The average speedup of LFTTM compared to LFTT over all executions is 212% (size 2), 134% (size 4), 36% (size 8), and 50% (size 16). As expected, the speedup of LFTTM over LFTT for the red-black tree decreases as the transaction size increases. The mixed workload on the NUMA system for the transactional red-black tree is depicted in the fourth graph of Figure 4.14. The average speedup of LFTTM compared to LFTT over all executions is 60% (size 2), 45% (size 4), 32% (size 8), and 47% (size 16). The speedup of the write-dominated configuration in comparison to the mixed configuration demonstrates potential performance gains of transactional merging for a write-dominated workload.

The write-dominated workload on the Dell Precision for the transactional list is depicted in the first graph of Figure 4.15. The Dell Precision contains 24 cores on a single die, enabling good scalability across all 24 cores. The average speedup of LFTTM compared to LFTT over all executions is 70% (size 2), 75% (size 4), 81% (size 8), and 162% (size

16). Similar to the transactional list performance on the NUMA system, the speedup of LFTTM over LFTT increases as the transaction size increases. The mixed workload on the Dell Precision for the transactional list is depicted in the second graph of Figure 4.15. The average speedup of LFTTM compared to LFTT over all executions is 20% (size 2), 17% (size 4), 31% (size 8), and 136% (size 16). The reduced speedup of LFTTM over LFTT for the mixed workload is due to FIND reducing the opportunity for performance gains by merging operations.

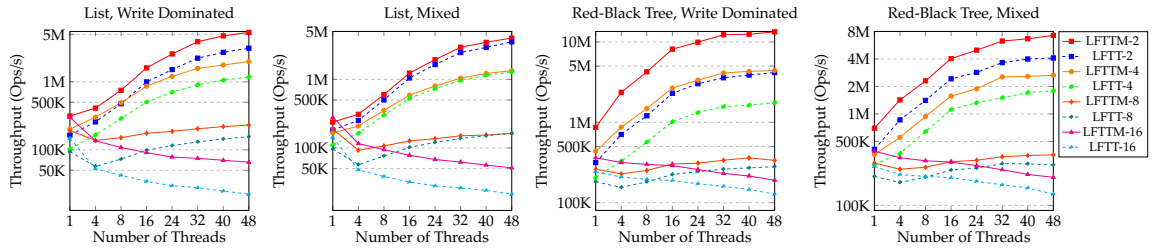


Figure 4.15: List and Red-Black Tree Results on Dell Precision

The write-dominated workload on the Dell Precision for the red-black tree is depicted in the third graph of Figure 4.15. The average speedup of LFTTM compared to LFTT over all executions is 229% (size 2), 154% (size 4), 34% (size 8), and 50% (size 16). The speedup of LFTTM over LFTT decreases as the transaction size increases due to the lock-free tree repair operation. The mixed workload on the Dell Precision for the transactional red-black tree is depicted in the fourth graph of Figure 4.15. The average speedup of LFTTM compared to LFTT over all executions is 70% (size 2), 47% (size 4), 28% (size 8), and 48% (size 16). A small speedup reduction is observed for the mixed workload in comparison to the write-dominated workload due to the additional FIND operation.

Transactional Dictionary

The NUMA system and Dell Precision performance results for the TPC-C benchmark are shown in Figure 4.16 and Figure 4.17, respectively. The TPC-C benchmark assigns customers to a set of districts within a local warehouse. Clients place orders as a transaction with either a local warehouse or remote warehouse. The transactions are executed in the database by worker threads, where an increase in workers causes an increase in contention. The database size is set to four warehouses. The throughput is measured according to the number of completed transactions per second. The lock-free transactional dictionary using the transactional merging technique is denoted as LFTTM. The lock-free transactional dictionary using lock-free transactional transformation is denoted as LFTT. The Silo database is denoted as Silo.

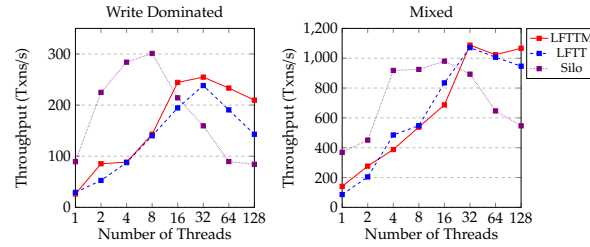


Figure 4.16: TPC-C Benchmark on NUMA system

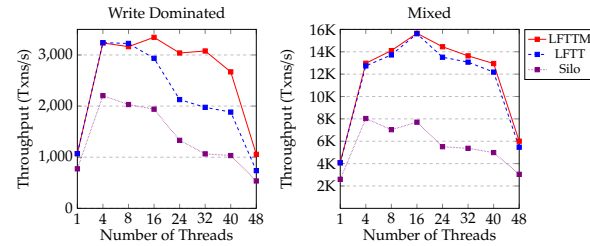


Figure 4.17: TPC-C Benchmark on Dell Precision

The general trend observed for LFTTM, LFTT, and Silo for all testing scenarios on the

NUMA system is that Silo outperforms LFTTM and LFTT for 16 threads or less. This occurs because Silo does not experience read/write conflicts until the number of threads is 16 or higher. The increase in aborts due to read/write conflicts hurts Silo's throughput in comparison to LFTTM and LFTT because both LFTTM and LFTT are designed to avoid read/write conflicts. On the Dell Precision, LFTTM and LFTT outperform Silo for all testing scenarios. This performance gain for LFTTM and LFTT is likely attributed to improved execution times for a thread's access of another thread's transaction descriptor when all cores are located on a single die. The executions with thread counts higher than 24 are no longer fully concurrent, which causes a drop in performance for LFTTM, LFTT, and Silo.

The write-dominated workload on the NUMA system for the TPC-C benchmark is depicted in the first graph of Figure 4.16. For executions within one CPU chip (16 threads or less), the average speedup of LFTTM over LFTT is 16%, while no speedup is obtained for LFTTM over Silo. However, the speedup averaged over executions with more than 16 threads is 25% for LFTTM over LFTT and 123% for LFTTM over Silo. The mixed workload on the NUMA system for the TPC-C benchmark is depicted in the second graph of Figure 4.16. For executions within one CPU chip (16 threads or less), the average speedup of LFTTM over LFTT is 12%, while no speedup is obtained for LFTTM over Silo. The speedup averaged over executions with more than 16 threads is 5% for LFTTM over LFTT and 58% for LFTTM over Silo. LFTTM obtains the highest speedup over LFTT and Silo for the write-dominated workload.

The write-dominated workload on the Dell Precision for the TPC-C benchmark is depicted in the first graph of Figure 4.17. The speedup averaged over all executions is 24% for LFTTM over LFTT and 98% for LFTTM over Silo. The mixed workload on the Dell Precision for the TPC-C benchmark is depicted in the second graph of Figure 4.17.

The speedup averaged over all executions is 4% for LFTTM over LFTT and 112% for LFTTM over Silo. LFTTM obtains the highest speedups for the write-dominated workload when compared to LFTT and for the mixed workload when compared to Silo. Although LFTTM theoretically performs best for write-dominated workloads, the mixed workload for TPC-C issues more transactions, providing more opportunities for performance gains by reducing aborted transactions.

CHAPTER 5: CONCLUSION

This dissertation presents tools and techniques for verifying safety and liveness properties for state-of-the-art multiprocessor programs. My correctness verification tools focus on multiprocessor programs that utilize a composition of data structure operations performed in both a non-atomic and atomic manner. My progress verification technique addresses the challenges associated with thread interference introduced by descriptor-based helping mechanisms.

I present CCSpec, the first tool that checks the correctness of a composition of concurrent multi-container operations performed in a non-atomic manner. I develop a lightweight custom specification language that allows the user to define a correctness condition associated with the abstract function layer and a correctness condition associated with the data structure layer. The experimental results demonstrate the practical application of CCSpec and its ability to produce counterexamples for the specified correctness condition at the data structure layer and abstract function layer. CCSpec will allow designers to check the correctness of a composition of data structure operations as a concurrent algorithm undergoes routine maintenance, design modifications, or optimizations such as relaxing the correctness condition with the benefit of an improvement in performance.

I present 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. I accommodate a diverse assortment of transactional correctness conditions by presenting a technique for defining correctness as a happens-before relation. Since the 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. I account for transactional correctness conditions that do not require a total order on a transactional execution, such as causal consistency, by maintaining a per-thread transactional happens-before graph. The case studies demonstrate the practical application of TxC-ADT to check the correctness of cutting-edge transactional data structures.

I present the first methodology for verifying progress guarantees for descriptor-based non-blocking algorithms. Previous related techniques have assumed that the ability of a thread to exit a CAS-based loop implies lock-freedom and that wait-freedom is a thread local property. The methodology advances the existing approaches for progress verification by presenting a technique for formally reasoning about the helping mechanisms required to preserve the designed progress guarantee. The technique includes a functional specification that defines the state of the auxiliary structure of descriptor objects in the loop invariant. To enable the semi-automatic verification of the specifications, I implement a framework that extends the Verified Software Toolchain to accommodate Local-Rely-Guarantee reasoning. I demonstrate the effectiveness of the methodology by formally verifying progress for a lock-free transactional list and a wait-free queue.

I introduce transactional merging, a technique that relaxes the semantic conflict resolution of transactional data structures by merging conflicting operations of transactions to reduce aborts. A function is provided that enables the designer to specify which semantic conflicts are eligible to be eliminated to allow transactional merging to be configured such that it meets the needs of the system. Transactional merging guarantees strict serializability

through a strategy that will undo the effects of partially merged operations given that the transaction attempting to merge operations aborts. The experimental evaluation demonstrates that transactional merging achieves an average speedup of up to 162% over LFTT for the lock-free transactional linked list and 229% over LFTT for the lock-free transactional red-black tree. To demonstrate a practical application, transactional merging is applied to a lock-free transactional dictionary and integrated into the Silo database by replacing the Masstree indexing structure and epoch-based group commit protocol with the lock-free transactional dictionary. Transactional merging achieves an average speedup of up to 123% over the Masstree indexing used in the Silo database and 25% over LFTT when evaluated on the TPC-C benchmark.

APPENDIX A: CORRECTNESS OF CCSPEC

The correctness discussions for CCSpec’s general approach are adapted from Peterson et al. [81]. A formal definition of commutativity for data structure methods and abstract functions is provided as follows.

Definition A.0.1. Two methods m_1 and m_2 *commute* if for all histories h , if $h \cdot m_1$ and $h \cdot m_2$ are both legal, then $h \cdot m_1 \cdot m_2$ and $h \cdot m_2 \cdot m_1$ are both legal and define the same abstract state.

Definition A.0.2. Two abstract functions f_1 and f_2 *commute* if for all histories h , if $h \cdot f_1$ and $h \cdot f_2$ are both legal, then $h \cdot f_1 \cdot f_2$ and $h \cdot f_2 \cdot f_1$ are both legal and define the same abstract state.

Theorem A.0.1. Let method (or abstract function) i and method (or abstract function) j be commutative and allowed to be reordered according to the happens-before graph. Let h be all possible histories generated by a topological sort of the happens-before graph. Algorithm 6 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 method (or abstract function) i and method (or abstract function) j commute and false otherwise. Let *reorder_matrix* be a boolean two-dimensional matrix where position (i, j) is true if method (or abstract function) i and method (or abstract function) j have no ordering constraints and false if method (or abstract function) i and method (or abstract function) j are ordered by the happens-before graph. Let L be a partial list of methods or abstract functions that are sorted according to the happens-before graph. Let N be a list of all methods or abstract functions with no incoming edges. Let $n \in N$ be a method or abstract function that is under consideration for being amended to the end of L . By Definition A.0.1 and Definition A.0.2, if $L.back()$ and n commute, then these

methods or abstract functions executed in either order will yield the same abstract state. If $L.back()$ and n are allowed to be reordered according to the happens-before graph, then either $L.back() \cdot n$ or $n \cdot L.back()$ can be pruned from the search space if $L.back()$ and n commute. If pruning is possible, arbitrarily choose to explore the ordering in which method id n is greater than method 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 method (or abstract function) i have a smaller id than method (or abstract function) j . Since the if-statement on line 6.2 will resolve to true if $n < L.back()$, $L.back()$ and n commute, and $L.back()$ and n can be reordered according to the happens-before graph, the history $h \cdot i \cdot j$ will be explored and the `PrunedRecTopologicalSort` call will terminate for $h \cdot j \cdot i$. \square

Theorem A.0.2. `RecTopologicalSort` returns a set S of all legal sequential histories defined by the happens-before graph.

Proof. Let N be a list of all methods or abstract functions with no incoming edges in the happens-before graph. Any selection of $n \in N$ will yield a valid topological sort of the happens-before graph. The foreach-statement on line 5.5 calls `PrunedRecTopologicalSort` with the parameter $n \in N$. Since L is initially empty, pruning is not yet possible so n is amended to the back of L and removed from N . All methods or (abstract functions) m with an edge from n to m are removed from the happens-before graph on line 6.8. If m has no incoming edges, it is amended to N on line 6.10. Any selection of $n' \in N$ will yield a valid topological sort of the happens-before graph. The foreach-statement on line 6.11 calls `PrunedRecTopologicalSort` with the parameter $n' \in N$. Since L is not empty, n' may possibly be pruned from the recursive topological sort on line 6.2. By Theorem A.0.1, `PrunedRecTopologicalSort` will only explore one ordering of commutative methods or abstract functions that are allowed to be reordered according to the happens-before graph.

Given that n' is not pruned from the recursive topological sort, n' is amended to the back of L and removed from N . The recursive call terminates when N is empty and amends the topological sort L to S on line 6.14. 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 A.0.1, Definition A.0.1, and Definition A.0.2 set S will contain all legal sequential histories defined by the happens-before graph. \square

Theorem A.0.3. (Soundness) Let h be a concurrent history of implementation X . If h does not satisfy the specified correctness condition then $\text{IsHistoryCorrect}(h)$ (Algorithm 2) returns false.

Proof. By Theorem A.0.2, $\text{RecTopologicalSort}$ returns a set S of all legal sequential histories defined by the happens-before graph. The foreach-statement on line 2.4 iterates through every legal sequential history in S . The foreach-statement on line 2.7 iterates through each method or abstract function called in legal sequential history $s \in S$. For each method or abstract function the observed output from the concurrent history is amended to the list *concurrent_output* on line 2.8. The function pointer associated with the method or abstract function is invoked on line 2.9 to generate the sequential output, which is amended to the list *sequential_output* on line 2.10. If the if-statement on line 2.11 evaluates to true, then concurrent history h is equivalent to a legal sequential history generated from the happens-before graph. Since the happens-before graph represents the allowable orderings of the methods or abstract functions according to the specified correctness condition, concurrent history h satisfies the specified correctness condition and $\text{IsHistoryCorrect}(h)$ returns true. If the end of the for-loop on line 2.4 is reached, concurrent history h is not equivalent to any legal sequential history generated from the happens-before graph. Concurrent history h does not satisfy the specified correctness condition and $\text{IsHistoryCorrect}(h)$ returns false. Therefore, the theorem holds. \square

Theorem A.0.4. (Completeness) Let H be the set of concurrent histories generated from a unit test m of implementation X . If for any arbitrary $h \in H$ `IsHistoryCorrect(h)` (Algorithm 2) returns false, then implementation X does not satisfy the specified correctness condition and `IsUnitTestCorrect` (Algorithm 3) returns false.

Proof. If there exists some concurrent history $h \in H$ such that `IsHistoryCorrect(h)` returns false, then by Theorem A.0.3, 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 *outcome* on line 3.6 is set to false. Implementation X does not satisfy the specified correctness condition and `IsUnitTestCorrect` (Algorithm 3) returns false. Therefore, the theorem holds. \square

APPENDIX B: CORRECTNESS OF TXC-ADT

A formal definition of commutativity between transactions is provided as follows.

Definition B.0.1. 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 B.0.1. 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 6 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 B.0.1, 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 6.2 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 $h \cdot j \cdot i$. \square

Theorem B.0.2. `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 5.5 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 6.8. If m has no incoming edges, it is amended to N on line 6.10. Any selection of $n' \in N$ will yield a valid topological sort of the happens-before graph. The foreach-statement on line 6.11 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 6.2. By Theorem B.0.1, `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 topological 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 6.14. 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 B.0.1 and Definition B.0.1, set S will contain all legal sequential histories defined by the transactional happens-before graph. \square

Theorem B.0.3. (Soundness) Let h be a concurrent history of implementation X . If h does not satisfy the specified correctness condition then `IsHistoryCorrect(h)` (Algorithm 7) returns false.

Proof. By Theorem B.0.2, `RecTopologicalSort` returns a set S of all legal sequential histories defined by the transactional happens-before graph. The `foreach`-statement on line 7.4 iterates through every legal sequential history in S . The `foreach`-statements on line 7.7 and line 7.8 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 7.9. The method's function pointer is invoked on line 7.10 to generate the sequential output, which is amended to the list *sequential_output* on line 7.11. Aborted transactions are accounted for by invoking the inverse method's function pointer on line 7.14. If the `if`-statement on line 7.15 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 7.4 is reached, `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. \square

Theorem B.0.4. (Completeness) Let H be the set of concurrent histories generated from a unit test m of implementation X . If for any arbitrary $h \in H$ `IsHistoryCorrect(h)` (Algorithm 7) returns false, then implementation X does not satisfy the specified correctness condition and `IsUnitTestCorrect` (Algorithm 8) returns false.

Proof. If there exists some concurrent history $h \in H$ such that `IsHistoryCorrect(h)` returns false, then by Theorem B.0.3, 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 *outcome* is set to false on line 8.6. Implementation *X* does not satisfy the specified correctness condition and `IsUnitTestCorrect` (Algorithm 8) returns false. Therefore, the theorem holds. \square

APPENDIX C: CORRECTNESS OF TRANSACTIONAL MERGING

The correctness statements, provided in terms of the map abstract data type, show that transactional merging is 1) strictly serializable and 2) lock-free when applied to LFTT.

Definitions

A *method call* is a pair consisting of an invocation and matching response [46]. An *event* is 1) a change in the status of a transaction including transaction-begin, commit, or abort, or 2) a change in the status of a method including an invocation or response. A *history* is a finite series of instantaneous events [46]. An *object subhistory*, denoted $h|O$ is a subsequence of the events of h , restricted to an object O [46]. A *transaction subhistory*, denoted $h|T$ is a subsequence of the events of h , restricted to a transaction T [44]. A history is *legal* if each object subhistory within each transaction subhistory is legal for that object [46].

Definition C.0.1. A history h is *strictly serializable* if the subsequence of h consisting of all events of committed transactions is equivalent to a legal history in which these transactions execute sequentially in the order they commit [80].

Definition C.0.2. Two method calls I, R and I', R' *commute* if for all histories h , if $h \cdot I \cdot R$ and $h \cdot I' \cdot R'$ are both legal, then $h \cdot I \cdot R \cdot I' \cdot R'$ and $h \cdot I' \cdot R' \cdot I \cdot R$ are both legal and define the same abstract state.

The commutativity specification for map operations is the following, where $Op/false$ indicates operation Op returns false.

$$\begin{aligned}
& \text{INSERT}(x, A) \leftrightarrow \text{INSERT}(y, B), x \neq y \\
& \text{DELETE}(x) \leftrightarrow \text{DELETE}(y), x \neq y \\
& \text{INSERT}(x, A) \leftrightarrow \text{DELETE}(y), x \neq y \\
& \text{PUT}(x, A) \leftrightarrow \text{INSERT}(y, B), x \neq y \\
& \text{PUT}(x, A) \leftrightarrow \text{DELETE}(y), x \neq y \\
& \text{PUT}(x, A) \leftrightarrow \text{PUT}(y, B), x \neq y, \text{ or } x = y \text{ and } A = B \\
& \text{GET}(x) \leftrightarrow \text{INSERT}(x, A)/\text{false} \leftrightarrow \text{DELETE}(x)/\text{false} \leftrightarrow \text{PUT}(x, A)/\text{false}
\end{aligned} \tag{C.1}$$

Definition C.0.3. For a history h and any given invocation I and response R , let I^{-1} and R^{-1} be the inverse invocation and response. Then I^{-1} and R^{-1} are the *inverse operations* of I and R such that the state reached after the history $h \cdot I \cdot R \cdot I^{-1} \cdot R^{-1}$ is the same as the state reached after history h .

Rules

Any software transactional memory system that obeys the following correctness rules is strictly serializable [44].

Rule 1. Linearizability: For any history h , two concurrent invocations I and I' must be equivalent to either the history $h \cdot I \cdot R \cdot I' \cdot R'$ or the history $h \cdot I' \cdot R' \cdot I \cdot R$.

Rule 2. Commutativity Isolation: For any non-commutative method calls $I_1, R_1 \in T_1$ and $I_2, R_2 \in T_2$, either T_1 commits or aborts before any additional method calls in T_2 are invoked, or vice-versa.

Rule 3. Compensating Actions: For any history h and transaction T , if $\langle T \text{ aborted} \rangle \in h$, then it must be the case that $h|T = \langle T \text{ init} \rangle \cdot I_0 \cdot R_0 \cdots I_i \cdot R_i \cdot \langle T \text{ aborted} \rangle \cdot I_i^{-1} \cdot R_i^{-1} \cdots I_0^{-1} \cdot R_0^{-1} \cdot \langle T \text{ aborted} \rangle$ where i indexes the last successfully completed method call.

Strict Serializability and Recovery

It is now shown that transactional merging satisfies the correctness rules required to guarantee strict serializability. The concrete state of a map is denoted as a node set N . The abstract state observed by transaction T_i is denoted as $S_i = \{(n.key, n.info.val) | n \in N \wedge \text{IsKeyPresent}(n.info, desc_i)\}$, where $desc_i$ is the descriptor of T_i .

A concurrent method is linearizable if it appears to take effect instantaneously at some point between the invocation and response. A *linearization point* is the atomic statement that finitely decides the result of the method. A *state-read* point is the atomic statement where the map is read, which determines the outcome of the linearization point.

Lemma 1. The map operations INSERT, DELETE, PUT, and GET are linearizable, satisfying Rule 1.

Proof. For the transformed INSERT operation, the execution is divided into two code paths by the condition check on line 13.6. The code path on line 13.7 updates the existing node's logical status. If UPDATEINFO fails on line 12.20 or line 12.23, no write operation will be performed to change the logical status of the node. The state-read point for the failure case on line 12.20 occurs when the previous transaction status is read from *oldinfo.desc.status* on line 11.5, or when the previous transaction status is read from *oldinfo.desc.status* on line 10.4. The state-read point for the failure case on line 12.23 occurs when the current transaction status is read from *info.desc.status* on line 12.22. The abstract states S' observed by all transactions immediately after the reads are unchanged, $\forall i, S'_i = S_i$. The linearization point for a successful logical status update is when the CAS operation on line 12.24 succeeds. The abstract states S' observed by the transactions T_d executing this operation immediately after CAS is $i = d \implies S'_i = (S_i \setminus (n.key, n.info.val)) \cup (n.key, val)$. For all other

transactions $i \neq d \implies S'_i = S_i$. In all cases, the update of the abstract states conforms to the sequential specification of the INSERT operation. The code path for adding linkage to a new node if a node with the key of interest does not already exist in the list (line 13.12) is linearizable because the Do_INSERT function in the base data structure is linearizable.

The same reasoning can be applied to the transformed DELETE, PUT, and GET operations because they share the same logical status update procedure as INSERT. \square

Lemma 2. The conflict resolution policy in transactional merging satisfies commutativity isolation as defined in Rule 2.

Proof. By Equation C.1, two map operations commute if they access different keys. The one-to-one mapping from nodes to keys is formally stated as $\forall n_x, n_y \in N, x \neq y \implies n_x \neq n_y \implies n_x.key \neq n_y.key$. This implies that two map operations commute if they access different nodes. Let T_1 denote a transaction that accesses node n_1 , where $n.info.desc = desc_1 \wedge desc_1.status = Active$. If another transaction T_2 accesses n_1 , it must perform EXECUTEOPS for T_1 on line 12.7. Upon returning from EXECUTEOPS, T_1 will either commit or abort. It is therefore ensured that $desc_1.status = Committed \vee desc_1.status = Aborted$ before T_2 proceeds. \square

Lemma 3. When a transaction aborts, the logical rollback mechanism employed by transactional merging is equivalent to performing the inverse operations of all computed operations, satisfying Rule 3.

Proof. Let T denote a transaction that executes the operations $I_0 \cdot R_0 \cdots I_i \cdot R_i$ on nodes $n_0 \cdots n_i$ and then aborts. Let S_0 denote the abstract state immediately before I_0 . By Rule 3, T must execute the inverse operations of the successful method calls $I_i^{-1} \cdot R_i^{-1} \cdots I_0^{-1} \cdot R_0^{-1}$

after those method calls have succeeded. This is equivalent to requiring that the abstract state be restored to its original state S_0 .

When T aborts, the logical rollback mechanism updates T 's transaction descriptor status from Active to Aborted using a single CAS. The `IsKeyPresent` function ensures that for each node n_x in $n_0 \cdot \dots \cdot n_i$, the next operation that accesses n_x will interpret the current abstract state S_i to be equal to S_0 . A proof by cases is used for `INSERT`, `DELETE`, `PUT`, and `GET` that every possible operation $I_x \cdot R_x$ will interpret the current abstract state S_i to be equal to S_0 . For all cases, the value of n_x can be correctly recovered by reading *info.oldval*, which is set on line 12.11 or line 12.13 prior to the CAS operation.

INSERT. There are two cases for an `INSERT($n_x.key$, $n_x.info.val$)` call. In the first case, $(n_x.key, val) \notin S_0$, where *val* is any arbitrary value. The `INSERT` method inserts a new node n_x into the map (line 13.12), which has a transaction descriptor field associated with transaction T , or `INSERT` updates the existing node's transaction descriptor field to point to T (line 13.7). Then T aborts at some point after R_x , so T 's transaction descriptor status is set to Aborted. Any operation that accesses n_x will read the transaction descriptor field of n_x , observe T 's descriptor status as Aborted, and logically interpret that $(n_x.key, val) \notin S_i$ (line 11.14). Therefore, $S_i = S_0$.

In the second case, $(n_x.key, n_x.info.oldval) \in S_0$. If $n_x.info.desc.ops[n_x.info.opid].type = \text{Insert}$, then `INSERT` will merge its operation with the prefix transaction's operation associated with the existing node by setting *n.info.merge* to true (line 12.21) and change the existing node's transaction descriptor field to point to T (line 13.7). Then T aborts at some point after R_x , so T 's transaction descriptor status is set to Aborted. Any operation that accesses n_x will read the transaction descriptor field of n_x , observe T 's descriptor status as Aborted, and logically interpret that $(n_x.key, n_x.info.oldval) \in S_i$ (line 11.14), since *info.merge* is set to true.

Therefore, $S_i = S_0$. If $n_x.info.desc.ops[n_x.info.opid].type \neq Insert$, then merging operations is not possible and INSERT does not update the existing node's NODEINFO. Then T aborts at some point after R_x , so T 's transaction descriptor status is set to Aborted. Since the existing node's NODEINFO was not changed, the abstract state of n_x is unaffected, so $S_i = S_0$.

DELETE. There are two cases for a DELETE($n_x.key$) call. In the first case, $(n_x.key, n_x.info.oldval) \in S_0$. The DELETE method updates the existing node's transaction descriptor field to point to T (equivalent to line 13.7 of INSERT). Then T aborts at some point after R_x , so T 's transaction descriptor status is set to Aborted. Any operation that accesses n_x will read the transaction descriptor field of n_x , observe T 's descriptor status as Aborted, and logically interpret that $(n_x.key, n_x.info.oldval) \in S_i$ (line 11.14). Therefore, $S_i = S_0$.

In the second case, $(n_x.key, val) \notin S_0$, where val is any arbitrary value. If $n_x.info.desc.ops[n_x.info.opid].type = Delete$, then DELETE will merge its operation with the prefix transaction's operation associated with the existing node by setting $n.info.merge$ to true (line 12.21) and change the existing node's transaction descriptor field to point to T (equivalent to line 13.7 of INSERT). Then T aborts at some point after R_x , so T 's transaction descriptor status is set to Aborted. Any operation that accesses n_x will read the transaction descriptor field of n_x , observe T 's descriptor status as Aborted, and logically interpret that $(n_x.key, val) \notin S_i$ (line 11.14), since $info.merge$ is set to true. If $n_x.info.desc.ops[n_x.info.opid].type \neq Delete$, then merging operations is not possible and DELETE does not update the existing node's NODEINFO. Then T aborts at some point after R_x , so T 's transaction descriptor status is set to Aborted. Since the existing node's NODEINFO was not changed, the abstract state of n_x is unaffected, so $S_i = S_0$.

PUT. There are two cases for a PUT($n_x.key, n_x.info.val$) call. In the first case, $(n_x.key, n_x.info.oldval) \in S_0$. The PUT method updates the existing node's transaction descriptor

field to point to T (equivalent to line 13.7 of INSERT). Then T aborts at some point after R_x , so T 's transaction descriptor status is set to Aborted. Any operation that accesses n_x will read the transaction descriptor field of n_x , observe T 's descriptor status as Aborted, and logically interpret that $(n_x.key, n_x.info.oldval) \in S_i$ given that $info.oldval \neq \text{NULL}$ (line 11.14). The field $info.oldval$ is not NULL if PUT overwrites n_x 's existing value. Therefore, $S_i = S_0$.

In the second case, $(n_x.key, val) \notin S_0$, where val is any arbitrary value. The PUT method inserts a new node n_x into the map (equivalent to line 13.12 of INSERT), which has a transaction descriptor field associated with transaction T , or PUT updates the existing node's transaction descriptor field to point to T (equivalent to line 13.7 of INSERT). Then T aborts at some point after R_x , so T 's transaction descriptor status is set to Aborted. Any operation that accesses n_x will read the transaction descriptor field of n_x , observe T 's descriptor status as Aborted, and logically interpret that $(n_x.key, val) \notin S_i$ (line 11.14) given that $info.oldval = \text{NULL}$. The field $info.oldval$ is NULL if PUT inserts a new node into the map. Therefore, $S_i = S_0$.

GET. There are two cases for a GET($n_x.key$) call. In the first case, $(n_x.key, n_x.info.oldval) \in S_0$. The GET method updates the existing node's transaction descriptor field to point to T (equivalent to line 13.7 of INSERT). Then T aborts at some point after R_x , so T 's transaction descriptor status is set to Aborted. Any operation that accesses n_x will read the transaction descriptor field of n_x , observe T 's descriptor status as Aborted, and logically interpret that $(n_x.key, n_x.info.oldval) \in S_i$ (line 11.14). Therefore, $S_i = S_0$.

In the second case, $(n_x.key, val) \notin S_0$, where val is any arbitrary value. The GET method does not update the existing node's NODEINFO. Then T aborts at some point after R_x , so T 's transaction descriptor status is set to Aborted. Since the existing node's NODEINFO was not changed, the abstract state of n_x is unaffected, so $S_i = S_0$. \square

Theorem C.0.1. For a data structure that is generated using the transactional merging technique, the history of committed transactions is strictly serializable.

Proof. Follow lemmas 1, 2, 3, and the main theorem of Herlihy et al.'s work [44], the theorem holds. □

Progress Guarantees

Transactional merging guarantees lock-free progress because at least one thread makes progress in a finite number of steps by either committing or aborting a transaction. For a system with i threads, the maximum number of active transactions is i . The while loop within EXECUTEOPS is bounded by the maximum number of operations in a transaction (denoted as j), but threads may help complete other pending transactions if a semantic conflict is detected. The bound on the number of recursive helping invocations is the number of active transactions. If a cyclic dependency exists between transactions, the duplicate descriptor will be detected within $i * j$ steps since there are at most i active transactions with at most j operations. A detected cyclic dependency by transaction T will force T to abort. Therefore, the system guarantees that a transaction will either commit or abort in at most $i * j$ steps.

LIST OF REFERENCES

- [1] Afek, Y., Korland, G., and Yanovsky, E. (2010). Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems*, pages 395–410. Springer.
- [2] Alvaro, P., Bailis, P., Conway, N., and Hellerstein, J. M. (2013). Consistency without borders. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 23. ACM.
- [3] Appel, A. W., Dockins, R., Beringer, L., Hobor, A., Dodds, J., Blazy, S., Leroy, X., and Stewart, G. (2014). *Program logics for certified compilers*. Cambridge University Press.
- [4] Aristizábal, A., Bonchi, F., Palamidessi, C., Pino, L., and Valencia, F. (2011). *Deriving Labels and Bisimilarity for Concurrent Constraint Programming*. Berlin, Heidelberg.
- [5] Aspnes, J., Herlihy, M., and Shavit, N. (1994). Counting networks. *Journal of the ACM (JACM)*, 41(5):1020–1048.
- [6] Baek, W., Bronson, N., Kozyrakis, C., and Olukotun, K. (2010). Implementing and evaluating a model checker for transactional memory systems. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 117–126. IEEE.
- [7] Bar-Nissan, G., Hendler, D., and Suissa, A. (2011). A dynamic elimination-combining stack algorithm. In *International Conference On Principles Of Distributed Systems*, pages 544–561. Springer.
- [8] Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al. (1997). The coq proof assistant reference manual: Version 6.1.

- [9] Barrington, A., Feldman, S., and Dechev, D. (2015). A scalable multi-producer multi-consumer wait-free ring buffer. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1321–1328. ACM.
- [10] Bäumlér, S., Schellhorn, G., Tofan, B., and Reif, W. (2011). Proving linearizability with temporal logic. *Formal aspects of computing*, 23(1):91–112.
- [11] Bieniusa, A. and Thiemann, P. (2011). Proving isolation properties for software transactional memory. In *European Symposium on Programming*, pages 38–56. Springer.
- [12] Blundell, C., Lewis, E. C., and Martin, M. M. (2006). Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2).
- [13] Burckhardt, S., Dern, C., Musuvathi, M., and Tan, R. (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*, volume 45, pages 330–340. ACM.
- [14] Chapman, K., Hosking, A. L., and Moss, J. E. B. (2016). Hybrid stm/htm for nested transactions on openjdk. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 51(10):660–676.
- [15] Chapman, K., Hosking, A. L., Moss, J. E. B., and Richards, T. (2014). Closed and open nested atomic actions for java: Language design and prototype implementation. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 169–180. ACM.
- [16] Chen, H., Chen, R., Wei, X., Shi, J., Chen, Y., Wang, Z., Zang, B., and Guan, H. (2017). Fast in-memory transaction processing using rdma and htm. *ACM Transactions on Computer Systems (TOCS)*, 35(1):3.

- [17] Cohen, A., O’Leary, J. W., Pnueli, A., Tuttle, M. R., and Zuck, L. D. (2007). Verifying correctness of transactional memories. In *Formal Methods in Computer Aided Design, 2007. FMCAD’07*, pages 37–44. IEEE.
- [18] Cohen, A., Pnueli, A., and Zuck, L. D. (2008). Mechanical verification of transactional memories with non-transactional memory accesses. In *International Conference on Computer Aided Verification*, pages 121–134. Springer.
- [19] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. (2009). Vcc: A practical system for verifying concurrent c. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer.
- [20] Coleman, J. W. and Jones, C. B. (2007). A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841.
- [21] Cotard, S., Queudet, A., Béchenec, J.-L., Faucou, S., and Trinquet, Y. (2015). Stm-hrt: A robust and wait-free stm for hard real-time multicore embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(4):66.
- [22] da Rocha Pinto, P., Dinsdale-Young, T., and Gardner, P. (2014). Tada: A logic for time and data abstraction. In *European Conference on Object-Oriented Programming*, pages 207–231. Springer.
- [23] da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P., and Sutherland, J. (2016). Modular termination verification for non-blocking concurrency. In *European Symposium on Programming Languages and Systems*, pages 176–201. Springer.
- [24] Dechev, D., Pirkelbauer, P., and Stroustrup, B. (2006). Lock-free dynamically resizable arrays. In *International Conference On Principles Of Distributed Systems*, pages 142–156. Springer.

- [25] Doherty, S., Groves, L., Luchangco, V., and Moir, M. (2013). Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, pages 1–31.
- [26] Dwyer, M. B., Edenhofner, J., Gopalakrishnan, G., Marianiello, A., Luo, Z., Rakamaric, Z., Rogers, M., Siegel, S. F., Zheng, M., and Zirkel, T. K. (2015). The concurrency intermediate verification language reference manual v0. 17.
- [27] Emmi, M., Majumdar, R., and Manevich, R. (2010). Parameterized verification of transactional memories. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2010. PLDI’10*, volume 45, pages 134–145. ACM.
- [28] Feldman, S., Valera-Leon, C., and Dechev, D. (2016). An efficient wait-free vector. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):654–667.
- [29] Feng, X. (2009). Local rely-guarantee reasoning. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, volume 44, pages 315–327. ACM.
- [30] Flanagan, C., Freund, S. N., and Yi, J. (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*, 43(6):293–303.
- [31] Flanagan, C. and Godefroid, P. (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*, volume 40, pages 110–121. ACM.

- [32] Fraser, K. and Harris, T. (2007). Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5.
- [33] Garcia-Molina, H. and Salem, K. (1987). *Sagas*, volume 16. ACM.
- [34] Gotsman, A., Cook, B., Parkinson, M., and Vafeiadis, V. (2009). Proving that non-blocking algorithms don't block. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, volume 44, pages 16–28. ACM.
- [35] Gotsman, A. and Yang, H. (2012). Linearizability with ownership transfer. In *CONCUR 2012 – Concurrency Theory*, pages 256–271. Springer Berlin Heidelberg.
- [36] Guerraoui, R., Henzinger, T. A., Jobstmann, B., and Singh, V. (2008a). Model checking transactional memories. *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008. PLDI'08*, 43(6):372–382.
- [37] Guerraoui, R., Henzinger, T. A., and Singh, V. (2008b). Completeness and nondeterminism in model checking transactional memories. *Lecture Notes in Computer Science*, 5201:21–35.
- [38] Guerraoui, R., Henzinger, T. A., and Singh, V. (2009). Software transactional memory on relaxed memory models. In *CAV*, volume 9, pages 321–336. Springer.
- [39] Guerraoui, R. and Kapalka, M. (2008). On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM.
- [40] Harris, T. (2001). A pragmatic implementation of non-blocking linked-lists. *Distributed Computing*, pages 300–314.

- [41] Hendler, D., Incze, I., Shavit, N., and Tzafrir, M. (2010). Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM.
- [42] Hendler, D., Shavit, N., and Yerushalmi, L. (2004). A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215. ACM.
- [43] Herlihy, M. (1993). A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770.
- [44] Herlihy, M. and Koskinen, E. (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*, pages 207–216. ACM.
- [45] Herlihy, M., Luchangco, V., and Moir, M. (2006). A flexible framework for implementing software transactional memory. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, volume 41, pages 253–262. ACM.
- [46] Herlihy, M. and Shavit, N. (2011). *The art of multiprocessor programming*. Morgan Kaufmann.
- [47] Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492.
- [48] Hoare, C. (1972). Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281.

- [49] Hoffmann, J., Marmar, M., and Shao, Z. (2013). Quantitative reasoning for proving lock-freedom. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 124–133. IEEE.
- [50] Holzmann, G. J. (1997). The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295.
- [51] Hutto, P. W. and Ahamad, M. (1990). Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 302–309. IEEE.
- [52] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. (2011). Verifast: A powerful, sound, predictable, fast verifier for c and java. *NASA Formal Methods*, 6617:41–55.
- [53] Jia, X., Li, W., and Vafeiadis, V. (2015). Proving lock-freedom easily and automatically. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 119–127. ACM.
- [54] Kaiser, G. E. and Pu, C. (1992). Dynamic restructuring of transactions.
- [55] Kaudel, F. J. (1987). A literature survey on distributed discrete event simulation. *ACM SIGSIM simulation digest*, 18(2):11–21.
- [56] Kirsch, C. M., Lippautz, M., and Payer, H. (2013). Fast and scalable, lock-free k-fifo queues. In *Parallel Computing Technologies*, pages 208–223. Springer.
- [57] Kogan, A. and Petrank, E. (2012). A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 47, pages 141–150. ACM.
- [58] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691.

- [59] Leroy, X. (2012). The compcert c verified compiler. *Documentation and user's manual*. INRIA Paris-Rocquencourt.
- [60] Lesani, M. and Palsberg, J. (2011). Communicating memory transactions. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11)*, volume 46, pages 157–168. ACM.
- [61] Liang, H., Feng, X., and Shao, Z. (2014). Compositional verification of termination-preserving refinement of concurrent programs. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 65. ACM.
- [62] Litz, H., Dias, R. J., and Cheriton, D. R. (2015). Efficient correction of anomalies in snapshot isolation transactions. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):65.
- [63] Low, Y., Gonzalez, J. E., Kyrola, A., Bickson, D., Guestrin, C. E., and Hellerstein, J. (2014). Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*.
- [64] Luchangco, V. and Marathe, V. J. (2011a). Revisiting condition variables and transactions. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing, San Jose, CA*.
- [65] Luchangco, V. and Marathe, V. J. (2011b). Transaction communicators: enabling cooperation among concurrent transactions. *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11)*, 46(8):169–178.
- [66] Manovit, C., Hangal, S., Chafi, H., McDonald, A., Kozyrakis, C., and Olukotun, K. (2006). Testing implementations of transactional memory. In *Proceedings of the 15th*

international conference on Parallel architectures and compilation techniques, pages 134–143. ACM.

- [67] Mao, Y., Kohler, E., and Morris, R. T. (2012). Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM.
- [68] Marathe, V. J., Scherer, W. N., and Scott, M. L. (2005). Adaptive software transactional memory. In *International Symposium on Distributed Computing*, pages 354–368. Springer.
- [69] Marathe, V. J., Spear, M. F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W. N., and Scott, M. L. (2006). Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*.
- [70] Michael, M. M. (2004). Scalable lock-free dynamic memory allocation. *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI)*, 39(6):35–46.
- [71] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. (2008). Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, volume 8, pages 267–280.
- [72] Muth, P., Rakow, T. C., Weikum, G., Brossler, P., and Hasse, C. (1993). Semantic concurrency control in object-oriented database systems. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 233–242. IEEE.
- [73] Ni, Y., Menon, V. S., Adl-Tabatabai, A.-R., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., and Shpeisman, T. (2007). Open nesting in software transactional memory.

- In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78. ACM.
- [74] Nodine, M. H. and Zdonik, S. B. (1992). Cooperative transaction hierarchies: Transaction support for design applications. *The VLDB Journal (The International Journal on Very Large Data Bases)*, 1(1):41–80.
- [75] Norris, B. and Demsky, B. (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*, volume 48, pages 131–150. ACM.
- [76] O’hearn, P. W. (2004). Resources, concurrency and local reasoning. In *International Conference on Concurrency Theory*, pages 49–67. Springer.
- [77] O’Leary, J., Saha, B., and Tuttle, M. R. (2009). Model checking transactional memory with spin. In *Distributed Computing Systems, 2009. ICDCS’09. 29th IEEE International Conference on*, pages 335–342. IEEE.
- [78] Oortwijn, W., Blom, S., Gurov, D., Huisman, M., and Zaharieva-Stojanovski, M. (2017). An abstraction technique for describing concurrent program behaviour. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 191–209. Springer.
- [79] Ou, P. and Demsky, B. (2017). Checking concurrent data structures under the c/c++11 memory model. *PPoPP*.
- [80] Papadimitriou, C. H. (1979). The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653.

- [81] Peterson, C. and Dechev, D. (2017). A transactional correctness tool for abstract data types. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(4):37.
- [82] Ramampiaro, H. and Nygård, M. (2004). Cagistrans: Providing adaptable transactional support for cooperative work—an extended treatment. *Information Technology and Management*, 5(1-2):23–64.
- [83] Raynal, M., Thia-Kime, G., and Ahamad, M. (1997). From serializable to causal transactions for collaborative applications. In *EUROMICRO 97. New Frontiers of Information Technology., Proceedings of the 23rd EUROMICRO Conference*, pages 314–321. IEEE.
- [84] Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE.
- [85] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. (2006). Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197. ACM.
- [86] Schmidt-Schauß, M. and Sabel, D. (2013). *Correctness of an STM Haskell implementation*, volume 48. ACM.
- [87] Schwarz, P. M. and Spector, A. Z. (1984). Synchronizing shared abstract types. *ACM Transactions on Computer Systems (TOCS)*, 2(3):223–250.
- [88] Sergey, I., Nanevski, A., and Banerjee, A. (2015a). Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 50, pages 77–87. ACM.

- [89] Sergey, I., Nanevski, A., and Banerjee, A. (2015b). Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, pages 333–358.
- [90] Shacham, O., Bronson, N., Aiken, A., Sagiv, M., Vechev, M., and Yahav, E. (2011). Testing atomicity of composed concurrent operations. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11)*, volume 46, pages 51–64. ACM.
- [91] Shavit, N. and Touitou, D. (1997). Software transactional memory. *Distributed Computing*, 10(2):99–116.
- [92] Siakavaras, D., Nikas, K., Goumas, G., and Koziris, N. (2016). Massively concurrent red-black trees with hardware transactional memory. In *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*, pages 127–134. IEEE.
- [93] Smaragdakis, Y., Kay, A., Behrends, R., and Young, M. (2007). Transactions with isolation and cooperation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA '07)*, volume 42, pages 191–210. ACM.
- [94] Spiegelman, A., Golan-Gueta, G., and Keidar, I. (2016). Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 682–696. ACM.
- [95] Timnat, S., Braginsky, A., Kogan, A., and Petrank, E. (2012). Wait-free linked-lists. In *International Conference On Principles Of Distributed Systems*, pages 330–344. Springer.
- [96] Tofan, B., Bäuml, S., Schellhorn, G., and Reif, W. (2010). Temporal logic verification of lock-freedom. In *Mathematics of Program Construction*, pages 377–396. Springer.

- [97] Treiber, R. K. (1986). *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center.
- [98] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. (2013). Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM.
- [99] Vechev, M., Yahav, E., and Yorsh, G. (2009). Experience with model checking linearizability. In *Model Checking Software*, pages 261–278. Springer.
- [100] Walborn, G. D. and Chrysanthis, P. K. (1995). Supporting semantics-based transaction processing in mobile database applications. In *Reliable Distributed Systems, 1995. Proceedings., 14th Symposium on*, pages 31–40. IEEE.
- [101] Wang, C., Liu, Y., and Spear, M. (2014). Transaction-friendly condition variables. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 198–207. ACM.
- [102] Weikum, G. and Schek, H.-J. (1992). Concepts and applications of multilevel transactions and open nested transactions.
- [103] Wu, P. and Fekete, A. (2003). An empirical study of commutativity in application code. In *Database Engineering and Applications Symposium, 2003. Proceedings. Seventh International*, pages 361–369. IEEE.
- [104] Wu, P., Fekete, A., and Rohm, U. (2008). The efficacy of commutativity-based semantic locking in a real-world application. *IEEE Transactions on Knowledge and Data Engineering*, 20(3):427–431.
- [105] Zhang, D. and Dechev, D. (2016a). An efficient lock-free logarithmic search data

- structure based on multi-dimensional list. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 281–292. IEEE.
- [106] Zhang, D. and Dechev, D. (2016b). A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):613–626.
- [107] Zhang, D. and Dechev, D. (2016c). Lock-free transactions without rollbacks for linked data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 325–336. ACM.
- [108] Zhang, L., Chattopadhyay, A., and Wang, C. (2015). Round-up: runtime verification of quasi linearizability for concurrent data structures. *IEEE Transactions On Software Engineering*, 41(12):1202–1216.